OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Atte Kettunen

TEST HARNESS FOR WEB BROWSER FUZZ TESTING

Master's Thesis
Degree Programme in Computer Science and Engineering
September 2014

Kettunen A. (2014) Test harness for web browser fuzz testing. University of Oulu, Department of Computer Science and Engineering. Master's Thesis, 45 p.

ABSTRACT

Modern web browsers are feature rich software applications available for different platforms ranging from home computers to mobile phones and modern TVs. Because of this variety, the security testing of web browsers is a diverse field of research. Typical publicly available tools for browser security testing are fuzz test case generators designed to target a single feature of a browser on a single platform. This work introduces a cross-platform testing harness for browser fuzz testing, called NodeFuzz. In the design of NodeFuzz, test case generators and instrumentation are separated from the core into separate modules. This allows the user to implement feature specific test case generators and platform specific instrumentations, and to execute those in different combinations. During development, NodeFuzz was tested with ten different test case generators and six different instrumentation modules. Over 50 vulnerabilities were uncovered from the tested web browsers during the development and testing of NodeFuzz.

Keywords: Security testing, testing harness, automated software testing, vulnerability finding, robustness testing.

Kettunen A. (2014) Testausjärjestelmä internet-selainten fuzz-testaukseen. Oulun yliopisto, Tietotekniikan osasto. Diplomityö, 45 s.

TIIVISTELMÄ

Nykyaikaiset internet-selaimet ovat monipuolisia ohjelmistosovelluksia, jotka ovat saatavilla useille eri alustoille, aina tietokoneista ja matkapuhelimista uusimpiin televisioihin. Tästä monimuotoisuudesta johtuen internet-selaimien tietoturvatestaus on monipuolinen tutkimusala. Tyypilliset julkisesti saatavilla olevat työkalut internet-selainten tietoturvatestaamiseen ovat fuzz-menetelmää hyödyntäviä testitapausgeneraattoreita, jotka on suunniteltu jonkin selaimen yksittäisen ominaisuuden testaamiseen yhdellä tietyllä alustalla. Tämä työ esittelee alustariippumattoman testausympäristön internet-selainten fuzz-testaukseen, NodeFuzzin. NodeFuzzissa testitapausgeneraattorit ja instrumentointi ovat irroitettu ytimestä erillisiksi moduuleiksi. Tällöin käyttäjä voi luoda erillisiä kohdennettuja testitapausgeneraattoreita ja alustakohtaisia instrumentointimoduuleja sekä suorittaa testausta moduulien eri yhdistelmillä. Kehitystyön aikana NodeFuzzia testattiin kymmenellä eri testitapausgeneraattorilla ja kuudella eri instrumentointimoduulilla. Kehitystyön ja testaamisen aikana testatuista internet-selaimista paljastui yli 50 uutta haavoittuvuutta.

Avainsanat: Tietoturvatestaus, testausympäristö, automatisoitu ohjelmistotestaus, haavoittuvuuksien etsiminen, luotettavuus testaus.

TABLE OF CONTENT

FOREWORD

When I started working at Oulu University Secure Programming Group, I never thought what the next two years would be like. I have been recognized as one of the top bug reporters in Google Chrome security bounty program, I have presented in conferences and had drinks with some of the people I would never even hoped to see in person. Truly it has been the greatest two years of my life.

I want to thank all the people who have been supporting and guiding me through this journey. Especially I want to thank my fiancé Elisa. I don't know how she has been able to listen me babbling about fuzzing and browsers all this time. Also Aki Helin and Christian Wieser deserve special thanks. Without Aki I would never have started fuzzing browsers and without Christian this thesis would never have been ready. Last but not least, I want to thank my thesis supervisor Juha Röning for his guidance throughout the writing process.

I also have to recognize the Google Chrome and Mozilla Firefox security teams. They have taught me so much and of course the rewards have been a great motivator throughout the whole time.

Oulu, 7th September. 2014

Atte Kettunen

LIST OF ABBREVIATIONS AND SYMBOLS

ASAN	AddressSanitizer
CLI	Command-Line Interface
CGI	Common Gateway Interface
CVE	Common Vulnerabilities and Exposures
DOM	Document Object Model
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
OS	Operating System
OUSPG	Oulu University Secure Programming Group
RAM	Random Access Memory
RFC	Request for Comments
SUT	System Under Test

1. INTRODUCTION

Web browsers have evolved from displaying static text-only content into feature rich software applications [1]. Modern browsers can be used for a multitude of everyday activities, such as, banking, entertainment and real time web applications. Web browsers are also available on many different platforms ranging from home computers to mobile phones and modern TVs. The amount of features embedded into modern-day browsers raises concern about the security of the implementations.

According to computer security and software company Symantec, 351 different security vulnerabilities were reported in popular web browsers in 2011. In 2012, the number of browser vulnerabilities was even higher, totaling to 891. At the same time, the total amount of vulnerabilities reported was 4,814 for the year 2011 and 5,291 for the year 2012 [2]. Considering that the statistics take into account only five popular browsers, the numbers are high when compared with the total amount of vulnerabilities found.

In the context of web browsers, the development of techniques used for vulnerability discovery has been driven forward by bug bounty programs. In these bug bounty programs, monetary rewards ranging from $500 to tens of thousands of dollars are offered for disclosure of previously unknown vulnerabilities to the vendor. Two vendors of popular browsers, Google and Mozilla, have had their bounty programs running for years, Mozilla since 2004 [3] and Google since 2010 [4].

There is a variety of approaches to vulnerability discovery from a piece of software. All approaches have their own advantages and disadvantages, and there is no single method that can reveal all possible vulnerabilities from a given target [5 p. 3]. One of the widely established techniques is fuzz testing [6, 7 p. 24]. Fuzz testing is a broad field of software testing and there is no specific definition for it. It can be described as a black-box or gray-box software testing technique, where malformed, semi-valid or unexpected input is injected into a System Under Test (SUT) through different interfaces, while the SUT is monitored for undesired behavior. [5 p. 22, 7 p. 24]

During the years 2010 and 2011, the Oulu University Secure Programming Group (OUSPG) performed fuzz testing against widely used web browsers [8]. The methods used in the research were efficient, even though they were not specifically designed for browser fuzzing. At the end of the research, it was noted that new vulnerabilities were not found as frequently as at the beginning of the research. Tested browsers were becoming more robust against the techniques used. From that final notion, a research question for this research was formed. How could the framework implemented in the previous research be redesigned to also allow use of browser-domain specific methods in order to enhance the effectiveness of the testing?

1.1. Fuzz testing

In this thesis, the process of fuzz testing is divided into three main parts: test case generation, test case injection and instrumentation of the SUT. Often these parts are bound together to form an automated testing system while utilizing available scripting tools [9], a framework [10, 11] or a harness [12]. Tools implementing some

or all parts of the process are called fuzzers. Figure 1 presents an overview of the process of fuzz testing as described in this thesis.

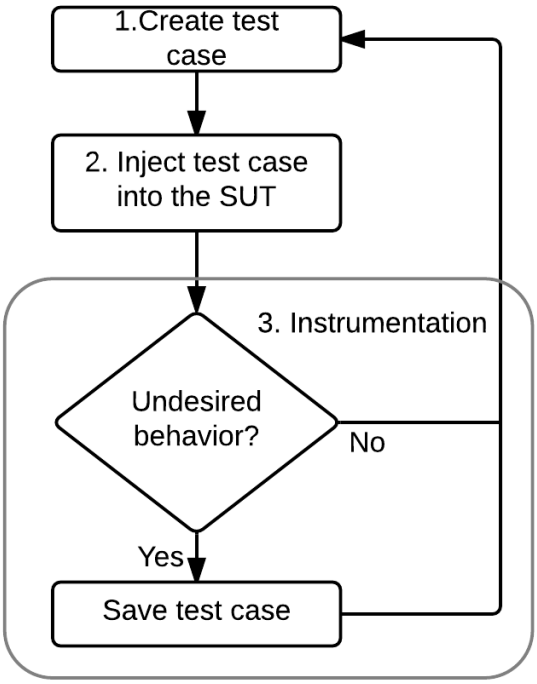

Figure 1 . Process of fuzz testing.

Methods used to test data generation in fuzz testing are typically divided into two groups: generation-based and mutation-based [5 p. 33, 13, 14]. In the mutation-based approach, one or more valid sample inputs are used as a base for the test case generation. Those inputs are mutated to create test cases. Mutations can be simple, like flipping a bit or replacing a character, or more advanced, like detection and manipulation of repeating structures in the data. Simple mutation-based general purpose fuzzers are easy to create and do not necessarily require any prior knowledge about the data they mutate. As a downside, the mutation-based approach is dependent on the quality of the samples provided. Without samples that cover the whole complexity of the target, it is unlikely that a mutation-based fuzzer could achieve high code coverage of the SUT [13]. One example of a mutation-based fuzzer is radamsa [8], a general purpose fuzzing tool containing multiple mutation-algorithms.

In the generation-based approach, known information, like Request for Comments (RFC) or the specification of the data, is used as a basis from which a model of the data is constructed. The model is then used as a base for the test case generation. Although this approach works well on data formats where formal specifications are available, the amount of work needed to understand the specification well enough to create the test case generator can be high. One example of a generation-based fuzzer is jsfunfuzz [15], a fuzzer designed for fuzz testing of JavaScript engines. Jsfunfuzz generates strings with JavaScript syntax, like functions and objects, with occasional syntax errors.

There are also hybrid approaches, which implement techniques from both groups. One example of these hybrid approaches is LangFuzz [16]. LangFuzz is a fuzzer designed for black-box testing of engines based on context-free languages. In

LangFuzz, a hybrid approach was chosen because the purely generation-based approach would require the introduction of generation rules that would bind the LangFuzz into specific languages. The purely mutation-based approach would instead bind LangFuzz to use only the information available in the samples. [16]

Test data injection concerns the process of delivering data from the generator into the SUT. Different types of SUT may have different interfaces that can be used to inject data. As an example, the interface can be a Command Line user Interface (CLI), a Graphical User Interface (GUI), a network protocol or a file. These interfaces can offer access to different features of the system and have some benefits over each other when designing fuzz testing on specific features. As an example, controlling a GUI application tend to be trickier than controlling a CLI application.

Instrumentation means observation of the SUT for unwanted behavior while testing. Instrumentation can be done in several different ways, for example by sending valid test cases between fuzzed inputs, monitoring of system resources, both locally and remotely and changing the way the application executes. [3] Tools like debuggers and memory error detectors can be used to help detect errors that could go unnoticed otherwise. When unwanted behavior is detected, the test data causing the problem, and all collected information, like error-logs, should be saved in some form, so that the behavior can be reproduced later.

When selecting proper instrumentation tools for fuzzing, one must consider the amount of hardware available for testing, the underlying operating system and the types of errors that should be detected. No single tool will work on all platforms and detects all errors.

The process of fuzz testing typically includes execution of millions of test cases, so it is practical to automate all the previously mentioned parts of the process into a constant flow of testing. Automation can be as simple as a shell-script with an infinite loop that executes test case generation, injection and instrumentation until an error is detected and then exits. More advanced systems can include features like automatic updating of the SUT, finding the change that introduced the problem into the SUT, minimization of the test cases causing the error and detection of duplicate findings.

