Felix Albrigtsen
Kristoffer Longva Eriksen
Kristoffer Juelsen

Uncovering Software Vulnerabilities Using Source Code Analysis and Fuzzing

Bachelor's thesis in Digital Infrastructure and Cyber Security
Supervisor: Donn Morrison
May 2024

Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of Science and Technology

Felix Albrigtsen
Kristoffer Longva Eriksen
Kristoffer Juelsen

Uncovering Software Vulnerabilities Using Source Code Analysis and Fuzzing

Bachelor’s thesis in Digital Infrastructure and Cyber Security
Supervisor: Donn Morrison
May 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Uncovering Software Vulnerabilities Using Source Code Analysis and Fuzzing

Felix Albrigtsen Kristoffer Longva Eriksen Kristoffer Juelsen

May 21, 2024

Preface

This thesis marks the end of our three years of studying Digital Infrastructure and Cyber Security under the Department of Computer Science at the Norwegian University of Science and Technology.

We would like to thank our supervisor, Donn Morrison, for his guidance during the course of our research. We would also like to thank Kine Albrigtsen, Sarah K. Johansen, Heine R. Løge, and Lea L. Raknes for providing valuable feedback when finalizing this thesis.

Due to the technical nature of this thesis, the reader will get the most out of reading this with some existing theoretical knowledge of computer science.

Abstract

This thesis aims to explore the capabilities of source code analysis and fuzzing for uncovering vulnerabilities in programs written in a systems programming language. We will be conducting a penetration test on NetSurf, an independent web browser written in the C programming language.

With a rapid increase in the number of cybersecurity threats, and more applications moving from the desktop into the web browser, the security of the browser is more important than ever. The browser is an essential tool for anyone who uses the web for productivity, communication, entertainment, finance, or a plethora of other tasks, emphasizing the need to keep it secure.

We will utilize tools and techniques for source code analysis and fuzzing to search for vulnerabilities in NetSurf. Throughout our research, we will compare the capabilities of these two methods, and evaluate their performance and usability in different situations. Before going into the practical penetration test, we will first explain central concepts surrounding penetration testing, source code analysis, fuzzing, and software vulnerabilities in general.

The findings and experiences from our testing will be used to evaluate and compare the different tools and techniques in question. We will use these results in combination with previous research as a background to find answers to our research questions and problem statement. In the end, we will conclude that source code analysis and fuzzing can complement each other, and the combination of both techniques can successfully uncover a wide range of bugs and vulnerabilities in a project like NetSurf.

Sammendrag

Målet med denne oppgaven er å utforske hvordan statisk kildekodeanalyse og fuzzing kan brukes til å avdekke sårbarheter i programmer skrevet i et lavnivåspråk. I denne oppgaven vil vi gjennomføre en penetrasjonstest av Netsurf, en selvstendig nettleser skrevet i programmeringsspråket C.

Ettersom antallet trusler innen cybersikkerhet øker, og stadig flere skrivebordsapplikasjoner flyttes inn i nettleseren, er behovet for å sikre nettleseren større enn noen gang. Nettleseren er et essensielt verktøy for alle som bruker den til produktivitetsarbeid, kommunikasjon, underholdning, banktjenester og en rekke andre formål, som tydeliggjør behovet for å sikre den.

Vi skal bruke verktøy og teknikker innen statisk kildekodeanalyse og fuzzing for å lete etter svakheter i NetSurf. Gjennom denne forskningen skal vi vurdere disse to metodene ved å sammenligne hvor godt de fungerer i forskjellige situasjoner. Før vi setter igang med den praktiske penetrasjonstestingen skal vi først forklare sentrale teoretiske konsepter og begreper rundt penetrasjonstesting, statisk kildekodeanalyse, fuzzing og sårbarheter i programvare.

Vi vil bruke våre funn og erfaringer fra denne testingen til å sammenligne de forskjellige verktøyene og metodene vi har brukt. Disse resultatene kan brukes sammen med tidligere forskning som bakgrunn når vi svarer på forskningsspørsmålene og problemstillingen vår. Til slutt vil vi konkludere med at disse to metodene kan utfylle hverandre godt, og at kombinasjonen av de to kan være effektiv for å avdekke en rekke forskjellige programvarefeil og sårbarheter i et prosjekt som NetSurf.

Contents

Preface . . . iii
Abstract . . . v
Sammendrag . . . vii
Contents . . . ix
Figures . . . xiii
Tables . . . xv
Code Listings . . . xvii
Acronyms . . . xix
Glossary . . . xxi
CWE List . . . xxiv
1 Introduction . . . 1
1.1 Background . . . 1
1.2 Purpose . . . 2
1.3 Target Selection . . . 3
1.4 Problem Statement . . . 3
1.5 Research Questions . . . 4
1.6 Scope . . . 4
1.7 Thesis Outline . . . 5
2 Theory . . . 7
2.1 Penetration Testing . . . 7
2.1.1 Common Methodologies . . . 7
2.1.2 Penetration Testing Use Cases . . . 8
2.2 Source Code Analysis . . . 8
2.2.1 Selection of SAST Tools . . . 9
2.2.2 Advantages of SAST . . . 9
2.2.3 Disadvantages of SAST . . . 10
2.3 Fuzzing . . . 10
2.3.1 Advantages of Fuzzing . . . 12
2.3.2 Disadvantages of Fuzzing . . . 13
2.4 Vulnerability Overview . . . 13
2.4.1 Common Vulnerability and Exposures (CVE) . . . 13
2.4.2 Common Weakness Enumeration (CWE) . . . 14
2.4.3 Types of Vulnerabilities . . . 15
2.4.4 Analysis of Some Relevant CWEs . . . 16

2.4.5 Program Control . . . 23
2.4.6 Other Vulnerabilities and Exploits . . . 24
3 Method . . . 25
3.1 NetSurf and the Build Process . . . 25
3.2 Working with SAST . . . 26
3.2.1 Tools . . . 26
3.2.2 Required Setup . . . 28
3.2.3 Managing the Results . . . 29
3.3 Working with Fuzzing . . . 30
3.3.1 Tools . . . 30
3.3.2 Required Setup . . . 30
3.3.3 Managing the Results . . . 38
3.4 Responsible Disclosure . . . 40
4 Results . . . 41
4.1 Source Code Analysis . . . 41
4.1.1 Snyk . . . 41
4.1.2 Semgrep . . . 43
4.1.3 Coverity . . . 44
4.1.4 Useful Findings . . . 46
4.1.5 Useless Findings . . . 52
4.2 Fuzzing . . . 55
4.2.1 Domato . . . 55
4.2.2 AFL++ . . . 57
4.2.3 Findings . . . 58
4.3 Disclosure of Discovered Vulnerabilities . . . 62
5 Discussion . . . 63
5.1 Sources of Error . . . 63
5.1.1 NetSurf Characteristics . . . 63
5.1.2 Manual Analysis . . . 64
5.1.3 Usage of SAST Tools . . . 64
5.1.4 Usage of Fuzzing Tools . . . 65
5.2 Exploring the Problem Statement . . . 66
5.2.1 Performance and Accuracy of SAST Tools . . . 66
5.2.2 Performance and Accuracy of Fuzzing . . . 67
5.2.3 General Security of the NetSurf Project . . . 68
5.3 Quality of Research . . . 68
5.4 Future Work . . . 68
6 Conclusion . . . 69
Bibliography . . . 71
A Additional Material . . . 81
A.1 Initial Findings - Snyk . . . 81
A.2 Initial Findings - Semgrep . . . 85
A.3 Initial Findings - Coverity . . . 90
A.4 Ignore Files . . . 96

A.5 Nix Flake 98
A.6 Docker Configuration 100

Figures

2.1 Magic Quadrant for Application Security Testing [66] 9
2.2 The Binary Fuzzing Pipeline 11
2.3 CVE Record Lifecycle - From the CVE Website [78] 14
2.4 Example Product Lifecycle - From the CWE Website [80] 15
2.5 Illustration of Stack Frames [83] 19
2.6 Freed Chunks in Tcache Bins 22
2.7 Linked List of Free Heap Chunks after Reusing One of Them 23
2.8 Output Demonstrating a Use After Free vulnerability 23
2.9 Output from a Double Free 23

3.1 Screenshot of fuzz-00000.html in netsurf-gtk3 33
3.2 The master section of docker-compose.yml 36
3.3 AFL++ Status Output 37
3.4 Program Stopped with Abort Signal 39

4.1 Pwndbg Output after a Stack Buffer Overflow 48
4.2 Heap Contents after Copying String to Heap Chunk 49
4.3 Pwndbg Output before Call to textarea_replace_text 50
4.4 Assembly Code for _chain_bloom_generate Displayed in Pwndbg . 51

A.1 Potential Vulnerabilities per Component - Snyk 81
A.2 Potential Medium Vulnerabilities - Snyk 82
A.3 Potential Low Vulnerabilities - Snyk 82
A.4 Potential Vulnerabilities per Component - Semgrep 85
A.5 Potential High Vulnerabilities - Semgrep 85
A.6 Potential Medium Vulnerabilities - Semgrep 86
A.7 Potential Low Vulnerabilities - Semgrep 86
A.8 Potential Vulnerabilities per Component - Coverity 90
A.9 Potential High Vulnerabilities - Coverity 90
A.10 Potential Medium Vulnerabilities - Coverity 91
A.11 Potential Low Vulnerabilities - Coverity 91

Tables

1 CWE Overview xxiv

4.1 Overview of Possible Security Vulnerabilities - Snyk 42
4.2 Severity of Possible Security Vulnerabilities - Snyk 42
4.3 Useful SAST Findings - Snyk 43
4.4 Overview of Possible Security Vulnerabilities - Semgrep 43
4.5 Severity of Possible Security Vulnerabilities - Semgrep 43
4.6 Useful SAST Findings - Semgrep 44
4.7 Overview of Possible Security Vulnerabilities - Coverity 45
4.8 Severity of Possible Security Vulnerabilities - Coverity 45
4.9 Useful SAST Findings - Coverity 46
4.10 Breakdown of the Crashes Discovered by AFL++ 58

Code Listings

2.1 C Code Demonstrating an Integer Overflow 17
2.2 Output Demonstrating an Integer Overflow 18
2.3 C Code Demonstrating a Stack Buffer Overflow 20
2.4 Output Demonstrating a Stack Buffer Overflow 20
2.5 C Code Demonstration of a Use After Free Vulnerability 21
2.6 Python Code Exploiting a Use After Free Vulnerability 22

3.1 Packages Section of flake.nix 31
3.2 Some Samples from a File Generated by Domato 33
3.3 Bash Script to Process the Files Generated by Domato 34
3.4 AFL-based CC and stdenv Defined in netsurf-nix/default.nix 35
3.5 GDB Backtrace After Opening fuzz-00002.html in NetSurf 38
3.6 fuzz-00002.html Minified 39

4.1 C Code from idna_encode 47
4.2 HTML Code Exploiting idna_encode 48
4.3 C Code from gui_get_clipboard 49
4.4 C Code in _chain_bloom_generate 51
4.5 C Code in fire_dom_keyboard_event 52
4.6 C Code in duk__handle_break_or_continue 53
4.7 C Code in DUK_TVAL_SET_U32 53
4.8 C Code in hlcache_handle_release 53
4.9 C Code for RING_ITERATE_STOP and END 54
4.10 C Code in plot_alpha_bitmap 54
4.11 Result Statistics from Testing with Domato 55
4.12 Minimal Script that Causes dom_crash/fuzz-00003.html 56
4.13 Minimal Script that Causes dom_crash/fuzz-00016.html 56
4.14 Result Statistics From Testing with Domato, Without JavaScript . . 56
4.15 Example Output from afl-showmap 57
4.16 C Code in box_normalise_table 58
4.17 C Code in calculate_table_row 59
4.18 HTML Code Exploiting box_normalise_table 59
4.19 C Code in table_calculate_column_types 59
4.20 HTML Code Exploiting table_calculate_column_types 60

4.21 C Code in current_node 60
4.22 HTML Code Exploiting current_node 60
4.23 C Code in textarea_reflow_multiline 61
4.24 HTML Code Exploiting current_node 62

A.1 .snyk File 96
A.2 .semgrepignore File 96
A.3 Nix Flake 98
A.4 Dockerfile to Run AFL++ on NetSurf 100
A.5 Docker Compose Configuration for Running Multiple Instances of afl-fuzz 100
A.6 Basic Dockerfile for Debugging an Unmodified NetSurf 101

Acronyms

AFL++ American Fuzzy Lop plus plus. xxi, 5, 11–13, 30, 31, 35–38, 40, 55, 57–60, 65, 67

ASLR Address Space Layout Randomization. 24

CI/CD Continuous Integration and Continuous Deployment. 8, 28, 64, 66, 69

CLI Command Line Interface. 28

CNA CVE Numbering Authority. 14

CVE Common Vulnerability and Exposures. xix, 13, 14, 40, 62

CVSS Common Vulnerability Scoring System. 14

CWE Common Weakness Enumeration. 14, 15, 24, 29, 30, 41

DOM Document Object Model. 32, 35, 64

GCC GNU Compiler Collection. xxi, 35

GDB GNU Debugger. xxii, 38, 39, 55

GOT Global Offset Table. 22

JOP Jump-Oriented Programming. 24

LOC lines of code. xxi, 4, 41

NVD National Vulnerability Database. 14

NX No-eXecute. 24, 49

PoC Proof of Concept. 5, 7, 14, 30, 40, 62

ROP Return-Oriented Programming. 24

SAST static application security testing. 3, 5, 8–12, 15, 26, 27, 29, 30, 38, 41, 42, 44–46, 52, 53, 58, 60, 63–69

XSS Cross-Site Scripting. 47, 56

Glossary

black box a term describing a situation where a (simulated) attacker does not have any internal knowledge of the target system. 7, 8

breakpoint a configurable point in a program where the debugger will stop execution [1, p. 33]. 49

clang an LLVM-based c compiler, as an alternative to GCC. 35

compiler a computer program that reads source code and generates an executable program. 11, 26, 35, 47, 50–52, 66

core dump a file containing a process' memory after it terminates unexpectedly [2]. 38

defect density the amount of vulnerabilities found per X lines of code. 27

Docker virtualization software designed to help developers build, share, and run container applications [3]. xxi, 30, 36, 37, 62, 67

Docker Compose a tool for orchestrating applications with multiple Docker containers. 30, 36

fuzzer an application that controls the fuzzing process, such as AFL++. xxii, 10–13, 30, 32, 35, 37, 38, 65, 67, 68

fuzzing a process that "delivers a large amount of machine-generated inputs as quickly as possible to the target in order to find some objectives"[4]. v, vii, xxi, 2–5, 8, 10–13, 30, 35–38, 55, 57, 58, 60, 63–69

GNU Make a tool which controls the generation of executables and other non-source files of a program from the program's source files [5]. The Makefile serves as a "recipe" for compiling and building an application. 25, 26, 28, 31, 35

GTK a free and open source "widget toolkit", useful for writing graphical user interfaces. 25, 33, 61

heap chunk parts of heap memory that has been allocated by `malloc` or similar functions [6, p. 2]. xxii, 20, 49, 50

input corpus the pool of files used by a fuzzer to retrieve input files [7]. 11–13, 37, 38, 57, 58, 68

instruction pointer A pointer to the next instruction to execute [8]. 18–20, 24, 48, 49

libc the GNU C Library, which implements all library functions which are specified by the ISO C standard, and others [9]. 15–17, 20–23, 58, 59

LLVM a collection of modular and reusable compiler and toolchain technologies [10]. xxi, 35

Nix a purely functional, cross-platform package manager and its corresponding ecosystem, including the Nix language, nixpkgs repositories and the NixOS Linux distribution. 26, 28, 30–32, 35, 36, 67

pwndbg a GDB plug-in that improves debugging with GDB [11]. 48, 49

pwntools an exploit development library written in Python, designed for rapid prototyping and development [12]. 21

stack canary an indicative value added to binaries to ensure that critical stack values, like the return pointer, are protected from buffer overflow attacks [13]. 49

stack pointer A pointer to the most recently pushed value on the stack [14]. 18

systemd an software suite for Linux, providing service management, logging, process control and other system critical features. 38

tcache bin data structure used for keeping track of heap chunks that have been freed. 20, 22

vulnerability a "flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components" [15]. v, 1–5, 7–10, 12–18, 23, 24, 27–30, 38, 40–45, 47, 49–54, 56, 58, 60–69

weakness "A condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities" [15]. 14–16, 45, 54

white box a term describing a situation where a (simulated) attacker has complete access to internal details of the target system. 7

CWE List

CWE ID	Title
CWE-14	Compiler Removal of Code to Clear Buffers [16]
CWE-20	Improper Input Validation [17]
CWE-22	Improper Limitation of Pathname to a Restricted Directory ("Path Traversal") [18]
CWE-23	Relative Path Traversal [19]
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer [20]
CWE-120	Buffer Copy without Checking Size of Input ("Classic Buffer Overflow") [21]
CWE-121	Stack-based Buffer Overflow [22]
CWE-122	Heap-based Buffer Overflow [23]
CWE-125	Out-of-Bounds Read [24]
CWE-131	Incorrect Calculation of Buffer Size [25]
CWE-170	Improper Null Termination [26]
CWE-190	Integer Overflow or Wraparound [27]
CWE-197	Numeric Truncation Error [28]
CWE-252	Unchecked Return Value [29]
CWE-328	Use of Weak Hash [30]
CWE-367	Time-of-Check Time-of-Use (TOCTOU) Race Condition [31]
CWE-369	Divide By Zero [32]
CWE-404	Improper Resource Shutdown or Release [33]
CWE-415	Double Free [34]
CWE-416	Use After Free [35]
CWE-457	Use of Uninitialized Variable [36]
CWE-467	Use of sizeof() on a Pointer Type [37]
CWE-476	NULL Pointer Dereference [38]
CWE-484	Omitted Break Statement in Switch [39]
CWE-561	Dead Code [40]
CWE-563	Assignment to Variable without Use [41]
CWE-606	Unchecked Input for Loop Condition [42]
CWE-676	Use of Potentially Dangerous Function [43]
CWE-775	Missing Release of File Descriptor or Handle after Effective Lifetime [44]
CWE-787	Out-of-Bounds Write [45]
CWE-916	Use of Password Hash With Insufficient Computational Effort [46]

Table 1: CWE Overview

Chapter 1

Introduction

1.1 Background

Software has become an important part of modern society, and just about every industry, company, and individual uses a wide array of different computer programs every day. All of these pieces of software are built upon a stack of different technologies, techniques, and programming languages, and they are generally built to address a specific problem or use case.

Looking at the Stack Overflow Developer Survey, an annual survey where around 90.000 software developers from around the world self-report what technologies they are using at work and for fun, we can see several trends and patterns in what technologies are in use. One of the largest and most undeniable patterns, is that the World Wide Web is one of, if not the most important area of software development today. We can see that over 65% of developers use JavaScript for work, and over 50% use HTML and CSS in the most recent survey from 2023 [47].

The web browser is one of the core parts of many modern applications, and is gradually taking over many jobs that native applications used to have. Today, it is common to edit documents, play video games, and chat with friends directly in the browser, even allowing entire operating systems like ChromeOS to be built around the web browser.

On the other end of software development is systems programming, which can be quite different than web development. In systems programming, you use lower level languages like C, Zig, and Rust, and work closer to the hardware. Despite being developed over 50 years ago, `C` is one of the most widely used systems programming languages, used by around 16% of professional developers according to the 2023 Stack Overflow survey [47]. It is a popular choice for many applications in a wide range of industries, from low-power embedded devices to high-performance servers for business-critical tasks. When writing C, the programmer is responsible for managing the programs memory. This results in great performance and freedom, but also responsibility. When writing code with many complex memory operations, vulnerabilities can be easy to introduce and difficult to detect.

The number of cybersecurity threats have grown over the last few years and are expected to keep growing at a fast pace. Threat actors use software vulnerabilities to exploit companies and users for personal gain, in the increasingly lucrative business of cyber crime [48]. Ensuring that applications and services are secure is therefore more vital than ever, as the threat landscape adapts and evolves with emerging technologies, such as Artificial Intelligence (AI) and Large Language Models (LLMs). To protect against the threat of cyberattacks, we need to know where the potential vulnerabilities are. This thesis will investigate methods for finding vulnerabilities, specifically looking at a browser written in C. We will utilize common methods created for this purpose, specifically source code analysis and *fuzzing*.

The increasing number of cybersecurity threats has lead to an increased amount of research and discussion about the topic, both in and outside technical communities. Research papers regarding cybersecurity have increased by orders of magnitude since 2011 [49]. Regarding the specific field of searching for vulnerabilities in software, there have been several research papers in the last few years investigating integrating static analysis into the process of fuzzing. Some of these papers include *Static program analysis as a fuzzing aid* [50], *Shfuzz: A hybrid fuzzing method assisted by static analysis for binary programs* [51], and *Hybrid testing: Combining static analysis and directed fuzzing* [52]. However, we have not been able to find much prior research specifically comparing source code analysis to fuzzing. Other researchers have used similar techniques when attacking web services and APIs using URL fuzzing [53], a technique that has little in common with our approach to binary fuzzing. Another aspect we have not seen covered in other research is the possible advantage of using both methods together as a part of the development process, rather than for penetration testing. This thesis will therefore be exploring and comparing these tools when looking for vulnerabilities in the context of an open source application.