Complete testing frameworks contain a logic to control and automate the whole testing system. General purpose fuzzing frameworks like Peach [10], implement interfaces for test case generators, test case injection and instrumentation. Frameworks like Peach allow users to create multiple different test case injection methods and test case generators that can be used in different combinations, but getting familiar with the provided interfaces can be time consuming.

1.2. Related research

Miller, Fredriksen and So were among the first to research fuzz testing on real life applications in 1990 [9]. The research was conducted, because of an observation, that commonly used UNIX utilities sometimes crashed when used through a noisy dial-up line. The authors implemented a simple tool to create random input strings, that were injected to utility under test via pipeline or pseudo-terminal created with a second tool implemented in the research. The method was able to cause a crash or hang in over 24% of the tested 88 common UNIX utilities. Since then, fuzz testing has been used to test a wide variety of applications including web browsers.

One of the first released browser specific fuzzer, that drew attention, was mangleme [17], a Hypertext Markup Language (HTML) generator, which generates broken HTML strings that are injected into a browser through a Common Gateway Interface (CGI). Mangleme was released in 2004 by Michal Zalewski and was originally used to find dozens of bugs from browsers of the time.

Web browser fuzzers are quite common in the internet and with a quick search one can find tools to fuzz test a multitude of different features from browsers. This chapter presents an overview of techniques commonly used in those tools.

1.2.1. Test case generation

Both mutation-based [8] and generation-based [15, 17] test case generation approaches have been utilized in fuzz testers for web browsers. Because of the variety of file-formats and protocols supported by web browsers, most of the web browser fuzzers are designed for fuzz testing of only a single feature.

Typical browser fuzzers like jsfuzzer [18] and crossfuzz [19], are generation-based fuzzers targeted to the Document Object Model (DOM) [20]. DOM allows a user to dynamically alter the content of the current page, which in some cases leads to memory errors like use after frees and out of bounds accesses. DOM fuzzers are often designed to target only a small subset of DOM, for example HTML element addition and removal.

In contrast to HTML fuzzers like mangleme, fuzzers targeted to DOM must follow strict syntactic rules of JavaScript. Syntax errors in JavaScript code cause the interpreter to reject the test case before execution, limiting the testing to lexical and syntactic analysis of the JavaScript interpreter [16]. In DOM fuzzing, strict syntax also limits the effectiveness of mutation-based fuzzers that do not follow JavaScript syntax.

While jsfuzzer and mangleme generate test data on server side and then inject the data into the browser, fuzzers like crossfuzz and jsfunfuzz, do not generate actual test cases, instead they are written in JavaScript that is executed in the browser. In-browser fuzzing can be done, either by directly executing JavaScript commands one at a time, or by generating hundreds of lines of JavaScript and executing the whole script. When the test case generator is running outside of the browser, the test cases can be recorded and saved to a file when needed. A test case generator running inside the browser cannot report its state, if a fatal error occurs and the browser crashes. To solve this problem, both crossfuzz and jsfunfuzz, use seeding where usage of the same seed number consistently generates the same output.

Mutation-based fuzzing, in the context of browsers, is mainly used for binary file-formats, like images and videos, and formats with less strict syntactic rules than JavaScript. The mutation-based approach is also used to mutate files that previously caused error in some browser. This approach can reveal new bugs and situations where an applied patch did not actually fix the root cause of the bug. In the previous research on OUSPG, radamsa was used for fuzz testing of web browsers with 9 different file-formats [8].

1.2.2. Test case injection

In general web browsers offer two different interfaces for test case injection: CLI and network interface. CLI can be used to open Uniform Resource Locators (URLs) and local files. Testing with local files can be done by generating files on the local file system and executing browser to load the files via CLI. This approach was used in the previous research in OUSPG, where radamsa was used to generate files that were then loaded into to the browser with a shell-script. One weakness of this approach was that different test cases need different amount of time to be executed inside the browser, and no feedback is provided when the execution of a test case was finished. Another weakness is that through the command line, a new instance or a new tab is opened for each file loaded. This causes an overhead in resource usage for each file. Especially with simple test cases, like images, most of the processing power used is spent on the overhead. Resources are also used when a test case is written into a file. Writing test cases to files prior to testing can also be an advantage if the test cases can cause the whole system to freeze or crash, because the test cases can be retrieved after the system is restarted. In case of generating the test cases in memory, a sudden system freeze or hang would remove all data related to the executed test case.

Fuzzers like jsfuzzer and mangleme use browser's network interface and Hypertext Transfer Protocol (HTTP) to inject the test case into the browser. As an advantage to the previous methodology, this approach allows usage of browser's internal features to request a new test case and no new browser instance or tab is needed for each test case.

In mangleme, the HTTP-EQUIV META tag is used to set refresh for the page static time after the HTML page is received. This technique does not verify if the test case execution was finished before loading a new test case. Jsfuzzer uses the JavaScript window-objects onload-event to detect when the page is completely loaded before requesting a new test case.

ADBFuzz [12] is a browser fuzzer harness that uses WebSocket [21] protocol to deliver test cases into the browser. Before the WebSocket protocol, communication between a client and a server needed constant HTTP polling where every message needed a new connection to be created. Especially when using small payloads, the usage of HTTP for each message creates overhead compared to the actual payload delivered. WebSocket's allow bidirectional communication, which allows the fuzzer to monitor the browser's behavior more precisely during testing.

1.2.3. Instrumentation

Memory access bugs remain a serious problem in programming languages like C and C++, so most of the browser fuzzing efforts are dedicated to detect memory related errors, like use-after-free conditions or out-of-bounds-accesses. There are many different tools available for memory error detection on different platforms.

On Linux and MacOS, two tools are commonly used, Valgrind [22] and AddressSanitizer [23]. Valgrind [22] is a dynamic binary instrumentation framework. Valgrind has built-in tools, for detection of memory leaks and errors, that are used by both Mozilla and Google in their in-house testing. Google has also created their own Valgrind tool, ThreadSanitizer [24], that can be used to detect errors in threaded processes. With its default tools, Valgrind is resource intensive and causes slowdown of process execution time from 4 to 20 times of the original.

AddressSanitizer (ASAN) [23] is an efficient memory error detection tool, created at Google in 2011. ASAN is designed to detect out-of-bounds accesses and use-after-free bugs with only 73% slowdown of the executed process. In context of web browsers, ASAN has been used to detect over 300 previously unknown bugs from the Chromium browser. Both Google and Mozilla are using ASAN in their in-house testing. ASAN has made it possible for individuals, with limited hardware, to efficiently execute fuzz testing for Chrome and Firefox.

PageHeap [25] is a Microsoft Windows functionality that enables heap allocation monitoring for selected executables. By default PageHeap detects only buffer overruns. Optionally, the functionality can be expanded to detect buffer underruns and use after free conditions. Because of constant validation of the heap structures, usage of the PageHeap causes additional overhead, that may result in performance loss for the monitored process. Google uses PageHeap in their in-house testing.

Both Valgrind and ASAN have an internal error logging which, once an error has been detected, provides additional information about the error. This information can be used to identify the error, determine possible security impact and help in analysis of the cause of the error. PageHeap does not provide any additional information without an attached debugger.

Some browsers provide internal features that help in detection and analysis of different errors. For example, ADBFuzz uses Mozilla Firefox internal crash reporter to aid testing on Android OS.

1.2.4. Testing automation

Testing system automation for browsers should be able to automate testing on all the platforms a browser is available on. Also, the automation should be simple to configure for different target browsers, instrumentations, test case generators and test case injection methods.

Some browser fuzz testing harnesses like ADBFuzz are designed to test a single browser on a single platform with a single instrumentation. To switch any of the components a user would need to modify the core of the harness.

In the previous research in OUSPG, usage of radamsa was automated with a framework built from shell-scripts [8]. The framework could be adjusted for different instrumentations and target browsers. Shell-scripts allow direct control for the SUTs CLI and the underlying operating system, but limit the cross-platform compatibility of the framework and may impact the performance of the testing system, because no browser-specific features can be used to aid the automation.

Grinder [26] is an automated browser fuzzing framework consisting of fuzzing nodes and a centralized monitoring server. Grinder server provides a web interface for crash information handling from Grinder Nodes. Grinder nodes execute fuzz testing and inform the Grinder server when a crash occurs. Grinder can take use of different in-browser test case generators and can be targeted to different browsers, but Grinder nodes require a Windows operating system.

In 2012, Google published information about their proprietary web browser fuzzing infrastructure ClusterFuzz [27]. At that time, ClusterFuzz was executing over fifty million test cases per day on thousands of web browser instances. Because of the volume in which with the testing was done, the implementation of ClusterFuzz was designed to automate every step of the process. Features include test case generation and distribution, automatic crash analysis and duplicate detection with

ASAN, test case minimization, analysis of regression range and verification of the bug fixes. No information was published whether ClusterFuzz could be used with other instrumentation tools, but with ASAN instrumentation and virtual machines, ClusterFuzz would be limited to testing only on Linux.

2. IMPLEMENTATION

As described in Chapter 1, the testing framework implemented in previous research at OUSPG [8] had many features. The framework could use different instrumentations and could be used to test different target browsers. On the other hand, the cross-platform compatibility of the framework was limited, only CLI is used to control the browser and only radamsa is used as a test case generator. In the framework, the instrumentation and test case injection were a single module. When creating a new module, a user had to reuse code from existing modules or write new code for both parts.

NodeFuzz is a cross-platform browser fuzzing harness written in JavaScript and runs on a server-side JavaScript environment which is Node.js [28]. NodeFuzz is designed to overcome the limitations of the previous framework and to improve its efficiency in web browser fuzz testing.

The design of the previous framework was divided into parts, which are described in detail in Section 2.2. In NodeFuzz, users can implement modules used for test case generation and instrumentation, with only a few interface requirements. This enables users to create small test case generator modules targeted to a single feature of the browser, which was not possible with the previous framework. In terms of instrumentation, NodeFuzz enables a user to implement instrumentation tool specific modules, which is important for more effective testing and cross-platform compatibility. Instrumentation can make use of all the features of Node.js, which allows instrumentation of web browsers running on an external device.

In contrast to the previous framework, the modular design of NodeFuzz allows a user to create test case generator and instrumentation modules without any knowledge of the other modules or the core of NodeFuzz. For example, different individuals of a testing group could implement test case generator and instrumentation modules.

To utilize the modular design, NodeFuzz allows a user to specify which modules are loaded on start-up. A user can specify multiple test case generator modules and a single instrumentation module to be loaded, either via a configuration file or CLI. This feature is important, because it allows platform- and browser specific configurations.

NodeFuzz uses the network interface of the web browser to inject test cases into the browser, which allows testing on mobile devices that are not capable of running NodeFuzz on the device. The network interface also allows two-way communication between NodeFuzz and the target browser. In the current version of NodeFuzz the method of test case injection is predefined, because user defined test case injection modules may cause unwanted dependencies between different modules. Different test case injection methods might require different kind of parameters to be passed on to the browser, so the instrumentation module has to be aware of what kind of test case injection method is used.