1.2 Purpose

The purpose of this thesis is to explore tools and techniques that can be used to discover vulnerabilities in software written in systems programming languages like C. We will describe, use, and evaluate some of these techniques and find out what tools are the most useful in different circumstances. We will look at source code analysis and fuzzing, and aim to give a better understanding of how they work, how to use them, and how they can be effective at finding different types of software vulnerabilities.

1.3 Target Selection

When comparing and analyzing different tools and techniques, the results are highly dependent on the project we are targeting. Various programming languages, project structures, coding styles, and types of software can greatly affect what tools we can use, their performance, and our own performance. We decided to put down a set of requirements and considerations that could help us select a target.

We wanted to find a project that is written in a suitable programming language, not too small or too large, and able to impact a large number of users. Regarding programming languages, we wanted to look at a project where different code analysis utilities could be the most useful. For static analysis, a strongly and statically typed language is more suitable than a dynamically typed language, as every constant and variable has a known type and size, even before compilation [54]. This rules out everything written in languages like Python or JavaScript, as types are implied and might change at runtime, making static analysis more difficult. With a large group of modern languages removed, we are left with more strongly typed languages like Java, Go, Rust, C++, and C. Out of these languages, we kept all possibilities open, but with a preference for technologies that we are familiar working with, mostly C. From experience, we also know that software written in C are more prone to memory management problems than Rust or Java, which might make it easier to find vulnerabilities [55].

The other two requirements are more subjective in nature, but equally important to choose a good target. In our search, we considered a range of programs in different styles and categories, from Integrated Development Environment (IDE)s to web browsers and video streaming utilities. The size of the project is important to ensure that we have enough varied code to get a representative view of how our tools handle different sections and styles of code, without having to repeat the same type of work many times. When looking for vulnerabilities and vulnerability analysis techniques, we also want to know how our potential findings can impact users and businesses.

Based on the above criteria, we have decided that *NetSurf*, an independent web browser written in C, is the perfect target to demonstrate and evaluate our set of tools and techniques.

1.4 Problem Statement

This thesis describes our processes and results from investigating how security researchers can discover software vulnerabilities using tools like fuzzing and source code analysis. To accomplish this, we will perform a penetration test, more specifically a code audit, of some components in NetSurf. We will use static application security testing (SAST) tools, fuzzing, and a combination of these methods. After testing, we will evaluate our findings, and compare how the different tools and techniques fare against different scenarios and vulnerabilities. This should give a

broader understanding on how a penetration tester can use these tools in combination for a more effective code audit. With this in mind, we will try to find out if *source code analysis, fuzzing, or a combination of the two can be effective at finding real vulnerabilities in software written in a systems programming language.*

1.5 Research Questions

To elaborate on, and find an answer to our problem statement, we need to target our testing to help us explore the surrounding topic. These research questions aim to concretize the specifics of the problem, and allow us to build a factual background before attempting to answer the problem statement.

> **RQ1:** How effective are source code analysis tools at detecting and describing security vulnerabilities in NetSurf?
>
> **RQ2:** How effective is fuzzing at uncovering security vulnerabilities in NetSurf, in comparison to source code analysis tools? Are the benefits of fuzzing worth the additional costs?
>
> **RQ3:** Are we able to find software vulnerabilities in the chosen parts of the NetSurf web browser project?

1.6 Scope

To focus our work on the most relevant and interesting software components, we have decided to include the following parts of the NetSurf project in our scope.

- `netsurf/netsurf` - The main web browser; The core logic of NetSurf, and all the different frontends.
- `netsurf/libhubbub` - An HTML5 compliant parsing library
- `netsurf/libsvgtiny` - An implementation of `SVG Tiny` for NetSurf
- `netsurf/libdom` - An implementation of the `W3C DOM`
- `netsurf/libcss` - A CSS parser and selection engine

We chose these components because they are highly relevant to the browsers operations, as well as appearing manageable and understandable. They cover a substantial part of NetSurf, with the largest component being `netsurf`, consisting of over 300.000 *lines of code (LOC)*. We will keep referring to each directory or component in lowercase, and the NetSurf project stylized in this way, for the rest of this thesis.

1.7 Thesis Outline

The outline and structure of this thesis is based on NTNU's guide on how you structure an empirical thesis [56]. This thesis consists of the following six chapters:

1. **Introduction** - This chapter will describe the background for the thesis topic, including presenting how we selected our target. This chapter will also explain the relevance and purpose for conducting our research. We will then go on to present the problem statement we want to investigate, as well as the research questions we have chosen to assist us in better understanding our problem statement, before setting the scope for our thesis.

2. **Theory** - In this chapter, we will present and explain the central theoretical concepts we will utilize in our thesis. We will describe fundamental theory around the tools we will utilize in our methodology, as well as introducing concepts these tools rely on to function. This chapter will include the explanation of theories surrounding penetration testing, source code analysis, and fuzzing, as well as giving an overview of some important types of vulnerabilities.

3. **Method** - This chapter will describe our methodological approach to penetration testing, highlighting how we will use the results from SAST and fuzzing. We will describe how we set up our working environment to ensure reproducibility of our research. This chapter will include the steps for setting up our SAST tools Snyk, Semgrep, and Coverity, as well as how we conducted our fuzzing campaigns with Domato and AFL++. It will also specify how we go about categorizing and analyzing the findings discovered through both static analysis and fuzzing.

4. **Results** - In this chapter we will provide an analysis of our data by presenting, explaining, and evaluating our findings. We will provide an overview of our findings from both SAST tools and fuzzing, as well as showing statistics based on these findings. After categorizing the exploitable vulnerabilities, we will present Proof of Concept exploits for the most important issues, in addition to pointing out some of the non-exploitable findings presented by the SAST tools.

5. **Discussion** - This chapter will include our discussion of the findings and their implications. Based on our results, we will explain how they can answer our problem statement, in addition to discussing the answers to our research questions. The chapter will also provide an evaluation of our research, to asses its validity and reliability, including our biases, potential error sources, and need for further research.

6. **Conclusion** - In this chapter we will approach a conclusion based on our results and discussion from the previous chapters.

Chapter 2

Theory

2.1 Penetration Testing

Penetration testing encapsulates many different tools and techniques that are similar to the Tactics, Techniques, and Procedures (TTPs) potential attackers might use [57]. In essence, penetration testing can be defined as an authorized simulated attack on a computer system to evaluate its security and find vulnerabilities [58]. In addition, a penetration test might include *Proof of Concept (PoC)* exploits to prove and demonstrate that the discovered vulnerabilities are exploitable [59]. A penetration tester could also utilize other methods, such as social engineering and physical penetration testing, with the goal of compromising a business and their computer systems [58]. In this thesis, however, we will solely focus on the technical aspects of penetration testing software.

Penetration testing can be roughly divided into *black box* and *white box* testing. These are defined by how much knowledge the penetration tester has of the target system [60]. During black box testing, the penetration tester has very limited knowledge of the system, and must gain understanding of the system by seeing how it responds to input, often using automated tooling. A more thorough and complete penetration test can be achieved with black box testing, as it gives the tester full knowledge of source code, running services, documentation, and whatever else might be needed, including information that would not be available to a real external attacker.

2.1.1 Common Methodologies

While there exists a wide range of different methodologies when it comes to penetration testing, all specialized to handle different targets, they mostly follow the same format. According to IBM, one of the first and most important parts of a penetration test is setting the scope [61]. The scope should define when, where, and how the test will be performed, and an exact definition of the target that will be tested. The scope includes if it is a black- or white box test, as well as deciding on tools and techniques to be used. We will use source code analysis, which is a

black box testing method further described in Section 2.2, and fuzzing, which can be both a black- and white box testing method further described in Section 2.3.

When executing a penetration test it is common to follow a specific set of steps to ensure that the penetration test follows a set of standards. A typical standard to follow is the Penetration Testing Execution Standard [62], that encapsulates steps from intelligence gathering to exploitation. Normally, gathering information about the target is the first step after pre-engagement, to build a background for laying a strategic testing plan [63].

As the purpose of this thesis is to compare SAST and fuzzing, the most important steps for this thesis are intelligence gathering, vulnerability analysis, and some exploitation [62]. General procedures for responsibly disclosing vulnerabilities are described in Section 2.4.1, and how we plan to disclose our findings is described in Section 3.4.

2.1.2 Penetration Testing Use Cases

Conducting a penetration test can be a good way to prevent attacks and data breaches. Having a security testing company properly audit your code base or infrastructure can be expensive, but is potentially much cheaper than paying ransoms, reparations, and fines after an actual attack. A proper penetration test is much more comprehensive than a simple vulnerability assessment [57]. In many sectors, such as finance and health care, security audits can also be required by law or contractual agreements, ensuring public safety by avoiding dangerous attacks.

2.2 Source Code Analysis

Source code analysis is one of the most common methods used in a black box penetration test. This can be done manually, or using automated *static application security testing (SAST)* tools [64]. Using SAST tools is great for scalability, and excellent at discovering some common vulnerabilities such as buffer overflows and double frees. They are also very effective at checking software dependencies, finding known vulnerabilities in the supply chain, to recommend updating or replacing third party libraries and utilities. SAST tools are often used by developers for these purposes, as they can be run often, even automatically in a *Continuous Integration and Continuous Deployment (CI/CD)* pipeline, to catch small errors before they become serious problems in production code.

Automated source code analysis tools can become less accurate in complicated systems where multiple components interact, making the results more difficult to predict. This is where manual source code analysis and fuzzing both shine. They can take a lot of time, but can be very effective at finding complex issues that are difficult to spot using SAST methods. Using SAST tools are faster than manually reviewing code, but they can have issues with a large number of false positives [65]. Even with just a few false positives, a human has to go through the findings to verify each one, quickly losing some of the time gained by using the SAST tool in the first place.

2.2.1 Selection of SAST Tools

The choice of tools depends on the structure and programming language of the specific application. Different tools also have different rates of false positives, some are easier to set up than others, and some have significant licensing costs. The OWASP Foundation's list of source code analysis tools [64] contains many of the most used tools for various applications and purposes. It contains several tools that are specifically aimed at applications written in C, such as Coverity, HCL AppScan, and CodeSonar.