This chapter describes the implementation of NodeFuzz. First, the key technologies used in NodeFuzz are presented. The second section presents an overview of the NodeFuzz architecture and relations among its components. The third and fourth section describe the NodeFuzz core and user defined external components. The last section describes a simple use case of NodeFuzz.

2.1. Technologies used

In NodeFuzz two key technologies are used: Node.js works as the software platform and the WebSocket-protocol as the communication channel between NodeFuzz and the target browser. This chapter presents an overview of these technologies and the reasons why they were chosen for the implementation of NodeFuzz.

2.1.1. Node.js

Node.js, also called Node, is a cross-platform server-side JavaScript environment built on Google Chrome's JavaScript runtime V8 [28]. Both V8 and Node are mostly implemented in C and C++, focusing on performance and low memory requirement [29]. There are multiple server-side JavaScript solutions, but Node was a logical choice because of the cross-platform capabilities and an active development community.

JavaScript was chosen because of the asynchronous and event-based nature of the language. Asynchronous behavior allows the system to generate test cases and analyze previously detected bugs while continuing with the actual testing. As a language JavaScript was a logical choice also because modern web browsers use JavaScript as their main scripting language, so a user using NodeFuzz can use the same language everywhere in the testing process.

2.1.2. WebSocket

The goal in the WebSocket Protocol is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections [21]. The protocol consists of an opening handshake that is followed by basic message framing over a TCP-connection. Because of the constant two-way communication and low data transfer overhead after the initial handshake, the WebSocket protocol is used in NodeFuzz to allow monitoring of the current state of the browser and to deliver test cases to the browser. WebSocket connections also allow injection of test cases into the target browser via a network connection, which makes it possible to test browsers that are not running on the same environment as NodeFuzz.

Node.js module *'node-websocket'* [30] is used in the NodeFuzz core for the WebSocket protocol implementation. Browsers supporting the WebSocket protocol have a JavaScript interface described in the W3C specification for the WebSocket API[31].

2.2. Architecture

NodeFuzz contains three separate components, each responsible for one part of the testing process. Core injects test cases to the browser, test case generator modules generate the data to be injected and instrumentation-module controls and instruments the browser under test. NodeFuzz also has a global configuration-object that contains predefined values, like ports to be used for servers and default locations for external files to be loaded by the NodeFuzz core. Figure 2 presents connections between different components in NodeFuzz.

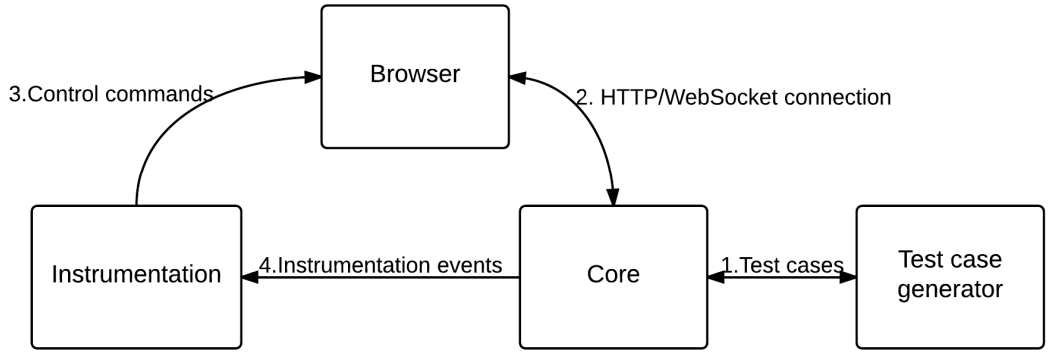

Figure 2 . NodeFuzz component connections.

2.2.1. Test cases

Test cases communication contains only a synchronous function-call from the core component to a single test case generator module. The test case generator module returns the generated test case.

2.2.2. HTTP/WebSocket connection

HTTP/WebSocket connection contains asynchronous two-way communication between the core and the browser. This communication is at first established as HTTP-connection, which is upgraded into a WebSocket-connection for test case delivery to the browser and for feedback-data delivery to NodeFuzz. The upgrade is handled from the browser and is done by a client-file that is sent to the browser as a response to the initial HTTP-request.

2.2.3. Control commands

Different browsers require different types of instrumentation, so this communication is user defined. In general this communication consists of browser start-, kill- and connect-commands from the instrumentation-module to the operating system or the browser and instrumentation-data like debugging-logs from the browser or operating system to the instrumentation-module.

2.2.4. Instrumentation events

This communication consists of events emitted from the NodeFuzz core. These events indicate that feedback from the browser was received, or something unexpected, like a disconnect or timeout, occurred in the WebSocket communication between NodeFuzz core and the browser. These events were implemented to offer the instrumentation a way to monitor the WebSocket connection if needed.

2.3. NodeFuzz Core

The NodeFuzz core is responsible for the initialization of the testing harness. The core initializes HTTP- and WebSocket-servers, loads all the user specified modules and verifies that the modules meet the requirements. The core also handles test case injection to the browser and emits events, when specific conditions are met while injecting test cases. This chapter describes all the components and phases of the NodeFuzz core.

2.3.1. Module loading

When initialized, the NodeFuzz core loads configuration-file, instrumentation module and test case generator modules. All the files must be valid JavaScript files that can be loaded via Node.js require. The configuration-file must be specified on CLI, or the NodeFuzz directory must contain a *'config.js'* file with all the required configurations. The instrumentation module file can be specified via the CLI, or by a property in configuration file. Test case generator modules can be loaded by specifying a single file or a folder either via CLI or configuration property. All the test case generator candidates are verified to contain the required method before the module is added to the list of loaded test case generators. Configuration-file, instrumentation module and at least one test case generator load must succeed, or the NodeFuzz core will exit. Requirements for external components are discussed in more detail in Section 2.4.

2.3.2. HTTP-server

The HTTP-server in NodeFuzz has only limited functionality. The server only responds to HTTP GET-requests, with the string defined in configuration object and only while no WebSocket connection is present. No further functionality was considered necessary, because after the initial handshake, all the communication between NodeFuzz and the browser is through a WebSocket connection. The HTTP-server uses the port defined in the configuration file.

2.3.3. WebSocket-server

The WebSocket-server in NodeFuzz offers two subprotocols that can be used when communicating with the browser. Connections using other subprotocols are rejected.

'fuzz-protocol' is used for test case delivery from the test case generator modules to the browser. Any message using this subprotocol will be answered with a new test case from a test case generator module, which is randomly selected from the list of loaded test case generator modules. To prevent any extra overhead in test case injection, no data should be sent from the browser to NodeFuzz with this subprotocol.

'feedbackloop-protocol' can be used to deliver feedback from the browser to NodeFuzz. This subprotocol was implemented to allow a user to deliver feedback from the browser to support instrumentation and test case generation algorithms.

2.3.4. Events

The NodeFuzz core implements a global event-emitter object for instrumentation purposes. In this thesis these events are called instrumentation-events. The core can emit five different kinds of events through the event-emitter.

testCasesWithoutRestartLimit

When a new WebSocket connection is created, a counter is initialized with value zero. The counter is incremented each time a new test case is sent through the WebSocket connection. The value of the counter is compared to the value of the configuration-object property *'testCasesWithoutRestart',* when a request for a new test case is received. If the value of the counter is higher than the defined value, the *'testCasesWithoutRestartLimit'*-event is emitted and the counter set back to zero. This event was implemented to provide information about the progress of the testing to the instrumentation-module.

websocketTimeout

The *'websocketTimeout'*-event is emitted when too much time has passed from the previous request for a new test case. The interval is defined in milliseconds by the configuration-object property *'timeout'*. When a new test case is requested, the previous timer is removed and a new timer is initialized. This event can be used by the instrumentation-module to detect when the browser has hung, or has become otherwise incapable to request a new test case.

websocketDisconnect

The *'websocketDisconnect'*-event is emitted when the WebSocket connection is disconnected. This event can be used by the instrumentation-module to detect that the connection between NodeFuzz and the browser has been lost.

feedbackMessage

The *'feedbackMessage'*-event is emitted when a message using *'feedback-protocol'* is received from the browser via a WebSocket-connection. The emitted event contains UTF-8 encoded message content. The content of the message is user defined and can be used to aid instrumentation- and test case generator-modules.

startClient

The *'startClient'*-event is emitted when all synchronous initializations are done and the NodeFuzz core is ready to accept connections. This event can be used by the instrumentation-module to detect when the browser can be started.

2.3.5. Test case injection

When the browser sends a message using *'fuzz-protocol'* through the WebSocket-connection, the NodeFuzz core responds with a new test case. The process of test case injection is shown in the Figure 3.

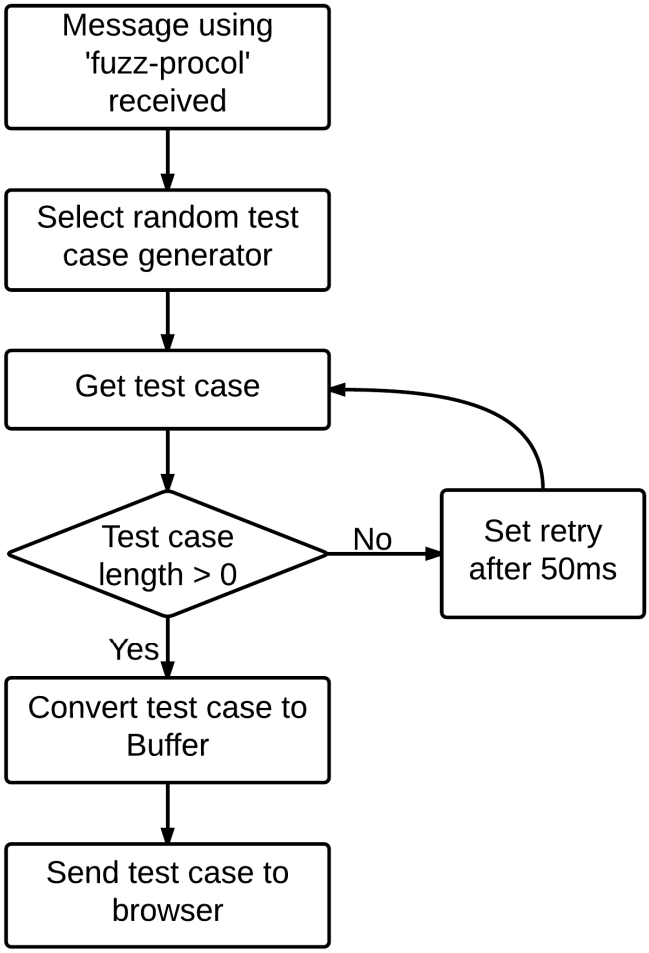

Figure 3. Test case injection process.

The core implements a test case buffer that holds test cases ready to be sent to the browser. The purpose of the test case buffer is to allow a new test case to be sent while the next test case is still being generated. When no test cases are available in the buffer, the core calls two randomly selected test case generators to generate test cases into the buffer, and a third test case generator to generate a test case to be sent to the browser immediately when ready. If the buffer has under four test cases available, two test case generators are called to generate more test cases into the buffer, while the oldest test case is taken from the buffer and sent to the browser. If the buffer has four or more test cases ready, then the oldest test case is taken from the buffer and sent to the browser.