Another useful source when it comes to selecting the correct tools, is Gartner's magic quadrants. Gartner has a specific magic quadrant for many different fields, including application security testing, as shown in Figure 2.1. While the tools shown in the figure are not specifically for source code analysis, it can still provide a good overview over potentially suitable tools.

Figure 2.1: Magic Quadrant for Application Security Testing [66]

2.2.2 Advantages of SAST

Despite the limitations that come with static application security testing, there are still many of advantages to using it. Most prevalent is the ability to discover potential vulnerabilities early in the product life-cycle. This is often referred to as *Shift*

Left Security, and while not a major part of this thesis, it is still an important reason for why SAST is a relevant security measure. In short, Shift Left Security revolves around integrating security earlier in a development lifecycle [67] to enable developers to address issues early, which reduces overall cost [68]. By implementing SAST, one becomes more proactive during development, in terms of vulnerability management. In addition, Shifting Left may also increase the resiliency and effectiveness of the code [67].

Another major advantage to SAST is that it is lightweight and efficient. Most of the time, getting started with a SAST tool is simple and not very time-consuming. A scan for most modern day SAST tools usually take at most a couple of minutes, depending on the size of the code base which is being scanned [69].

2.2.3 Disadvantages of SAST

In addition to SAST tools having limitations when it comes to detecting complex vulnerabilities, it is important to note that they also have other disadvantages. The most prominent issue that arises when SAST tools are involved, is the amount of false positives. SAST tools are often missing important context, as they are unable test the application's functionality, but rather just analyze the assumed behavior [65]. A possible consequence of this is that a vulnerability is reported, because the code contains a potential flaw, even if there is no other code to trigger the vulnerability in practice. Prior research has shown that using SAST tools to analyze certain C programs can result in a large number of false positives, even rendering them equivalent to random guessing [70, p. 18].

Another shortcoming of SAST tools is that they cannot discover runtime issues [71]. This is mainly a trade-off for being able to use these tools at any stage in the development process. Since SAST tools are designed to analyze source code, it is also left blind to configuration errors [72]. Such issues do not manifest from the source code alone, and can usually not be caught by source code analysis, but rather through manual review or fuzzing.

2.3 Fuzzing

In addition to the static tools described in the previous sections, we will use fuzzing, which is a more dynamic approach to explore and exploit a target application. On a high level, this means that instead of looking at the source code, the *fuzzer* can interact with the target by running the live application and observe its output. Fuzzing is the most common technique used to automatically discover software bugs [73]. By working with a compiled target binary, the fuzzing utility can be more flexible and adaptable to any sort of source, as opposed to SAST tools which need to be purposely built for each programming language.

One of the most popular tools for this sort of fuzzing is called American Fuzzy Lop (AFL), originally developed by Michal Zalewski at Google. It uses a new type

of compiler-based instrumentation and genetic algorithms to generate and discover clean and interesting test cases, substantially improving the efficiency and coverage from this type of fuzzing [74]. AFL, and derivatives such as *American Fuzzy Lop plus plus (AFL++)*, includes a highly optimized and relatively user-friendly fuzzer and a suite of related utilities aimed at security researchers. AFL++ is an evolution of the original project, with improved tools and techniques that improves performance and quality in almost every step of the process [75].

According to the AFL++ documentation, the process of fuzzing an application is divided into three steps [76]:

1. Compile the target source code with a special compiler that also prepares the target for fuzzing, for example by embedding coverage metrics. This step is called “instrumenting” the target.
2. Prepare the fuzzing by selecting and optimizing the *input corpus* for the target. The corpus must be a set of valid input files that can be used by the target application.
3. Perform the fuzzing of the target. This part is entirely automated, and works by taking a file from the corpus, randomly mutating it in some way, running the application on the new file, and observing the output. Crashes and hangs are recorded and saved, together with the new coverage data, that is also used as feedback to generate new inputs. This step is illustrated in Figure 2.2.

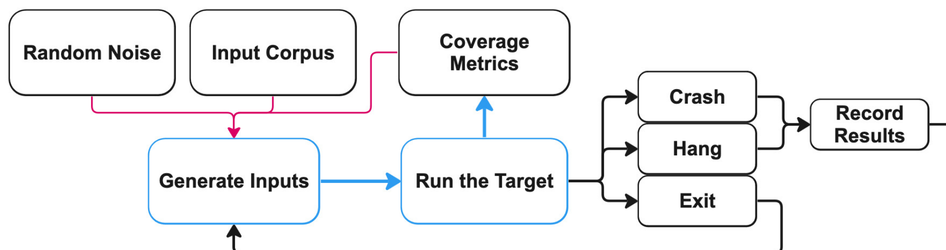


Figure 2.2: The Binary Fuzzing Pipeline

Given a set of input files and an instrumented binary, AFL++ will try to to continually modify the input such that all paths, branches, cases, and edge cases within the program are eventually visited. This can be a resource intensive, but effective way to discover bugs that SAST, manual unit tests, and integration tests might have missed.

All of these steps are essential to achieve an efficient fuzzing process. While most of these steps consists of concepts that might be familiar to most developers, *instrumentation* is probably not. Although some fuzzing is possible by analyzing and modifying existing program binaries, having the source code available and compiling it yourself makes the entire process easier, and yields better results. The *compiler* that ships with AFL++ will, in addition to actually compiling the target,

embed code that reports its status back to the fuzzer. For most types of fuzzing, *coverage* is an essential part of this instrumentation, describing what parts of the program is actually used and executed in each run. For example, if all executions until now have taken the `false` branch of a particular if-statement, and a new input file causes the `true` branch to run instead, it means we found a new *edge*, a previously unvisited part of the binary. When searching to find as many bugs as possible, the fuzzer will attempt to visit all of these edges, thus increasing coverage.

To build an efficient input corpus, it is imperative to know exactly what that means. The corpus is, in short, where test cases are stored [7], more specifically the set of files the fuzzer uses as the basis for new inputs. One should aim to fulfill a few important criteria when making or finding the files for the corpus. Most notably, the files should represent typical data that the application would expect to receive [76], with little overlap or repetition, while retaining a small file size. While some of these factors seem to be incompatible, like covering the largest amount of code while keeping the size minimal, AFL++ ships with a suite of tools to help with building a balanced corpus.

One way to generate a broad corpus is to start with a small sample of good inputs, and then programmatically mutate the inputs which are given to the fuzzer. The mutator will preprocess the corpus by creating new files with slight changes in the input to increase the covered area [77].

Having many large input files in the corpus takes more time and memory, as the target will have to go through more data in every one of the many thousands of executions. Many resources and utilities in the AFL++ project are dedicated to aid with creating small and efficient corpuses, as it is crucial for good performance. The AFL++ project recommends using `afl-cmin` to trim down the corpus by removing excess files, followed by `afl-tmin` to minify each of the remaining files to be as small as possible [76]. This is very similar to what we ended up doing in Section 3.3.2 to generate an efficient corpus. In total, this means that the input corpus should be broad and cover as much of the target functionality as possible, while remaining compact without repetition or additional noise.

2.3.1 Advantages of Fuzzing

Fuzzing is a type of dynamic analysis that is not dependent on how the source code is written. This means that it can detect vulnerabilities that are very difficult to find by reading the source, like complex nested functionality or pointer arithmetic. The fuzzer does not need to understand what the program attempts to do, it can simply test inputs and measure the outcome.

Fuzzing automates a lot of the work that would require manual analysis and review, by trading manual labor for a large amount of computing power. Even when using techniques like SAST, that already speeds up the process of finding issues in source code, a lot of work is still required to process and assess the findings they produce. When using fuzzing, you know that all the findings are actual issues that can occur in the compiled binary, and not false positives originating from misunderstandings or far-fetched theoretical issues.

2.3.2 Disadvantages of Fuzzing

While fuzzing can uncover many useful findings, it can still be a complicated and expensive endeavor. As most types of fuzzing involves some sort of iterative process that is continually working through the target, it is inherently resource intensive. A fuzzer might need to execute the target millions of times over the course of several weeks to test parts of a program, requiring expensive hardware, power, and time.

A fuzzer like AFL++ relies on feedback from the target, specific details of the input corpus, and also randomness to generate new inputs, meaning that it is nondeterministic and unpredictable. As opposed to the predictable nature of a static analysis tool, where you quickly get a report of the number and types of potential findings that you need to go through, a fuzzer may not give any output for hours, days or even weeks between crashes. The user has little control over the focus or direction of the fuzzer or how long it takes. This makes makes it difficult for a developer or business to plan or budget towards fuzz testing, as it always comes with some uncertainty.

Finally, fuzzing can only directly uncover issues that actually lead to practical problems when running the application. If an application requires user input, long-running work, interacting with an external system, or other similar features, it cannot be fuzzed properly without writing a test harness that wraps and simulates these inputs. Code issues like breaking best practices or style guides, as well as problems that are not directly reachable, cannot be discovered by fuzzing. Problems like these should also be fixed, as they might manifest as bugs and vulnerabilities in the future, but they can be undetectable by simply running the compiled binary.

2.4 Vulnerability Overview

In our testing, we are focusing on memory management vulnerabilities, as the goal of the thesis is to investigate methods for discovering vulnerabilities specifically in systems programming languages. Exploitation of such vulnerabilities aims to take advantage of the program logic to leak information or take control of the program. In this section, we discuss different ways of categorizing and identifying vulnerabilities . We then go on to present some common vulnerabilities and vulnerability types, to give an idea of the kind of vulnerabilities we will be looking for. For some of the vulnerabilities, we will provide examples of exploitation techniques, to describe the consequences of successful exploitation.

2.4.1 Common Vulnerability and Exposures (CVE)

The *Common Vulnerability and Exposures (CVE)* database is a catalog of publicly known hardware and software vulnerabilities. After finding a vulnerability, one usually tries to disclose that vulnerability to the vendor, and requesting it to be

assigned a CVE record[1]. CVEs are often specific to a certain program, software, or hardware, and the same CVE is not usually found across different vendors.

In Figure 2.3, the lifecycle of a CVE is shown, from being found, until the CVE is published or rejected. When filing a CVE, it is important to contact the correct *CVE Numbering Authority (CNA)* and request a CVE ID. They will verify that the submission is filed in accordance with the appropriate requirements and disclosure policies [78].