After the test case is obtained, its size is checked. If the size is zero, the test case is not sent to the browser, but instead a retry is set to be executed after 50ms. This behavior is for compatibility with asynchronous test case generators that require more time to finish generation of a test case.

Before sending the test case to the browser, the test case is saved into a configuration-object property *'previousTestCasesBuffer'* and converted into the Buffer type.

2.4. External components

This section describes requirements of the external components, instrumentation-modules, test case generator-modules and configuration file.

2.4.1. Instrumentation

In NodeFuzz, instrumentation modules are responsible for control and instrumentation of the browser. Because of the different methods and tools available for the instrumentation, the instrumentation modules can be freely designed by the user and do not have any specific requirements. This allows implementation of different instrumentation modules that can be designed to reveal specific types of issues from the browser.

In general, the instrumentation module should start the browser, direct the browser to connect to the NodeFuzz core HTTP-server, start the instrumentation of the browser and restart the browser when needed. When instrumentation detects an error, the module should analyze the error and save the test case responsible of the error. The previous test cases can be obtained from a configuration-object property *'previousTestCasesBuffer'*.

Instrumentation-modules can take use of the instrumentation events provided by the NodeFuzz core.

2.4.2. Test case generator

Because browsers support a wide variety of different features, the idea in NodeFuzz test case generator modules is that the user has as much freedom as possible when implementing the module. This allows a user to use different approaches of fuzz testing without the possible restrictions of predefined generators or mutations.

The only requirement is that the module must export a *fuzz*-method that allows the NodeFuzz core to request a new test case when needed. The fuzz-method must always return a new test case synchronously and the returned test case must be of type, string or, Buffer. If the module is designed asynchronously and the new test case cannot be returned when requested the module should return an empty string. The module can also export an init-method, which is executed when the NodeFuzz core requires the module. This feature was implemented because in some cases the user wants to use their module independently from NodeFuzz, and the execution of NodeFuzz related initializations is not needed or might cause errors.

2.4.3. Configuration file

A configuration file is mandatory for NodeFuzz. By default NodeFuzz loads the file *'config.js'* from the NodeFuzz folder. The user can define other file to be loaded with the CLI argument *'-c <filename>'* or *'--config <filename>'*. The configuration file

must export an object that holds a set of required attributes. The attributes are used by the NodeFuzz core to define ports used in communication with the browser, default external files to be loaded if CLI parameters are not given, timeout for test case requesting and limit how many test cases are sent to the browser between restarts. To allow a user to define test case generator and instrumentation configurations via a single configuration file, the exported object is loaded into a global variable *'config'*. Table 1 lists all mandatory attributes of the configuration object.

Table 1. Mandatory configuration object attributes.

Attribute	Description
bufferSize	Defines previousTestCasesBuffer size
clientFile	HTTP-GET response HTML-string
defaultInstrumentationFile	Default instrumentation file path
defaultModuleDirectory	Default test case generator path
port	Port to be used by HTTP-server
testCasesWithoutRestart	Test cases without restart limit
timeout	Test case request timeout

The instrumentation file path can be overwritten with CLI arguments *'-i <filename>'* and *'--instrument <filename>'*. The test case generator path can be overwritten with arguments *'-m <filename>'* and *'--module <filename>'*.

2.5. Use case

Many of the modules for NodeFuzz are defined by the user. This chapter presents a use case of NodeFuzz, with implementations of configuration file, test case generator, instrumentation and client-file. These examples are only meant to clarify how the actual implementations could be done.

Configuration file

Figure 4 presents a use case configuration file. The configuration file exports object *'config'* that has all the NodeFuzz mandatory properties set and three additional properties for the instrumentation module. The additional properties define location where the results should be saved and what arguments are used when launching the browser.

```
var fs=require('fs')
var config = {}
//Mandatory configurations
config.bufferSize=10
config.clientFile=fs.readFileSync('./ClientFile.html').toString();

config.defaultModuleDirectory='./TestCaseGenerator.js'
config.defaultInstrumentationFile='./Instrumentation.js'

config.port=1000
config.testCasesWithoutRestart=100
config.timeout=15000

//Configurations for the instrumentation.
config.resultDir='./results/'
config.browserLaunchCommand="google-chrome"
config.browserArguments=[ '--incognito', 'http://127.0.0.1:'+config.port]

module.exports = config
```

Figure 4. Use case configuration file.

Test case generator

Figure 5 presents an example test case generator that generates a HTML-file which displays the text *'Hello Fuzz'* in the browser. The generator also generates ten lines of JavaScript code, that change the value of the style attribute *'zoom'* of the documents body. The value is changed to a random value between zero and three, with a random interval between zero and 30 milliseconds.

The test case generator exports the required method *'fuzz'* that synchronously returns a new test case when called. An optional *'init'* method is implemented to define a new value for the configuration object property timeout.

```
function generateTestCase(){
   var returnString='<html><body><p>Hello Fuzz</p><script>\n';
   for(var x=0;x<10;x++){
     returnString+='setTimeout(function(){';
     returnString+='document.body.style.zoom='+(Math.random()*3);
     returnString+='},'+Math.floor(Math.random()*30)+')\n';
   }
   return returnString+'</script></body></html>';
 }

module.exports={
 fuzz:generateTestCase,
 init:function(){
     config.timeout=100;
 }
}
```

Figure 5. Use case test case generator module.

Client file

Figure 6 shows an example client file, that has been tested on Mozilla Firefox and Google Chrome. The client-file of the example opens two WebSocket-connections to the NodeFuzz WebSocket-server, sends a message through *'feedback-protocol'* and starts loading test cases to an iframe-element, through a *'fuzz-protocol'*-connection. In the example, injected test data is defined to be type *'text/html'*, so test cases of

other file-formats might not be executed as intended. The example creates, a repetition of test case requesting, by initializing timeout to execute a new test case request 20ms after the *'onload'*-event of the iframe is triggered.

```
<html>
<script>
 var loc="ws://127.0.0.1:1000"
 var testCaseDelay

 window.addEventListener("load", function(event) {
  //Fuzz-protocol connection initialization.
  tcs = new WebSocket(loc, "fuzz-protocol");
  tcs.addEventListener("open", function(event) {
   tcs.send('');
  });
  tcs.addEventListener("message", function(event) {
   newTestCase(new Blob([event.data],{type:'text/html'}));
  });
  //Feedback-protocol connection initialization.
  fbs = new WebSocket(loc, "feedback-protocol");
  fbs.addEventListener("open", function(event) {
   fbs.send((new Date())+': Feedback-loop initialized.');
  });

  var target=document.createElement('iframe');
  document.body.appendChild(target)
  target.onload=function(e){
   clearTimeout(testCaseDelay);
   testCaseDelay=setTimeout(requestNewTestCase,20)
  }
 });

 function requestNewTestCase(){
  tcs.send('')
 }

 function newTestCase(data){
  target.src=window.URL.createObjectURL(data)
 }
</script>
</html>
```

Figure 6. Use case client-file.

Instrumentation

Figure 7 presents an example use case instrumentation module. The instrumentation module does not involve any external instrumentation tools, like ASAN, instead the NodeFuzz *'websocketTimeout'*-event is used for the detection of unexpected behavior of the browser. As a real world instrumentation, usage of only timeouts to determine when unexpected behavior has happened is inefficient and would result into a large amount of duplicates, but for the use case it is the easiest to demonstrate.

```
var spawn=require('child_process').spawn
var fs=require('fs')
var browser={}

function cloneArray(obj){
        var copy = [];
        for (var i = 0; i < obj.length; ++i) {
            copy[i] = obj[i];
        }
        return copy;
}

function saveResults(){
 var testCases=cloneArray(config.previousTestCasesBuffer)
 var time=new Date().getTime()
 if(!fs.existsSync(config.resultDir+'/'+time)){
  fs.mkdirSync(config.resultDir+'/'+time)
  testCases.forEach(function(testCase,index){
   fs.writeFileSync(config.resultDir+'/'+time+'/'+index+'.html',testCase)
  })
  restartBrowser()
 }
}

var startBrowser = function(){
 browser = spawn(config.browserLaunchCommand, config.browserArguments)
}

function restartBrowser(){
 browser.kill()
 setTimeout(startBrowser,1000)
}

function handleFeedback(data){
 console.log(data)
}

instrumentationEvents.on('websocketTimeout',saveResults)
instrumentationEvents.on('startClient',startBrowser)
instrumentationEvents.on('testCasesWithoutRestartLimit',restartBrowser)
instrumentationEvents.on('feedbackMessage',handleFeedback)
```

Figure 7. Use case instrumentation module.

The instrumentation starts when a *'startClient'*-event is detected. The binary defined by the configuration object is executed with the arguments defined by the configuration object.

When the *'websocketTimeout'*-event is detected, the current content of the *'previousTestCasesBuffer'*-array is saved to a location specified by the property *'resultDir'* from the configuration object. Results from each event are saved into a folder with a unique name defined by the timestamp of the event. After the results are saved, the current browser instance is killed and a new instance is set to be executed after one second.

If the *'testCasesWithoutRestartLimit'*-event is detected the current browser instance is killed and a new browser instance is set to be started after one second wait. If the *'feedbackMessage'*-event is detected, the received message is printed to the console.

3. EXPERIMENTS

During the development of NodeFuzz, ten test case generators and six instrumentation modules were implemented. These modules were implemented to prove the capabilities of NodeFuzz to take use of the test case generation and instrumentation techniques presented in Chapter 1.

This chapter outlines the implemented modules and the techniques utilized in them. Two of the test case generators, that revealed most of the found vulnerabilities, and two of the most used instrumentation modules are discussed in more detail in the following two sections.

To verify that the NodeFuzz test case generation can reach the same functionality as the framework implemented in previous research at OUSPG [8] *RadamsaModule* was implemented. *RadamsaModule* is described in more detail in Section 3.1.1.

To verify that NodeFuzz can take use of other test case generation techniques presented in Section 1.2.1, a group of other test case generation modules was created. *TableModule* uses the generation-based approach to DOM fuzzing similarly to jsfuzzer [18] and crossfuzz [19], but is targeted only to HTML table-elements. *HTMLModule* generates HTML text documents in a similar manner than mangleme [17]. *WebSocketModule* is the only in-browser test case generator implemented. It is targeted to the browsers WebSocket API by generating a bundle of WebSocket related method calls based on a seed-number that is sent to NodeFuzz prior to execution via WebSocket '*feedbackloop-protocol*'.

To prove that the NodeFuzz test case generation capabilities are not limited only to the techniques presented in Section 1.2.1 a second group of test case generator modules was created. *RadamsaDOMModule* extends the *RadamsaModule* HTML-fuzzing with DOM fuzzing. The module first mutates a HTML-sample with radamsa, then searches the file for HTML-element attributes like *id* and *class*, and uses those as base for DOM fuzzing. The module generates JavaScript that executes DOM manipulations inside the browser and inserts the script into the HTML-file before passing the file to the NodeFuzz core for injection. Table 2 lists all the implemented test case generator modules, the targeted feature and the used fuzz testing approach.