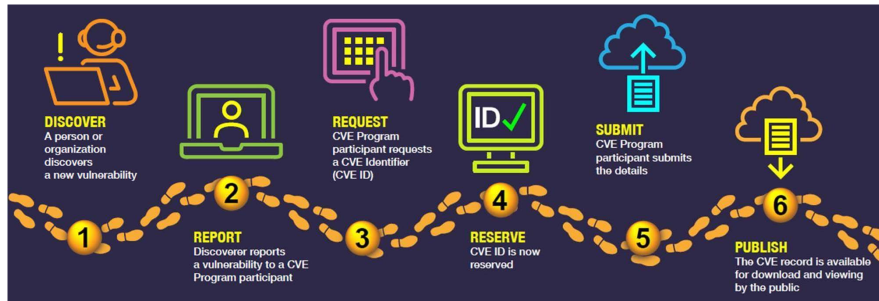


Figure 2.3: CVE Record Lifecycle - From the CVE Website [78]

Related Projects

In addition to the CVE database, there are several related projects that ensure vulnerabilities are categorized, logged, and published. Similarly to the CVE database, there is the *National Vulnerability Database (NVD)*. The NVD was created by the U.S. National Institute of Standards and Technology (NIST), and contains an entry for every CVE record along with additional resources for PoC exploits, documentation, and publications [79].

Vulnerability databases like CVE and the NVD often use the *Common Vulnerability Scoring System (CVSS)*. The CVSS is a scoring system that "can be used to score the severity of software vulnerabilities" [78]. A higher CVSS number indicates a more severe vulnerability with larger impact, attack surface, or other worsening quality, without needing to understand the details of the vulnerability. These resources are important tools for both penetration testers and unethical hackers, as well as developers wanting to secure their software from known vulnerabilities.

2.4.2 Common Weakness Enumeration (CWE)

Contrary to the CVE database that catalogs concrete vulnerabilities, CWE IDs are used to map general *weaknesses*. Rather than presenting and listing exploitable vulnerabilities, the "*Common Weakness Enumeration (CWE)* is a community-developed list of common software and hardware weaknesses" [80]. The main

[1] Structured data about a Vulnerability associated with a CVE ID

idea behind keeping a list of all the known weaknesses that could result in vulnerabilities is to enable developers to eliminate them before deployment [81]. Getting rid of a potential weakness is usually much easier and cheaper to do in the early stages of development, rather than having to create patches to fix a vulnerability after deployment.

The focus of the CWE list is to act as a preventative measure by assisting with identification of common vulnerabilities during the process of developing a product [81]. As shown in Figure 2.4, the cost of ignoring potential security vulnerabilities increases the further a product moves in the development process. The main goal of CWE initiative, according to the CWE Community, is to “stop vulnerabilities at the source by educating software and hardware acquirers, architects, designers, and programmers on how to eliminate the most common mistakes before a product is delivered” [81].

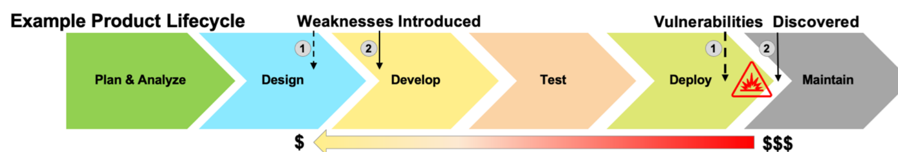


Figure 2.4: Example Product Lifecycle - From the CWE Website [80]

By utilizing CWEs, one can both increase security of a product, as well as reduce the potential costs that would arise from having vulnerabilities go unnoticed. As mentioned in Section 2.2.2, catching these potential vulnerabilities early in the development life-cycle is favorable, and the CWE initiative assists with uncovering potential risks early on, if implemented correctly. Several available modern SAST tools use CWE IDs to map vulnerabilities in a project. This provides developers with an explanation for why a code snippet triggered a warning.

While CWEs are very useful when organizing findings, it is important to remember that the different IDs are not mutually exclusive. For example, a stack buffer overflow, as will be described in Section 2.4.4, is categorized as CWE-121. At the same time, it involves writing outside the bounds of a buffer, making it an out-of-bounds write as well, which is identified by CWE-787.

2.4.3 Types of Vulnerabilities

When discussing software vulnerabilities, we mean any technical flaw resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of the system. While there is a myriad of different vulnerabilities and categories, we will discuss a few that are particularly relevant to our research. One common consequence is denial of service, which compromises the availability of a program or system. A relevant example of a denial of service attack could be a website crashing or freezing a browser.

Some systems implement protections from certain errors, such as *libc* detecting an illegal operation like a double free, killing the process. One way to handle these

situations is with a signal, such as a SIGABRT, which is raised when an exception is caught. A signal is handled by the operating system, taking control away from the process before stopping it. Another common signal[2] that can be raised when encountering a fatal problem is a SIGSEGV, often referred to as a "segmentation fault" or "segmentation violation". This is caused by the program attempting to access memory that is restricted or does not exist.

SIGABRTs and segmentation faults often lead to denial of service as the program terminates. If a situation leading to a segmentation fault can be controlled by an attacker, it can often lead to more serious exploitation. This type of exploit can be the most severe, as it may be escalated and extended to almost any type of attack, maybe even taking control of the entire computer without the victim noticing.

2.4.4 Analysis of Some Relevant CWEs

This section will describe five common weaknesses in detail, to provide an idea of the types of vulnerabilities we are looking for in this thesis. We will explain typical programming errors like integer overflows and improper null termination as well as stack and heap based vulnerabilities. Finally, we will discuss some advanced exploitation techniques for maintaining control of a vulnerable system. The vulnerabilities in the following sections are often complex and difficult to avoid.

Fortunately this is not the case for all vulnerabilities. A few examples of this are *division by zero* errors (CWE-369), *unchecked return values* (CWE-252), and *improper input validation* (CWE-20). To avoid division by zero errors, the programmer should check if the denominator is zero before attempting to use it, and throw an error if it is zero. If this is not done properly, the CPU will raise a floating point exception signal [82, p. 284], stopping program execution without exiting cleanly. Most library functions return values indicating different types of successes or errors. These return values should be checked to provide defined error handling. For input validation, the same return values can often be used. When input is read using `fread`, the return value will be the amount of bytes read into the destination buffer [9, p. 288].

Improper Null Termination (CWE-170)

In C, a string is a data structure defined by the ISO standard as "a contiguous sequence of characters terminated by and including the first null character" [82, p. 190]. Such strings are considered *null-terminated*. Common library functions such as `strcat`, `strcmp`, and `strdup` use this to decide how many bytes to copy, compare, and duplicate respectively. The `strlen`-function counts the number of bytes from a given address until the next *null byte*, thus returning the length of the string. The string length is often used as a parameter to other functions like

[2] A more complete overview of the different signals and their meanings can be found in the libc manual [9, p. 728-732].

`malloc`, meaning that missing null bytes could have great effects on program execution. We will present examples of this later, as copying the incorrect amount of bytes into a fixed buffer can cause buffer overflows and further problems.

Improper null termination can be avoided by using safe library functions such as `fgets`, which automatically null-terminates any input. If manual null termination is required, the programmer must be very careful to ensure that it is handled properly. Since this can be complicated and tricky to do manually, libc provides alternative functions like `strncat`, `strncmp`, and `strndup` to prevent programmer mistakes to cause unwanted overflows. Safety is provided by taking an extra argument n to determine the maximum length to copy, compare, or duplicate, to avoid writing outside the intended area. Functions like `strnlen` and `strndup` make sure to always null-terminate their target string, safeguarding against both invalid lengths and missing null termination. It cannot, however, be assumed that all string functions with a size argument have this functionality. The `strncpy`-function, for example, does not [9, p. 110].

Integer Overflow (CWE-190)

An integer overflow can occur when an arithmetic operation on an integer results in a value larger than the maximum or smaller than the minimum value the integer type is bound by. To demonstrate this we can attempt to do an addition by adding 200 to 200, using unsigned 8-bit integers. Since these integers consist of 8 bits, they can hold values from 0 to 255. If we were to add 200 to 200 (each represented in binary as the 8-bit value 11001000), the result of the arithmetic operation would be 400 (binary: **1**10010000). Since the integer can only hold 8 bits, only the rightmost 8 bits will be stored in memory, which would give the value 144, rather than the expected 400. This behavior can be used intentionally as a modulo operation, as long as the program prevents writing the overflowing part of the value into neighboring memory addresses.

The code and output in Listings 2.1 and 2.2 demonstrate the described integer overflow. If the code relies on the affected integer being 400, it could crash or cause new security vulnerabilities. If we were to overflow signed integers instead of unsigned integers, we would also have to take the sign bit into account. For example, adding 1 to 127, when stored as a signed 8 bit integer, would result in $127 + 1 = -127$, overflowing the variable.

Code listing 2.1: C Code Demonstrating an Integer Overflow

```
1  #include <stdint.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5      uint8_t important_number = 200;
6      printf("before addition: %hhu\n", important_number);
7      important_number += 200;
8      printf("after addition: %hhu\n", important_number);
9      return 0;
10 }
```

Code listing 2.2: Output Demonstrating an Integer Overflow

```
$ ./integer_overflow
before addition: 200
after addition: 144
```

Vulnerabilities related to integer overflows can be avoided by using macros like UINT8_MAX to clamp the value, and selecting appropriate integer types for variables based on their expected values [9, p. 612].

Stack Buffer Overflow (CWE-121)

The stack is an important segment of every process on most modern operating systems. It is a Last In First Out (LIFO) structure, meaning that new values are "pushed" onto the top of the stack, and the top value can be "popped" off, without the ability to read and write values at other positions in the stack. The *stack pointer* is used to keep track of the address of the last pushed value [83]. The purpose of the stack is to store local variables constrained to the scope of each function. In this section we will explain the stack and how it works, more specifically in Linux on the AMD64 architecture commonly found in modern computers with Intel or AMD processors, but the implementation of the stack is similar on other architectures and operating systems.

For each function call, a new stack frame is created, with enough space for the local variables in that function. This is demonstrated in Figure 2.5. The required space for a called function is subtracted from the base pointer and stored in the new stack pointer, to keep track of the bottom and top of the new stack frame [83]. Local variables are stored close to each other on the stack, and one could imagine that improper usage of one stack variable can impact neighboring variables.

When a function is called, the state of the previous function must be stored to be able to proceed from the same place once the function returns. This is done by storing the caller's *instruction pointer* at the top of the previous stack frame, and the caller's base pointer at the bottom of the new stack frame. When the called function returns, these values are loaded back into their appropriate registers, and the topmost stack frame is popped, resuming execution in the caller. Since the instruction pointer points to the next instruction to be executed, overwriting it results in full control of the program execution once the called function returns [84]. Changing the stored base pointer could also be dangerous, as it is used to determine the location of the caller's stack frame which again determines where to search for variables and the next stored instruction pointer when execution has been given back to the caller.

A stack buffer overflow involves writing outside of a designated buffer on the stack. A *buffer* is a section of contiguous memory, with a fixed size. For example, the C-code "`char text[5]`" defines a buffer named "text" with enough room for 5 characters, or bytes. If this code is placed within a function scope, it will be placed on the stack, within the top section marked "Local Variables" in Figure 2.5. As described previously in Section 2.4.4, a string is a series of contiguous bytes

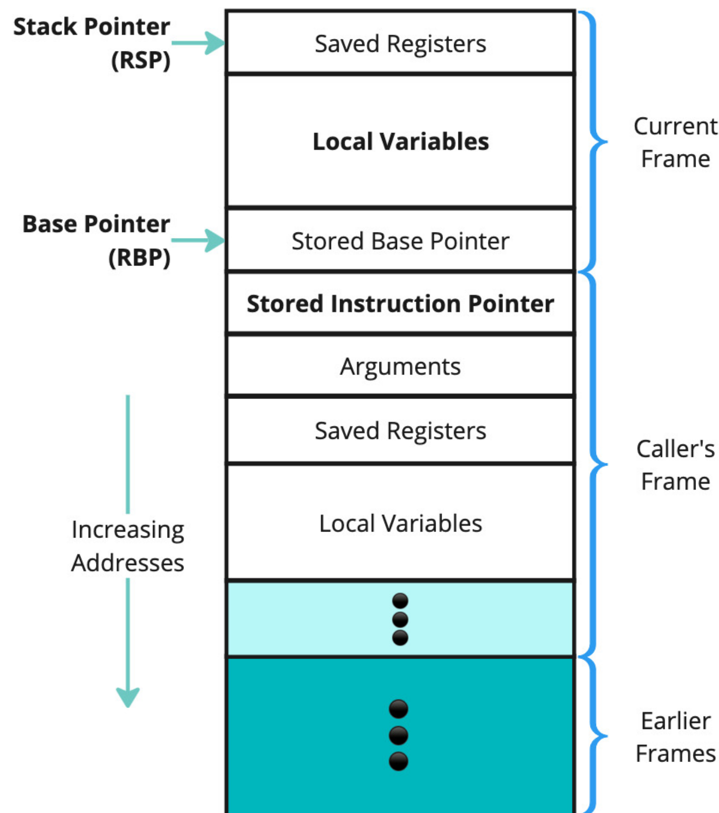


Figure 2.5: Illustration of Stack Frames [83]

terminated by a null byte. When copying a string into a buffer, it is important to validate that the length of the null-terminated string does not exceed the allocated length of the buffer.

When the program operates on buffers, either through function calls or regular instructions, it does not impose any limitations on the size of the buffer, unless this is explicitly specified by the programmer. Copying the contents of a large buffer into a smaller buffer will result in data being written past the end of the smaller destination buffer, essentially "overflowing" that buffer. When data is written to the memory area of other variables on the stack, the values of these variables will be changed. As shown in Figure 2.5, the stored instruction pointer is saved on the stack, somewhere below the local variables. When the top function returns, this value will be loaded back into the instruction pointer, resuming execution at that position. This means that a large stack buffer overflow in a local variable can allow overwriting other variables and then the stored base-, and instruction pointer.

An example of a simple buffer overflow on the stack is shown in Listings 2.3 and 2.4. As we can see, copying the large buffer overwrites the small buffer, and

places the address of the otherwise unused function where the previous instruction pointer is stored. This results in the unused function being called once the vulnerable function returns, demonstrating control of the programs execution. The required padding of 16 bytes to reach the stored instruction pointer can be calculated based on the 8 bytes for the small buffer and the 8 bytes for the stored base pointer between the local variables and the stored instruction pointer.

Code listing 2.3: C Code Demonstrating a Stack Buffer Overflow

```c
#include <stdio.h>
#include <string.h>

void unused_function() {
    puts("This function should never be called!");
}

void vulnerable_function(char* large) {
    char small[8];
    strcpy(small, large); // copy the string stored in "large" into "small"
    return;
}

int main(int argc, char* argv[]) {
    char large[64];
    fgets(large, 64, stdin); // read at most 64 bytes into the buffer "large"
    vulnerable_function(large);
    return 0;
}
```

Code listing 2.4: Output Demonstrating a Stack Buffer Overflow

```
$ objdump -t stack_overflow | grep unused_function
0000000000401146 g     F .text  0000000000000016   unused_function
$ echo -e "AAAAAAAAAAAAAAAA\x46\x11\x40\x00" | ./stack_overflow
This function should never be called!
zsh: segmentation fault  ./stack_overflow
```

Use After Free (CWE-416)

While the stack is used for local variables with fixed sizes, the heap is used for dynamically allocated variables that may have varying sizes, such as user input [9, p. 46]. When required, programmers can allocate *heap chunks* with specified sizes, which are used to hold memory until it is no longer needed, upon which the memory should be freed. Using libc functions, heap chunks can be allocated with `malloc` or its derivatives and freed with `free` from `stdlib.h` [82, p. 362].

Heap chunks are created using `malloc`, which returns a pointer to that chunk. Once the use of a heap chunk is finished, that chunk should be freed, to be stored into *tcache bins* for later reuse [6]. However, the pointer returned from `malloc` still exists in the variable that was used. If this memory is used again after being freed, it is called a use after free. This allows an exploit to edit the information used by libc to keep track of free heap chunks, and influence where new heap chunks will be created.

The example in Listing 2.5 demonstrates this with a minimal example in C. The exploitation technique shown is called tcache poisoning[3]. For a less cluttered demonstration, we have linked our binary with glibc 2.31[4], but the technique still works very similarly on modern versions of libc.

Code listing 2.5: C Code Demonstration of a Use After Free Vulnerability

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

void unused_function() {
    puts("This function should never be called!");
}

char* heap_chunks[2];
int chunk_count = 0;

void create_chunk() {
    heap_chunks[chunk_count] = malloc(32);
    chunk_count++;
}

void edit_chunk(int index, char* input) {
    strncpy(heap_chunks[index], input, 31);
}

void delete_chunk(int index) {
    free(heap_chunks[index]);
    // missing heap_chunks[index] = NULL
    chunk_count--;
}

int main(int argc, char* argv[]) {
    char input[50];
    char *action;
    while(1) {
        fgets(input, 50, stdin);
        action = strtok(input, " ");
        if (strncmp(action, "create", 5) == 0) {
            create_chunk();
        } else if (strncmp(action, "edit", 4) == 0) {
            char* index = strtok(NULL, " ");
            edit_chunk(atoi(index), strtok(NULL, " "));
        } else if (strncmp(action, "delete", 6) == 0) {
            delete_chunk(atoi(strtok(NULL, " ")));
        }
    }
}
```

The program in Listing 2.5 stores the heap pointers returned from `malloc` in the `heap_chunks`-array. They are still stored there after being freed in `delete_chunk`, and can therefore still be edited using `edit_chunk`. The script for exploiting the code in Listing 2.5 can be found in Listing 2.6, where we use python with *pwntools* to interact with the program.

[3]For a more thorough explanation of tcache poisoning, see [85]

[4]Linking with a custom libc can be done using patchelf [86]

Code listing 2.6: Python Code Exploiting a Use After Free Vulnerability

```
1  from pwn import process
2
3  p = process("./use_after_free")
4
5  p.sendline(b"create") # create the first chunk
6  p.sendline(b"create") # create the second chunk
7  p.sendline(b"delete 1") # free the second chunk
8  p.sendline(b"delete 0") # free the first chunk
9  p.sendline(b"edit 0 " + b"\x38\x40\x40".ljust(8, b"\x00")) # edit to point at
       ↪ fgets@got
10 p.sendline(b"create") # create chunk
11 p.sendline(b"create") # create chunk at the address of fgets@got
12 p.sendline(b"edit 1 " + b"\x16\x12\x40".ljust(8, b"\x00")) # edit fgets@got to
       ↪ point at unused_function
13
14 p.interactive()
```

The exploit script first allocates two chunks before deleting them. When they are freed, the two free chunks are inserted into the linked list of the corresponding tcache bin. The last chunk to be freed will be placed at the front of the linked list, with a pointer to the previously freed chunk, as shown in Figure 2.6. When a chunk with the same size is allocated later, libc will reuse the freed chunk. It then uses the pointer inside the reused chunk to know where the next free chunk in that tcache bin is. Since the pointer stored in `heap_chunks[0]` still points to the first chunk, this can be edited by the program to control where that chunk is created.

```
0x22cf2a0 0x0000000000000000 0x0000000000000031 ........1.......
0x22cf2b0 0x00000000022cf2e0 0x00000000022ce010 ..,.......,.....   <-- BinType.TCACHE
0x22cf2c0 0x0000000000020000 0x0000000000000000 ................
0x22cf2d0 0x0000000000000000 0x0000000000000031 ........1.......
0x22cf2e0 0x0000000000000000 0x00000000022ce010 ..........,.....   <-- BinType.TCACHE
0x22cf2f0 0x0000000000000000 0x0000000000000000 ................
0x22cf300 0x0000000000000000 0x000000000001df01 ................   <-- Top chunk
```

Figure 2.6: Freed Chunks in Tcache Bins

This gives the attacker the opportunity to edit any writable section of memory. An exploit could use this to overwrite data in the program's memory, change program logic, or take control of the program's execution flow. When attempting the latter, a typical target is the *Global Offset Table (GOT)*, which is used to resolve library functions to their actual memory address at runtime. Listing 2.6 shows an example exploit that edits the pointer in the last freed chunk to point to the address of `fgets` in the GOT.

It then creates a new chunk, causing `malloc` to expect the next free chunk to be located at `fgets@got`, as demonstrated in Figure 2.7. Once that is achieved, editing a new allocated chunk will change the pointer `fgets` is resolved to, which is currently the one on the right in Figure 2.7.

```
pwndbg> bins
tcachebins
0x30 [  1]: 404038 (fgets@got[plt]) —▸ 0x7f4eccfea630 (fgets)
```

Figure 2.7: Linked List of Free Heap Chunks after Reusing One of Them

The exploit proceeds to create another "chunk", now at the address of `fgets@got`, and edits this pointer to point at the address of `unused_function`. The address of the unused function can be found with the same method shown in Listing 2.4. Now that `fgets@got` points to `unused_function`, any call to `fgets` will be resolved to `unused_function`, resulting in the output seen in Figure 2.8. Since `fgets` is called at the start of each iteration of the while loop in `main`, we know that our new target will be called some time after `edit_chunk` returns.

```
$ python3 uaf_exploit.py
[+] Starting local process './use_after_free': pid 414985
[*] Switching to interactive mode
This function should never be called!
```

Figure 2.8: Output Demonstrating a Use After Free vulnerability

Double Free (CWE-415)

Another serious heap vulnerability happens when the same memory address is `free`d twice without first being allocated as a new chunk. The most severe consequences from this vulnerability are mitigated in later versions of libc. When a double free happens on modern Linux systems, the program will be stopped with a SIGABRT signal, as shown in Figure 2.9, limiting the consequences to a denial of service.