Table 2. Implemented test case generator modules.

Module	Tested feature	Generation-based	Mutation-based
HTMLModule	HTML-parsing	X	-
SurkuModule	File-formats	-	X
SurkuHTMLModule	HTML-parsing	X	X
RadamsaModule	File-formats	-	X
RadamsaDOMModule	HTML+DOM	X	X
TableModule	HTML tables	X	-
CanvasModule	HTML5 Canvas	X	-
SVGModule	SVG	X	-
WebAudioModule	Web Audio API	X	-
Base64ImgModule	Image files	X	X
WebSocketModule[1]	WebSocket API	X	-

[1)]Test case generation is done in the target browser and only the generator seed is returned to instrumentation via WebSocket connection.

For the NodeFuzz to work as intended in terms of instrumentation, five requirements must be met: cross-platform capability, capability to execute testing against different target browsers, capability to use OS internal features, capability to use instrumentation tools and capability to test a browser running on a separate device.

To verify that NodeFuzz instrumentation modules can function on different platforms, two modules were implemented: *LinuxDmesg* and *WinEventLog*. At the same time that these modules verify the cross-platform capabilities, they also verify that the instrumentation can be used against different targets and can take use of the OS internal features. *LinuxDmesg* utilizes the command *dmesg*, available in most Linux- and UNIX-based OSs. The command writes the content of the kernel ring buffer to the standard output. The instrumentation module takes use of the kernel ring buffer data about processes terminated with an uncaught fatal signal. *WinEventLog* takes use of the Windows built-in Event log that holds various system messages including application errors. Both modules can be used to instrument browsers that do not handle exceptions internally. For example, Mozilla Firefox handles all exceptions internally, so these modules cannot be used to instrument it.

To verify that NodeFuzz can use instrumentation tools available on different OSs three additional instrumentation modules were implemented: *ASAN, PageHeap* and *SyzyASan*. *ASAN* and *PageHeap* modules are described in more detail in Section 3.2. *SyzyASan* module takes use of SyzyASan [32], a Windows specific tool similar to ASAN.

The last instrumentation module was implemented to verify that NodeFuzz can be used also to instrument a browser running on another device. *LinuxAndroid* module runs on Linux and uses the Android Debug Bridge to control and monitor Google Chrome on Android devices.

Table 3 lists all the implemented instrumentation modules, the supported operating systems and browsers the module was tested with.

Table 3. Implemented instrumentation modules.

Module	**Operating system**	**Browser**
LinuxAndroid	Linux[1]	Chrome on Android
ASAN	Linux, OSX	Chrome, Firefox
LinuxDmesg	Linux	Chrome, Iceweasel
WinEventLog	Windows 7	Chrome, Internet Explorer
PageHeap	Windows 7	Chrome
SyzyASan	Windows 7	Chrome

[1)]NodeFuzz is executed on Linux, but the target browser is executed on Android OS.

3.1. Test case generator experiments

This section presents two of the implemented test case generator experiments: the radamsa and Web Audio modules.

The first module utilizes the general purpose fuzzer radamsa and its TCP-server feature to provide the test cases. Radamsa was selected for this experiment because it was used in the previous browser fuzzing research at OUSPG and because it can be used to fuzz any types of data. The purpose of this module was to prove that

NodeFuzz test case injection can be used to inject test cases of different file-formats into the browser and to prove that NodeFuzz test case generation can take use of existing external tools.

The second module was designed to test the Web Audio API[33] implementation of the target browser. The Web Audio API was chosen for this experiment because of the simplicity of the specification. The purpose of this module was to prove that the test case generators can be written in JavaScript running on the Node.js platform.

Both modules were tested with instrumentation modules described in Section 3.2.

3.1.1. Radamsa module

Radamsa is a general purpose fuzzing tool that uses the mutation-based approach. Previously radamsa was used in OUSPG to discover around 60 bugs from Google Chrome and Mozilla Firefox.

In practice, radamsa is a bundle of different mutation algorithms in a single tool. Some of the algorithms randomly alter input data, some look for commonly used patterns and mutate input accordingly.

As an input radamsa can read files from a location specified via CLI, or the data can be passed through *stdin*. Radamsa can output the fuzzed input via *stdout*, TCP connection, or write the output into a file. For this experiment radamsa was set to read sample-files from a folder located in a RAM-disk and to output results through TCP connection.

The sample-files used with radamsa were collected from Google Chrome and Mozilla Firefox source code repositories. Both repositories contain groups of test cases, for conformance and regression testing, for multiple file-formats.

Because the test cases are actually generated by radamsa, the implemented module only needs to handle sample-files and passing of test cases from radamsa to the NodeFuzz core.

The module is separated into two asynchronous parts. The first one handles sample-files, client-file and radamsa. The second handles a buffer of test cases to be forwarded to the NodeFuzz core when requested.

Radamsa control

When the module is initialized, the CLI arguments given to NodeFuzz are checked for the argument *'filetype=<filetype>'*. If the parameter is present, the value is compared with a list of available file-formats. If the value is in the list, a configuration object for that file-format is loaded. If the given value is not in the list of available file-formats, or the value is not given, the module defaults to use HTML file-format.

The configuration object contains custom configuration for the specified file-format. In addition to the mandatory NodeFuzz configurations, the configuration object includes the location of the sample-files, amount of sample-files to be used and how many test cases radamsa should generate before samples are switched.

After the configuration is loaded, the module creates a RAM-disk and copies sample-files into the RAM-disk. Sample-files are selected randomly from the files available in the sample-folder.

When sample-files are ready, the module spawns an instance of radamsa with the values provided by the configuration object. The TCP-port to be used by radamsa is the port number, defined in NodeFuzz configuration, incremented by 1.

To detect when radamsa has generated the amount of files specified by the configuration, an event listener is attached to the spawned radamsa *'exit'*-event. When the event is emitted, a new round is started by copying new sample-files to the RAM-disk.

The radamsa control flow is presented in Figure 6.

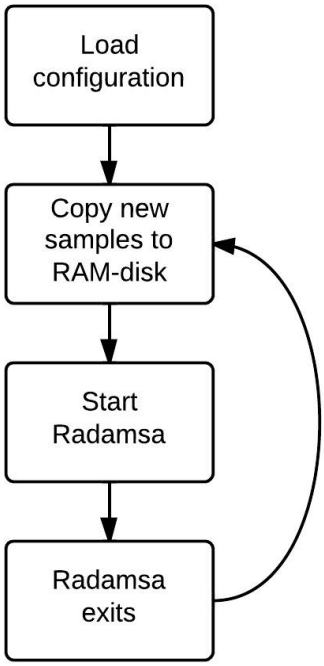

Figure 8. Radamsa control flow.

Test case control

When the NodeFuzz core requests a test case from the module, a new test case is taken from a test case buffer and passed on to the core. The test case buffer was implemented to eliminate the delays caused by the TCP connection and radamsa restarts.

When a test case is taken from the buffer, a recursive function is called to fill the buffer. The filler function tries to connect to the TCP-port defined for radamsa. If a connection is established and a test case received, the test case is added to the buffer as a raw Buffer-object. The process is repeated as long as there is less than 6 test cases to the buffer. If the connection cannot be established, the recursion is stopped and no further action is taken before the next test case is requested from the core.

The buffer can run out of test cases if the browser consumes test cases faster than radamsa can generate new ones, or if a restart of radamsa takes too long. When a request for a new test case is made by the core and no test cases are in the buffer, an empty string is returned.

Results

The module was tested with 19 different file-formats. The file extensions of tested file-formats are listed in Table 4. In addition to the listed file extensions, different versions of the file-formats were tested and in case of HTML different APIs like WebGL were also included in the tests.

Table 4. Tested file-format extensions.

.apng	.bmp	.gif	.html
.ico	.jpg	.mht	.mml
.mp3	.mp4	.ogg	.ogv
.pdf	.png	.swf	.svg
.wav	.webm	.webp	

NodeFuzz was able to inject all file-formats supported by the tested browsers, except the web page archive format MIME HTML (MHTML) (file extension .mht). For some reason none of the mime-types specified for the MHTML worked for the file construction in the browser and the MHTML files were shown only as plain-text.

Some of the file-formats, for example mp3, are not supported without third-party decoders or plugins.

The total amount of vulnerabilities found with the radamsa module is unclear due to loss of log records tracking the reported vulnerabilities. Vulnerabilities found from stable versions of the browsers can be tracked with Common Vulnerabilities and Exposures(CVE) identifiers. The CVE identifiers of vulnerabilities found with the module are listed in Table 5.¨

Table 5. Vulnerabilities found with the radamsa module.

Google Chrome	Mozilla Firefox
CVE-2012-5108	CVE-2012-4186
CVE-2012-2887	CVE-2012-4187
CVE-2012-5120	CVE-2012-4188
CVE-2012-5121	CVE-2012-4202
CVE-2012-5145	CVE-2013-0744

3.1.2. Web Audio module

The Web Audio API specification developed by W3C describes a high-level JavaScript API for processing and synthesizing audio in web applications. The goal of the specification is to include the capabilities found in modern game audio engines as well as some of the mixing, processing, and filtering tasks that are found in modern desktop audio production applications. The primary paradigm in the API is an audio routing graph, where a number of AudioNode objects are connected to define the overall audio rendering.[33]

Web Audio API specification

In the Web Audio API, different types of AudioNodes are created from an AudioContext interface. When a new AudioNode is created, the node is expanded with an interface according to what type of node is created.

Different types of interfaces have different sets of properties that are described in the specification. Some properties are defined as an AudioParam interface, that include functionality to control the property value with precise timing.

AudioNode and AudioParam interfaces are used in the specification to uniformly present different types of AudioNodes. Some of the AudioNodes implement unique properties.

JavaScript interface

Usage of the Web Audio API in browsers is straightforward. Create an AudioContext object, from the context create a number of AudioNodes, connect the nodes and adjust the properties of the AudioNodes. A sample program is shown in Figure 9. The sample program creates an AudioContext, OscillatorNode and GainNode. The OscillatorNode is connected to the GainNode which is connected to the AudioContext destination. The GainNode has its gain value set to 0.1 and the OscillatorNode is set to generate a sine-wave. The OscillatorNode starts generating the sine-wave at time 0, which means immediately when the code line is executed. The generated sine-wave is passed from the OscillatorNode to the GainNode and from the GainNode to the playback device, AudioContext destination.

```
var Context = new window.AudioContext();
var Oscillator=Context.createOscillator();
var Gain=Context.createGain();

Oscillator.connect(Gain);
Gain.connect(Context.destination);

Gain.gain.value=0.1;

Oscillator.type="sine"
Oscillator.start(0);
```

Figure 9. Web Audio API JavaScript interface example.

AudioNode interface modeling

Before any test cases can be generated, different AudioNode interfaces had to be modeled into a structure. In general, an interface can be described with the name of the interface, a set of attributes and a set of methods.

As an example of how the interfaces were modeled, the model for the DynamicsCompressorNode is presented in Figure 10 as a JavaScript. The *'AudioParamMethods'*-function is a helper function that creates common methods of AudioParams for each name in the input array.

```
var AudioNodes=[
    ['DynamicsCompressor',[DynamicsCompressorAttributes,
                           DynamicsCompressorMethods]
    ],
    ...
]

var DynamicsCompressorAttributes=[
    ['attack.value',function(){return Math.random()}],
    ['knee.value',function(){return Math.random()*rint(45)}],
    ...
]

var DynamicsCompressorMethods=AudioParamMethods(['knee',
                                                 'attack',
                                                 ...])
function rampToValue(){
    return randomNumber()+','+getTimeValue()
}

function AudioParamMethods(names){
    var returnArray=new Array()
    var methods=[
        ['exponentialRampToValueAtTime',[rampToValue]],
        ['cancelScheduledValues',[function(){
                                      return getTimeValue()
                                  }]],
        ...
    ]
    names.forEach(function(name){
        methods.forEach(function(method){
            returnArray.push([name+'.'+method[0],method[1]])
        })
    })
    return returnArray
}
```

Figure 10. DynamicsCompressorNode generation model.

Test case generation

The test case generation creates five blocks of code: static HTML, AudioContext creation, AudioNode creation, connections and properties. The size of the blocks, as lines to be generated, is defined with a configuration object. Figure 11 presents example output of the Web Audio API test case generator.

```
<html>
<script>
//AudioContext creation block
try{var Context0 = new window.webkitAudioContext();}catch(e){}
try{var Context1 = new window.webkitAudioContext();}catch(e){}
//AudioNode creation block
try{var DynamicsCompressor0=Context0.createDynamicsCompressor();}catch(e){}
try{var DynamicsCompressor1=Context0.createDynamicsCompressor();}catch(e){}
//AudioNode connections and properties block
try{DynamicsCompressor0.knee.cancelScheduledValues("0.42704");}catch(e){}
try{DynamicsCompressor1.connect(Context0.destination);}catch(e){}
try{DynamicsCompressor1.attack.value="0.69304624688816";}catch(e){}
</script>
</html>
```

Figure 11. Web Audio API test case generator output example.

Static HTML is generated around the JavaScript to enable the execution of the script inside the browser. The HTML code includes only starting and ending tags from html- and script-elements.

The AudioContext and AudioNode creation blocks contain JavaScript lines that, in the browser, create new AudioContexts and different types of AudioNodes. The type of the AudioNode to be generated is randomly selected from the AudioNodes-array presented in Figure 10.

The generator counts how many different types of objects are created and uses the count when generating a variable name to store the object. The count is also used when generating connections and properties within the generator.

The connections and properties blocks contain JavaScript lines that, in the browser, set attributes and call methods of the generated objects. The lines are generated by walking down the generation model.

The model is walked down so that the process starts from the AudioNodes-array. One of the cells presenting an AudioNode type that has been generated in the AudioNode creation block is selected first, then either the attributes or the methods array is selected before the actual attribute or method to be generated is selected.

Results

With the Web Audio module, a total of eleven vulnerabilities was found from Google Chrome stable releases. From Mozilla Firefox, a total of nine vulnerabilities were found. In Firefox the Web Audio implementation was only available on the development version of the browser. The CVE identifiers for Chrome vulnerabilities and Mozilla Bugzilla bug numbers for Firefox vulnerabilities are listed in Table 6.

Table 6. Vulnerabilities found with the Web Audio module.

Google Chrome	Mozilla Firefox
CVE-2013-2845	876252
CVE-2013-0904	877125
CVE-2013-0916	878478
CVE-2013-0879	884459
CVE-2013-2906	874952
	876118
	880202
	874869
	874915

3.2. Instrumentation experiments

To demonstrate the capabilities of the NodeFuzz instrumentation, this section describes two NodeFuzz instrumentation modules.

The first instrumentation uses ASAN. ASAN was chosen for the instrumentation experiment because of its relatively low slowdown on the executed process and because Mozilla and Google use ASAN on their in-house testing. Both vendors provide instructions on how to use ASAN together with their products and offer pre-built binaries for the current source code trunk.

The second instrumentation uses the Microsoft Windows PageHeap. The PageHeap was selected for the second instrumentation experiment because Google uses it in their in-house testing and instructions for its usage can be found from the Chromium developer network[34]. No instructions for Mozilla Firefox were found, so only Google Chrome was supported in this instrumentation implementation. The PageHeap instrumentation demonstrates that NodeFuzz can be used also on other platforms than Linux and MacOS.

3.2.1. AddressSanitizer instrumentation

When the ASAN-instrumented binary is executed and an error is detected, the process execution ends and an error-report is printed to the process *stderr*. The raw report does not contain any symbols in the stack trace, so it doesn't provide much information. Even without the symbols, the trace can be used for detection of errors, like null-pointer dereferences and intentional crashes to fixed address, that are unlikely to cause any security threat. The report can be symbolized by a python-script available in the ASAN source code repository. The process of symbolization can take lots of RAM and processing power, so it should not be done if not necessary.

Error detection

The ASAN instrumentation monitors the browsers *stderr* for ASAN-output. During the error detection state, test cases are sent to the browser through a WebSocket connection. Instrumentation will exit from the error detection state only if ASAN-output is detected, or if a WebSocket timeout- or disconnect-event is received.

Error analysis

Before executing symbolization, the raw output is analyzed to verify that the output contains ASAN-output and that the output is not a null-pointer dereference. This is done by checking that the error report contains the string "ERROR: AddressSanitizer" and that the error reported by ASAN is not of type SEGV with a crash address of null.

Interesting outputs are symbolized and parsed for the error type, the page-aligned instruction pointer and the symbols of the first frame in the stack. Figure 12 presents an example snippet from symbolized ASAN-output of a vulnerability in the Google Chrome Web Audio API implementation. The example output contains general information about the error detected, such as error type, registry values, access type, access size and three stack frames. In the example, the information used by the ASAN instrumentation is shown in red.

```
==23393== ERROR: AddressSanitizer: heap-use-after-free on address
0x601600035d48 at pc 0x7f130433ad46 bp 0x7fffd8e6f090 sp 0x7fffd8e6f088
WRITE of size 32 at 0x601600035d48 thread T0 (chrome)
    #0 0x7f130433ad45 in WebCore::Biquad::reset()
third_party/WebKit/Source/WebCore/platform/audio/Biquad.cpp:203
    #1 0x7f130433a64c in WebCore::AudioDSPKernelProcessor::reset()
third_party/WebKit/Source/WebCore/platform/audio/AudioDSPKernelProcessor.cpp
:105
    #2 0x7f12ffe8f216 in WebCore::BiquadFilterNode::setType(unsigned int)
third_party/WebKit/Source/WebCore/Modules/webaudio/BiquadFilterNode.cpp:95
```

Figure 12 . ASAN stack trace parsing.

Saving the results

The data parsed in error analysis is used to form a crash identifier. From the trace presented in Figure 12, the resulting identifier would be *'heap-use-after-free-WebCoreBiquadreset-d46'*. The identifier is used as a folder name for the results, so duplicate detection can be done by checking if a folder with the same name as the identifier exists. If not, then the folder is created and the symbolized output and files from '*previousTestCasesBuffer*' are saved to the folder.

Workflow

Figure 13 presents ASAN instrumentation workflow. After loading, the ASAN instrumentation waits for *'startClient'*-event. When the event is emitted, the instrumentation spawns a browser instance with arguments from the configuration-object. After the spawn, the instrumentation attaches an event listener to the spawned browser's *stderr* and starts to wait for an ASAN-output. When an ASAN-output is detected, the instrumentation analyses the output. If the analysis indicates that the issue is interesting, the output is symbolized and the results are saved if necessary.

Platforms

AddressSanitizer is supported on Linux i386/x86_64 (officially tested on Ubuntu 10.04 and 12.04) and on MacOS 10.6, 10.7 and 10.8 (i386/x86_64) [35].

Results

During the development of NodeFuzz, the ASAN instrumentation has been the main instrumentation module. The module was tested to be functional on Ubuntu Linux for both Google Chrome and Mozilla Firefox, and on OSX for Google Chrome.

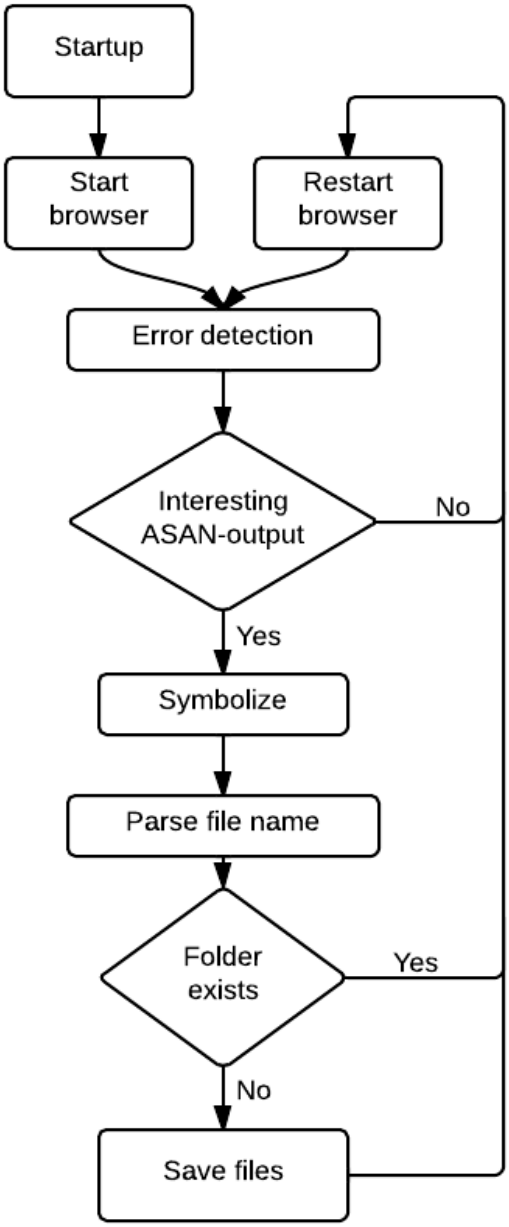

Figure 13. ASAN instrumentation workflow.

3.2.2. PageHeap instrumentation

When PageHeap detects an error from an instrumented process, the process execution is terminated. Without the attached debugger, PageHeap doesn't provide any information about the crash, but in case of Google Chrome, the user can enable minidump-generation when a crash occurs. Minidump can be analyzed with windbg or cdb, which both are provided by the Debugging Tools for Windows [36] package.

Platforms

PageHeap is provided for Microsoft Windows 2000 and newer. This instrumentation experiment was tested on Windows 7 Ultimate 64-bit only.

Enable PageHeap

Before starting the target browser, PageHeap must be enabled for the browser executable. Once enabled, PageHeap will instrument any instance of the executable launched until the instrumentation is disabled. PageHeap can be enabled for the target application with the gflags-tool that is provided by Debugging Tools for Windows package. PageHeap changes the memory-layout of the executed process and in some programs it may cause malfunctions.

For Google Chrome, the x86 version of gflags must be used. Google Chrome uses tcmalloc to allocate pages, so by default many of PageHeap's benefits won't work. The allocator can be changed to a PageHeap compatible one by setting the environment variable CHROME_ALLOCATOR to "winheap" [34].