```
free(): double free detected in tcache 2
Aborted
```

Figure 2.9: Output from a Double Free

However, the consequences on older or less secure systems can be far more severe. According to MITRE, double free vulnerabilities can provide attackers with write-what-where gadgets and arbitrary code execution [34], giving them the same severity as serious use after free vulnerabilities.

2.4.5 Program Control

Previously, we have described how exploits like buffer overflows and use after frees can be used to affect program flow to call unintended functions, but it might not be clear why that is useful to an attacker. In the following sections, we will describe how and why an attacker might want to use these techniques in combination with crafted payloads to extend the possible consequences into taking and

keeping control of the attacked system. One common objective is to spawn a shell on the system, giving the attacker access to run any commands as if they were the attacked user.

Shellcode

One such technique is using shellcode. This can be done after a stack buffer overflow as demonstrated, already in 1996, in Phrack Magazine's *Smashing The Stack For Fun And Profit* [84]. This has since been made harder using techniques such as *No-eXecute (NX)* bits and Address Space Layout Randomization (ASLR) [87, p. 10]. The NX-bit is used to control whether a memory section is executable, and ASLR is used to randomize the base of the binary's virtual memory address space. With modern protections, a threat actor might have to change the NX-bit of the stack and leak a memory address to bypass ASLR, but the main concepts of the technique remain the same.

Return-Oriented Programming (ROP)

Even when there is no available memory segment that is both writable and executable, it is still possible for an threat actor to circumvent these protections and perform an attack. An example of this is using *ROP*-chains. ROP-chains chain together ROP-gadgets to create ROP-attacks. ROP-attacks take advantage of stack buffer overflows and the `ret`-instruction on Intel systems to place the addresses of multiple ROP-gadgets in a sequence on the stack, starting at the address of the stored instruction pointer. ROP-gadgets consist of code the threat actor would like the program to execute, followed by a `ret`-instruction to execute the next gadget. Every time the program returns, it places the address on top of the stack in the register for the instruction pointer, calling the next gadget. For many programs, combining the correct ROP-gadgets gives powerful control of its execution, and often enough control to create a shell on the target machine [88].

ROP-attacks are viable on other architectures as well, for example on ARM. On ARM systems, the `ret`-instruction does not pop the instruction pointer off the stack, rather loading the previous instruction pointer from a special link register, making the design of ROP-chains more complicated [89, p. 10]. Instead, it is often easier to take advantage of ARMs branching instructions in *Jump-Oriented Programming (JOP)* [90]. This involves loading the addresses of ROP-gadgets from the stack into the correct registers and branching to specific registers. JOP-gadgets can be chained together similarly to ROP-gadgets, and result in similar levels of control.

2.4.6 Other Vulnerabilities and Exploits

There are many other vulnerabilities and exploit techniques, in addition to the ones we have mentioned. Table 1 shows all the relevant CWEs that we have found in NetSurf from our testing. There are many other vulnerabilities and exploits like format string vulnerabilities [91] or SigReturn Oriented Programming [92], that can result in similar program control.

Chapter 3

Method

The goal of this section is to describe how we work with the subject matter to answer our research questions. We will describe our approach to penetration testing in general, how we will configure our working environment to work with the different tools and techniques we will employ, and how we will process the attained results. This is important to ensure reproducibility of our research, as well as being transparent about how we gather, consider, categorize, and use our different findings. In general, we will use the tools as recommended by their vendors and other security researchers, and try to establish a common system for using the results, so the different methods can be compared.

3.1 NetSurf and the Build Process

NetSurf is a cross-platform application built to run on many different UNIX-like systems, including older systems like RISC OS and AmigaOS, but also modern UNIXes like Linux and macOS. Although NetSurf includes some features exclusive to RISC OS, we have landed on a common middle ground using Linux. That means that we will target building the application with a *GTK* or framebuffer frontend, on top of a Linux environment. We do this to create a standardized environment to get comparable results from all of our testing, and because most of the tools we use are tested on and optimized for Linux. GTK3 is a free and open source widget toolkit used to build graphical user interfaces with a standardized look and good accessibility features. NetSurf can also render its interface using the "framebuffer" target, where it can draw directly to a display surface without relying on external graphics libraries or assets. GTK3 is probably the most commonly used and most user friendly version of NetSurf on Linux, while the framebuffer version is a faster and simpler build.

To extend this concept of a standardized and controlled environment, we need a predictable way to build and run NetSurf. NetSurf is a complex project written in C, and they have created a bespoke build system that configures and builds all the required libraries and subprojects. Building the application as an end user is however not complicated at all, as it is abstracted away behind a simple Makefile

for use with *GNU Make*. This still requires some third party libraries and utilities to be installed on your system, like `make`, `libgtk`, and `gperf`.

To simplify the install process even further, and to ensure that everyone on our team is working with identical builds of NetSurf, we decided to use *Nix*. Nix is a cross-platform package manager that can be used with their own Linux distribution *NixOS*, on top of other Linux distributions like Ubuntu or Arch Linux, or even on other UNIX-like operating systems like macOS or FreeBSD. Nix packages are written in a domain specific programming language, also called Nix. This functional programming language lets you define packages and environments with great control over the build process. A package is defined by its build phases (actions) and set of inputs, including the actual source code and all of the required dependencies. The build process, including patching, compilation, and linking, is done in a sandboxed, virtual environment, such that *only* the defined inputs can be used, with no regard for what other tools, packages, and libraries might be installed on your system. When using our *Nix flake* to build NetSurf, we know that we all use the exact same build of every library, compiler, and utility used to build NetSurf, regardless of the underlying system. This provides reproducibility, an important factor when finding, documenting and recreating bugs, ensuring that everyone can access the same exact application.

The previous version of NetSurf was already packaged in nixpkgs[1], which made it easy to build NetSurf 3.11, the latest release at the time of writing. The required changes mainly consisted of incrementing version numbers, updating the content hashes, and adjusting some dependencies, and they were immediately upstreamed into nixpkgs. After NixOS 24.05 is stabilized, anyone should be able to run "*nix-shell -p netsurf.browser*", and have `netsurf-gtk3` in their path, ready to run the latest version of NetSurf. This Nix package serves as a hackable base for our further research and modification, and lets us tinker with the code and build tools as we need.

3.2 Working with SAST

3.2.1 Tools

This section describes our suite of SAST tools and how they are configured. We have chosen an array of different tools, with different advantages, disadvantages, and business models. These tools aim to fill the same role, but work very differently to each other. They can for example vary in functionality, user interfaces, working principles and whether they are cloud-based or run locally.

[1] `https://github.com/NixOS/nixpkgs/blob/nixos-23.11/pkgs/applications/networking/browsers/netsurf/browser.nix`

Snyk

Snyk Code is the first SAST tool we will use in our testing. It is a "developer-first" SAST tool that provides real-time scanning [93]. Snyk Code is a cloud-based solution which reviews your uploaded source code and then analyzes, tests, and debugs it, providing a detailed overview over the security vulnerabilities it finds.

Snyk is often used in a commercial environment, which is one of the reasons we have decided to utilize their tool. They have overall good reviews and are used by large companies worldwide to test and secure their code [94]. In this thesis we, utilize the free version of Snyk Code, without access to any of their "premium features"[2]. Most of them would however not be useful in our thesis, as we are only using Snyk to perform source code analysis.

Semgrep

Our second SAST tool, and our only tool with a free and open source engine, is Semgrep. While Semgrep is still aimed at a commercial audience, it remains open source and publicly available. Semgrep describes their tool as a "fast, open-source, static analysis engine for finding bugs, detecting dependency vulnerabilities, and enforcing code standards" [95]. As the name suggests, coming from "semantic grep", semgrep is a text-based system using semantic analysis to understand the code and match it to a large set of predefined rules to identify bugs and vulnerabilities [96].

Coverity

Our final SAST tool is Coverity. Coverity, in particular Coverity Scan, is "a service by which Synopsys provides the results of analysis on open source coding projects to open source code developers" [97]. Coverity is Synopsys's SAST tool, listed in OWASP's list of Source Code Analysis Tools[3] as specific for C code. Synopsys is also listed as a leader in Gartners magic quadrant for application security testing [66], as shown in Figure 2.1. It is also the highest ranking tool in the quadrant.

Coverity provides a clear overview of the issues it finds, as well as displaying the *defect density* in the submitted project. With the possibility to assign potential vulnerabilities to specific team members, Coverity makes it easier for us to both spread the workload among the members as well as separate the vulnerabilities into categories that a specific member has more knowledge about. In addition, Coverity has a reasonable number of classification features which make it easier to manually rule out false positives that are not already ruled out by Coverity itself.

[2] `https://snyk.io/plans/`

[3] `https://owasp.org/www-community/Source_Code_Analysis_Tools`

3.2.2 Required Setup

In this section we will provide a description of the working environment and the required steps to use the different tools.

Snyk

To get started with Snyk, you only need to create an account and submit your code. The code can be uploaded directly in the web interface, or imported from a repository on GitHub. After importing the repository, Snyk Code will begin scanning the code for security vulnerabilities.

In addition to importing an entire repository, Snyk offers the possibility of excluding certain files or folders which you do not wish to scan. This is done by creating a `.snyk` file and excluding folders using a `.gitignore`-like syntax, shown in Appendix A.4. For our testing, we want simplify the review process by excluding everything outside our scope, following the official documentation[4].

Semgrep

Similarly to Snyk, the required setup for getting Semgrep up and running is quite simple, with the possibility to either scan your code locally, or use Semgrep in your CI/CD pipelines automatically. For this thesis, we opted for scanning our source code locally using the Semgrep *Command Line Interface (CLI)*, which only takes a couple of minutes. We can get started using Semgrep by installing the CLI tool through `pip`, signing in with `semgrep login`, and finally start the scanning with `semgrep ci`. When the scan is finished, the results are sent to Semgrep's servers, where it is made available in a web interface, similar to Snyk.

Like Snyk, Semgrep has the ability to exclude files with a `.semgrepignore` file, similar to a common `.gitignore` file. By following the official documentation[5] we were able to exclude the same files and folders that we exclude in Snyk, as shown in Appendix A.4. This is to ensure the tools are scanning the same files, to provide the most consistent findings.

Coverity

Coverity requires you to build the code you want to scan. Therefore, the required setup for Coverity is a bit more convoluted than for Snyk and Semgrep, but still not difficult. As described in Section 3.3.2, we usually use our Nix flake to build and test NetSurf, but we can still use GNU Make to build NetSurf with its original build system.

To ensure that Coverity is able to build properly, we first built NetSurf using the `make`-command and verified that it runs as expected. Following the setup guide

[4] `https://docs.snyk.io/scan-with-snyk/snyk-code/import-repository-to-snyk/excluding-directories-and-files-from-the-import-process`

[5] `https://semgrep.dev/docs/ignoring-files-folders-code/`

for Coverity and downloading the Coverity Scan Build Tool for C/C++, we issue the build command, now through the Coverity Scan Build Tool.