The PageHeap instrumentation sets the environment variable and enables PageHeap for the Google Chrome executable before the first startup of the browser.

Error Detection

Without an attached debugger, PageHeap doesn't provide any information about the error when it occurs, so the PageHeap instrumentation cannot detect when an error occurs just by monitoring the application outputs.

In this implementation, the error detection waits for a WebSocket timeout- or disconnect-event, then retrieves the time of the newest Google Chrome related error event from the Windows Event Log. If the retrieved time is different from the previous time, an error has occurred and the analyzer is initialized. Otherwise the browser is restarted and the instrumentation set back to wait for the next event.

By default, Google Chrome errors are not shown in the Windows Event Log, but user can enable logging by setting "Automatically send usage statistics and crash reports to Google" in the Google Chrome settings.

Error analysis

When "Automatically send usage statistics and crash reports to Google" is enabled in the Google Chrome settings, also the minidump-generation is enabled.

Minidump is analyzed and symbolized with windbg and the output is parsed for the information about the error type and symbols. In comparison to the ASAN instrumentation, all errors must be analyzed and no performance gain can be achieved by filtering of the non-security related errors. Figure 14 shows, in red, the parsed information from windbg analysis.

```
0:000> !analyze -v
*******************************************************************************
*                                                                             *
*                        Exception Analysis                                   *
*                                                                             *
*******************************************************************************

FAULTING_IP:
chrome_6bd20000!WebCore::AudioDSPKernelProcessor::reset+1b
[c:\...\src\third_party\webkit\...\audiodspkernelprocessor.cpp @ 105]
6cb1d33c 8b11            mov     edx,dword ptr [ecx]

EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
.exr 0xffffffffffffffff
ExceptionAddress: 6cb1d33c (...!WebCore::AudioDSPKernelProcessor::reset+0x0000001b)
ExceptionCode:    c0000005 (Access violation)
ExceptionFlags:   00000000
NumberParameters: 2
Parameter[0]:     00000000
Parameter[1]:     3033bfa0
Attempt to read from address 3033bfa0

PROCESS_NAME:  chrome.exe
```

Figure 14. WinDbg analysis parsing.

Saving the results

Like in the ASAN instrumentation, the parsed data from WinDbg analysis is used to form a file name for the folder and files to be saved. From the example analysis in Figure 14, the resulting file name would be *'AccessViolation-WebCoreAudioDSPKernelProcessorreset01b'*. The analysis output and reproducing files from the *'previousTestCasesBuffer'* are saved into the result folder. Detection of duplicate issues is done by checking if a folder with the same name already exists before saving results.

Workflow

Figure 15 presents the PageHeap instrumentation workflow. After loading, the PageHeap instrumentation waits for *'startClient'*-event. When the event is emitted, the instrumentation enables PageHeap for the target browser and spawns a browser instance with arguments from configuration-object. After the browser is spawned, the instrumentation waits for an instrumentation-event. When an instrumentation-event is emitted, the instrumentation checks the Windows Event Log for new events. If a new event is found, the instrumentation starts to analyze the minidump-file and saves the analysis results if necessary.

Results

The PageHeap instrumentation module was used to verify Windows compatibility of NodeFuzz and the implemented test case generators. Instrumentation was tested on Windows 7 x64 with Google Chrome. During the testing of the instrumentation, four browser vulnerabilities that were not reproducible on Ubuntu Linux were revealed.

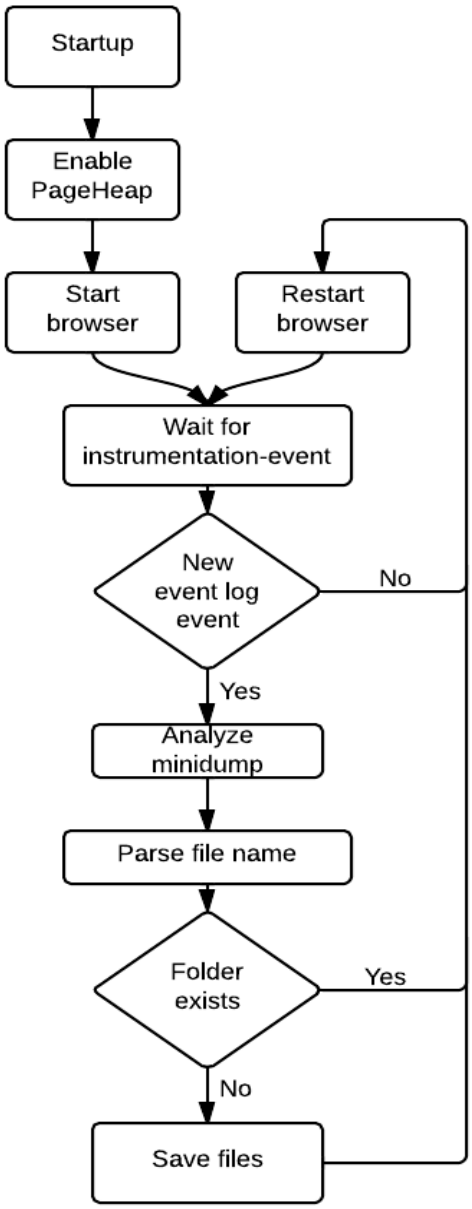

Figure 15. PageHeap instrumentation workflow.

4. DISCUSSION

Typical browser fuzzers, as presented in Section 1.2, are small test case generators that are targeted to a single feature of a browser. The experiments presented in this thesis prove that NodeFuzz can take use of the different test case generation techniques used in state of the art browser fuzzers. In contrast to browser fuzzers like mangleme and jsfuzzer, NodeFuzz provides automation for the whole process of fuzz testing, which renders further comparison impractical.

The framework previously created in OUSPG, as well as ADBFuzz and Grinder, automate the complete process of fuzz testing, but are not designed to be as modular as NodeFuzz in terms of test case generation and instrumentation. As a result, the user cannot as easily create and maintain a versatile browser testing system. Table 7 presents features of state of the art browser fuzzing frameworks and harnesses. There is no detailed information available about the capabilities of Google ClusterFuzz, so the comparison may not include all features.

Table 7. Browser fuzz test framework and harness features.

Name	Test case generation	Instrumentation	Testing platform[1]	Test case injection
OUSPG's framework	radamsa	multiple	Linux	CLI
ADBFuzz	multiple	single	Android	WebSocket
Grinder	multiple	single	Windows	HTTP
ClusterFuzz	multiple	single	Linux	-
NodeFuzz	multiple	multiple	All above	WebSocket

[1)]Testing platform means the platform where the browser under test is executed.

Allowing different test case injection methods would have caused dependencies and complexity for modules, so NodeFuzz was limited to use a single method of test case injection. NodeFuzz uses the browser's network interface and the WebSocket protocol for test case injection to create a two-way communication between NodeFuzz and the browser. From the test case injection methods presented in Section 1.2.2, the WebSocket protocol generates the lowest overhead in terms of process time and network traffic. The experiments show that this approach allows NodeFuzz to inject test cases of different file formats into the browser, and enables NodeFuzz to use the browser's internal features to monitor the test case execution and to deliver test cases to browser instances running on external devices. NodeFuzz uses the same test case injection method as ADBFuzz and, in terms of efficiency and control, WebSocket is better than CLI and HTTP, used by the OUSPG's previous framework and Grinder.

In addition to testing capabilities, NodeFuzz also provides an easy to use interface for the user defined modules. A user can implement new test case generation modules without any information about the instrumentation modules that are going to be used with it, and vice versa. OUSPG's previous framework also allows usage of multiple instrumentation techniques, but the instrumentation and test case generation are a single module, so for each new instrumentation technique all old modules have to be redesigned.

From June 2012 to October 2013, NodeFuzz has been running continuously on a cluster of 10 machines with a varying set of test case generator modules. During that time over 100 bugs were reported. From those, more than 50% were previously unknown bugs which were by the vendor considered to have a relevant security impact. From the reported issues, over 35 were on the newest stable release version at that time. Figure 16 presents mentions of the author of this thesis from Chromium Security Hall of fame.

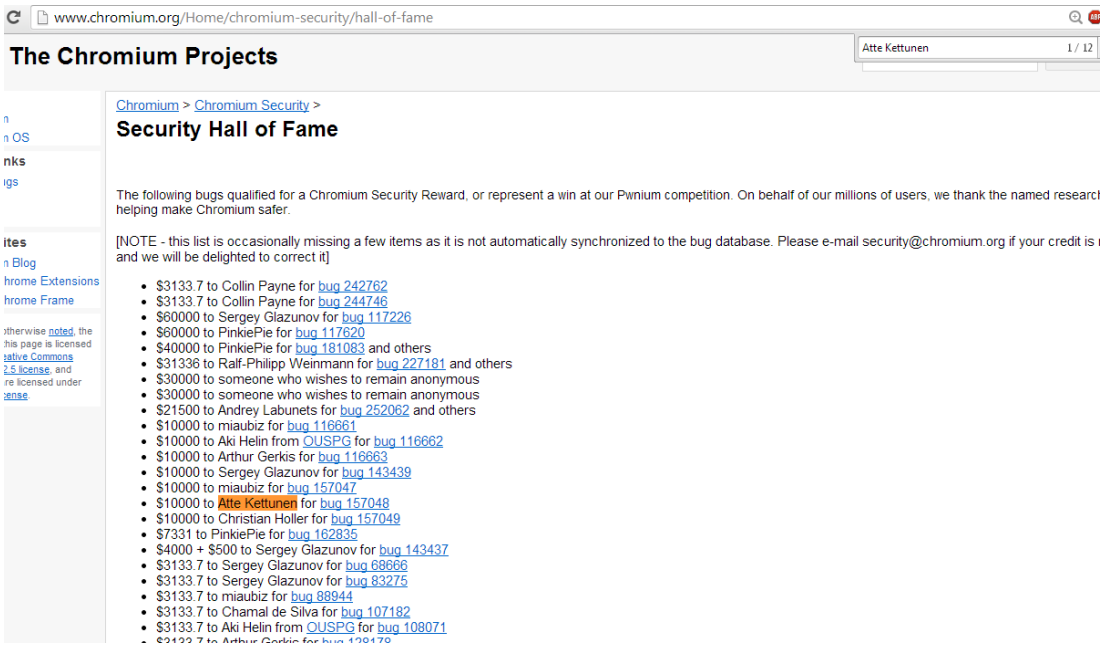

Figure 16. Chromium Security Hall of Fame mentioning the author of this thesis.

The damage that could have been caused by the vulnerabilities found cannot be estimated, but most of the vulnerabilities were found from commonly used third-party projects and libraries, for example WebKit [37] used by Google Chrome.

WebKit is also used in the two most popular mobile OS, Android and iOS and according to the WebKit wiki-pages [38] in at least 100 different applications. During 2013, Google has stated [39] that Android OS is powering over 1 billion smartphones and tablets. In June 2013, Apple chief executive officer Tim Cook announced [40] 600 million iOS devices, including iPhones, iPads and iPod devices, sold. Considering only the two mobile OSs, some of the issues found during this research affect the security of over 1.6 billion devices.

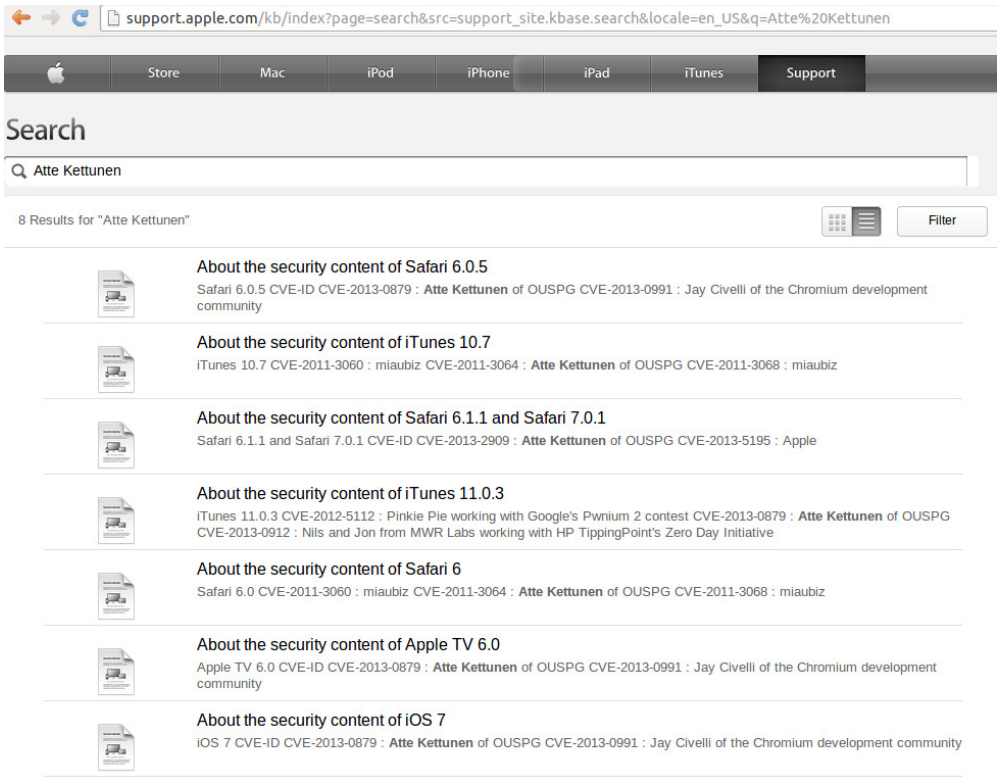

Figure 17. Apple release notes mentioning the author of this thesis.

NodeFuzz was proven to be an efficient approach to browser fuzzing, but there are still areas to improve. The main area to improve is the test case injection. Currently, NodeFuzz has the capability to use only the WebSocket connection for the test case injection. To get more extensive coverage of interfaces that can be used to inject test cases into the browsers, also the test case injection method should be definable in a similar way as the test case generator and instrumentation modules. The WebSocket connection is efficient when testing features that only require file loading, but some APIs, like WebRTC [41], cannot be tested by using only WebSocket.

5. SUMMARY

The goal of this research was to overcome the limitations of the previous browser fuzzing framework implemented by OUSPG, improve its efficiency in web browser fuzz testing and to allow usage of different techniques in test case generation and instrumentation. As a result, a cross-platform browser fuzzing harness called NodeFuzz was implemented.

NodeFuzz runs on the server-side JavaScript platform Node.js and uses the WebSocket-protocol to communicate with the browser under test. In NodeFuzz, test case generation and instrumentation is executed by user defined modules. The used test case generators and instrumentation can be selected either via a configuration file or CLI, which allows a user to flexibly switch between different configurations and platforms, without any modifications to the NodeFuzz core.

Originally, the idea was to use HTTP in communication between NodeFuzz and the browser under test. A HTTP-server implementation would have been simpler to implement, but in the end the constant two-way communication and low data transfer overhead of the WebSocket-protocol were considered more valuable. With the WebSocket-protocol, in addition to test case delivery, the implemented modules can monitor the state of the browser under test without the overhead that would be caused by the HTTP-protocol.

NodeFuzz achieved the goals of this research. It was proven to work cross-platform, to use test case generators implementing different approaches of fuzz testing and to utilize different types of instrumentation tools. Even though the amount of found bugs depends on the implemented test case generators and instrumentation modules, the total of over 100 reported bugs prove that NodeFuzz is powerful harness for browser fuzz testing.

With every new test case generator new bugs have been found, yet still the test case generators implemented during this research cover only a fraction of the features available in modern browsers. This suggests that there is still more research needed in the field of browser security testing.

6. REFERENCES

[1] A. Grosskurth and M. W. Godfrey, "A reference architecture for Web browsers," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 661–664.

[2] P. Wood, "Symantec, Internet security threat report, 2012 Trends Volume 18," 2013.

[3] "Security Bug Bounty Program Announcement." [Online]. Available: http://www.mozilla.org/en-US/press/mozilla-2004-08-02.html. [Accessed: 13-Jun-2013].

[4] "Encouraging More Chromium Security Research, blog, 28 Jan. 2010." [Online]. Available: http://blog.chromium.org/2010/01/encouraging-more-chromium-security.html. [Accessed: 13-Jan-2014].

[5] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Addison-Wesley, 2007.

[6] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding Software Vulnerabilities by Smart Fuzzing," *2011 Fourth IEEE Int. Conf. Softw. Testing, Verif. Valid.*, pp. 427–430, Mar. 2011.

[7] A. Takanen, J. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. 2008.

[8] P. Pietikäinen, A. Helin, R. Puuperä, A. Kettunen, J. Luomala, and J. Röning, "Security Testing of Web Browsers," *Communications of the Cloud Software*, 2011. [Online]. Available: http://www.cloudsw.org/issues/2011/1/1. [Accessed: 17-Oct-2013].

[9] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[10] "What is Peach?" [Online]. Available: http://peachfuzzer.com/WhatIsPeach.html. [Accessed: 13-Jun-2013].

[11] M. Vuagnoux, "Autodafe: An act of software torture." tech. report, Swiss Federal Institute of Technology (EPFL), Cryptograhy and Security Laboratory (LASEC), 2006.

[12] C. Holler, "ADBFuzz – A Fuzz Testing Harness for Firefox Mobile, blog, 9 Mar. 2012." [Online]. Available: https://blog.mozilla.org/security/2012/03/09/adbfuzz-a-fuzz-testing-harness-for-firefox-mobile/. [Accessed: 27-Jan-2014].

[13] C. Miller and Z. N. J. Peterson, "Analysis of Mutation and Generation-Based Fuzzing." tech.report, Independent Security Evaluators, 2007.

[14] P. Oehlert, "Violating Assumptions with Fuzzing," *IEEE Secur. Priv. Mag.*, vol. 3, no. 2, pp. 58–62, Mar. 2005.

[15] “Jesse Ruderman, Introducing jsfunfuzz.” [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. [Accessed: 14-Jun-2013].

[16] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” *Proc. USENIX Secur.*, 2012.

[17] “mangleme – Freecode.” [Online]. Available: http://freecode.com/projects/mangleme. [Accessed: 13-Jun-2013].

[18] “jsfuzzer - A browser based javascript fuzzer - Google Project Hosting.” [Online]. Available: https://code.google.com/p/jsfuzzer/. [Accessed: 29-Aug-2013].

[19] M. Zalewski, “Announcing cross_fuzz, a potential 0-day in circulation, and more, blog, 01 Jan. 2011.” [Online]. Available: http://lcamtuf.blogspot.fi/2011/01/announcing-crossfuzz-potential-0-day-in.html. [Accessed: 29-Aug-2013].

[20] “W3C Document Object Model.” [Online]. Available: http://www.w3.org/DOM/. [Accessed: 29-Aug-2013].

[21] I. Fette and A. Melnikov, “RFC 6455 - The WebSocket Protocol.” 2011.

[22] N. Nethercote and J. Seward, “Valgrind,” *ACM SIGPLAN Not.*, vol. 42, no. 6, p. 89, Jun. 2007.

[23] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: a fast address sanity checker,” in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012, p. 28.

[24] K. Serebryany and T. Iskhodzhanov, “ThreadSanitizer,” in *Proceedings of the Workshop on Binary Instrumentation and Applications - WBIA ’09*, 2009, p. 62.

[25] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows® Internals, Part 2: Covering Windows Server® 2008 R2 and Windows 7*. 2012.

[26] Stephen Fewer, “Grinder.” [Online]. Available: https://github.com/stephenfewer/grinder. [Accessed: 18-May-2014].

[27] A. Arya and C. Neckar, “Fuzzing for Security, blog, 26 Apr. 2012.” [Online]. Available: http://blog.chromium.org/2012/04/fuzzing-for-security.html. [Accessed: 04-Sep-2013].

[28] “Node.js.” [Online]. Available: http://nodejs.org/. [Accessed: 06-Jun-2013].

[29] S. Tilkov and S. V. Verivue, “Node . js : Using JavaScript to Build High-Performance Network Programs,” 2010.

[30] “Worlize/WebSocket-Node · GitHub.” [Online]. Available: https://github.com/Worlize/WebSocket-Node. [Accessed: 23-Sep-2013].

[31] “The WebSocket API.” [Online]. Available: http://www.w3.org/TR/websockets/. [Accessed: 04-Sep-2013].

[32] “SyzyASanDesignDocument - sawbuck - An address sanitizer based on Syzygy. - A log controller and viewer for Chrome Windows developers - Google Project Hosting.” [Online]. Available: https://code.google.com/p/sawbuck/wiki/SyzyASanDesignDocument. [Accessed: 02-Sep-2013].

[33] C. Rogers, “Web Audio API.” [Online]. Available: http://www.w3.org/TR/webaudio/. [Accessed: 26-Sep-2013].

[34] “Page Heap for Chromium - The Chromium Projects.” [Online]. Available: http://www.chromium.org/developers/testing/page-heap-for-chrome. [Accessed: 29-Aug-2013].

[35] “AddressSanitizer — Clang 3.4 documentation.” [Online]. Available: http://clang.llvm.org/docs/AddressSanitizer.html. [Accessed: 07-Jun-2013].

[36] “Download and Install Debugging Tools for Windows.” [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/gg463009.aspx. [Accessed: 29-Aug-2013].

[37] “The WebKit Open Source Project.” [Online]. Available: http://www.webkit.org/. [Accessed: 16-Oct-2013].

[38] “Applications using WebKit – WebKit.” [Online]. Available: http://trac.webkit.org/wiki/Applications using WebKit. [Accessed: 16-Oct-2013].

[39] “Android KitKat.” [Online]. Available: http://www.android.com/kitkat/. [Accessed: 16-Oct-2013].

[40] “WWDC ’13: Apple keynote, by the numbers | ZDNet.” [Online]. Available: http://www.zdnet.com/wwdc-13-apple-keynote-by-the-numbers-7000016583/. [Accessed: 16-Oct-2013].

[41] “WebRTC.” [Online]. Available: http://www.webrtc.org/. [Accessed: 16-Oct-2013].