```
cov-build --dir cov-int make
```

After the build tool has finished successfully, you can upload the results for analysis to your project in the Coverity web interface. Whether or not the build succeeded can be verified by checking the build log with `tail ./build-log.txt` and seeing that the build succeeds.

Similarly to Snyk, Coverity allows you to create components to organize the potential security vulnerabilities based on where they can be found in the project. This allows us to focus on vulnerabilities found within scope, instead of having to sort through every potential security vulnerability Coverity uncovers across the entire project.

3.2.3 Managing the Results

This section will describe how we are going to handle, categorize, evaluate, and document the potential vulnerabilities found by these SAST tools. This includes how we are going to handle false positives and how we will present the potential vulnerabilities. We will keep track of vulnerabilities that might be exploitable by documenting them in a Kanban board in a repository hosted on a private Gitea server. This tool has been important for internal planning and timekeeping, but will not be presented in detail.

In Chapter 4 Results, we will present our findings as a reviewed and organized compilation of vulnerabilities from each tool. We will list potential vulnerabilities found in the NetSurf browser, as well as how these security vulnerabilities are ranked by each tool individually. We will also emphasize any overlaps and discrepancies between the tools, should that be relevant, and evaluate their overall accuracy.

All of our tools categorize the potential vulnerabilities they find in different ways, but they all do so by severity and type. We will refer to each type of vulnerability by their respective CWE ID and name, as each tool has its own set of names for different vulnerabilities . When different CWEs are closely related, and used interchangeably about the same vulnerabilities , we have combined them into one to make the results more readable. This overview of vulnerability categories is shown in Table 1 in the CWE List. All the tools do however categorize their results into low, medium and high severity findings.

When working with SAST tools, handling false positives turned out to be a large and important task. As mentioned in Section 2.2.3, SAST tools interpret source code and makes assumptions, which in turn can lead to false positives [65]. How we are to report false positives in an organized manner was a major point of discussion when we established this thesis, and we ended up with the following format in categorizing our findings.

We will not split the findings by false positives and actual vulnerabilities , but by useful and useless findings. We will put the useless findings aside, as they do

not lead to actual vulnerabilities or problems that are relevant to a penetration test. These useless findings will be the findings that are not currently exploitable, as well as all findings that are unlikely to become vulnerabilities in the future. We are making this distinction as the terms true and false positives are not sufficient to describe the findings we would like to use during a penetration test. A SAST tool's claimed finding may be correct, but if we do not deem it helpful in reaching increased security, through the methods described in Section 2.1, we will categorize it as useless.

The useful findings are the findings that are currently vulnerable, or increase the likelyhood of new vulnerabilities being introduced. We will further categorize these findings by their respective CWE IDs, and note where a finding could be represented by more than one CWE.

We will demonstrate the most serious or interesting useful findings through examples, including the vulnerable code, and a PoC exploit for the most serious ones. For the useless findings, we will go into detail and showcase some common issues that best exemplify how the SAST tools struggle when it comes to comprehending convoluted code. This will showcase some of the steps we take to confirm or refute the validity of the findings our SAST tools present, when it comes to the findings representing a vulnerability or not.

3.3 Working with Fuzzing

3.3.1 Tools

Fuzzing is an entirely different way of finding bugs and vulnerabilities in software. It works by generating a series of inputs, executing the program and then observing how the program reacts to it. For this project, we will be working with two different types of fuzzing; a general binary fuzzer called AFL++, and Domato, a "fuzzer" specifically for testing web browsers. In addition to these base tools, we will use *Docker*, *Docker Compose*, and Nix to build and automate NetSurf, the fuzzing tools, and the rest of our working environment.

3.3.2 Required Setup

Build System

As we will get into these types of fuzzing, we are going to need to build and install NetSurf. We considered the different techniques in question, together with our goals and objectives, and found that we need several different feature sets to perform all of our tasks. In total, we found that we need to use these three different builds to be able to efficiently run our suite of tools:

- A "normal", graphical browser for user-friendly testing. This would be most useful in the beginning and for verifying samples of later results with visual feedback, instead of blindly trusting the results from automated testing.

- The fastest possible version of NetSurf, for manual and automatic tests. This can be achieved by using framebuffer mode with the "`-f ram`"-flag to avoid spawning and maintaining a graphical window, and by modifying the source code in such a way that NetSurf automatically quits as soon as the page is finished loading.
- A "fuzzable" build, compiled with AFL++ instrumentation. This will allow AFL++ to automatically run NetSurf and record coverage data.

With the original buildsystem and Makefiles that ship with NetSurf, we would have to clear our working environment, modify the browser code to toggle JavaScript and self-closing, and then rebuild for the appropriate frontend target, whenever switching between any of these versions. When doing many iterative changes to our tests, this would take a lot of time, and would be prone to human error. To fix this, we modified the Nix-based build system described in Section 3.1, to have simple options to enable and disable these settings. When a derivation, such as a package, is built, it is placed in a unique location in the Nix store. This gives us several advantages over the default build system, most notably that we can use several different builds at the same time, without them colliding, and that each step of the build process is cached. Even though `netsurf-fb` and `netsurf-gtk` use different frontends, they use the same HTML parser, PNG renderer, and so on, and these only have to be built once, saving a lot of time. This package configuration can be seen in Listing 3.1, an excerpt from the Nix flake shown in full in Appendix A.5.

```
24 netsurf-gtk3 = with pkgs; (lib.recurseIntoAttrs
25  (callPackage ./netsurf-nix { })).overrideScope' (final: pref: {
26     ui = "gtk3";
27     enableDebugging = true;
28 });
29
30 netsurf-fb = with pkgs; (lib.recurseIntoAttrs
31   (callPackage ./netsurf-nix { })).overrideScope' (final: pref: {
32     ui = "framebuffer";
33     enableDebugging = true;
34     forceEnableScripting = true;
35 });
36
37 netsurf-fb-afl = with pkgs; if (lib.hasSuffix "linux" system) then
38         ((lib.recurseIntoAttrs
39         (callPackage ./netsurf-nix { })).overrideScope'
40         (final: prev: {
41     ui = "framebuffer";
42     enableDebugging = false;
43     exitWhenDone = true;
44     forceEnableScripting = false;
45     stdenv = prev.stdenv_afl;
46 })) else { };
```

Code listing 3.1: Packages Section of flake.nix

The options `forceEnableScripting` and `exitWhenDone` are created by modifying the relevant parts of NetSurf's source code, and putting the changes behind new `#ifdef` directives. When Nix is building the project, we set a flag in `make` that adds the corresponding preprocessor-definitions to conditionally compile with these features. The result is that `netsurf-gtk3` is entirely unchanged from its original codebase, but `netsurf-fb-afl` automatically quits as soon as the page finishes loading.

Domato Preparations

When working with a *DOM* fuzzer, our process consists of generating the input files, running the application on the given input files, and analyzing the applications behavior and potential crashes.

For this part of our testing, we wanted to familiarize ourselves with the process by starting out manually, one test case at a time. We started out with the first step, and needed to find a way to generate web pages as HTML files that are technically valid but still likely to hit a wide range of features and edge cases. One of these tools is Domato, a free software tool by Google Project Zero. Domato consists of a python application, some templates, and a set of grammar files. It generates a user specified series of random, but valid files that conform to the specified grammar. The default configuration describes an HTML file with separate sections for CSS styling, JavaScript code, and HTML markup, each defined in their own grammar files. These templates describe a range of valid test files to challenge the DOM parser, renderer, and JavaScript engine of the browser.

After downloading Domato and installing the required dependencies, we can run the following commands to generate our initial set of input files:

```
$ mkdir inputs
$ python generator.py -i 1 -m generate -n 1000 -o inputs
```

This generated a thousand files numbered as *inputs/fuzz-00000.html*, *fuzz-00001.html*, and so on. We found that a couple hundred to a thousand files covered all of the interesting behavior described in the default template files. The template and grammar files describe a wide range of valid HTML markup, CSS styling and JavaScript code that can make up a website. Domato's objective is to cover as many DOM-related features as possible in different sections of the generated files. These different sections can range from very simple elements like text paragraphs, to complex systems of SVG graphics with nested paths and shapes. Listing 3.2 shows an excerpt from the first file, showing the basic structure of the files generated by Domato.

Code listing 3.2: Some Samples from a File Generated by Domato

```
<html>
<head>
<style>

#htmlvar00004 { orientation: auto; background-blend-mode: normal; -webkit-mask: url
    ↪ (#svgvar00005); -ms-text-combine-horizontal: all; border-left: red thick
    ↪ groove; outline-style: hidden;  }
[ Several hundred similar lines ]

</style>

<script>

/* newvar{htmlvar00001:HTMLTimeElement} */ var htmlvar00001 = document.
    ↪ getElementById("htmlvar00001"); //HTMLTimeElement
[ Several hundred similar lines ]

try { var00004.setProperty("-webkit-column-fill", "auto"); } catch(e) { }
try { if (!var00005) { var00005 = GetVariable(fuzzervars, 'SVGPoint'); } else {
    ↪ SetVariable(fuzzervars, var00005, 'SVGPoint');  } } catch(e) { }
[ Several hundred similar lines ]

</script>
</head>
<body onload=jsfuzzer()>

<time id="htmlvar00001" datetime="2000-02-01T03:04:05Z" name="fN][kPI#^W(Y+d;29"
    ↪ muted="muted" classid="&#x27;f0&quot;#~|9tSN1*c84}r58" usemap="#htmlvar00003
    ↪ " itemtype="_:[&#x27;R](ucPk" width="0">q (!AVIY&#x27;T&gt;~</tt>
[ Several hundred similar lines ]

</body>
</html>
```

In our initial round of testing, we built and ran netsurf with GTK3 to get some graphical feedback to show that everything was working. Upon running "netsurf-gtk3 -v ./fuzzing/inputs/fuzz-00000.html", a window pops up, showing the netsurf browser as in Figure 3.1, and our terminal shows some warnings of unsupported JavaScript methods. The output does not look particularly beautiful, as it is randomly generated by Domato, but shows that the browser reads our file and processes it without throwing any errors.

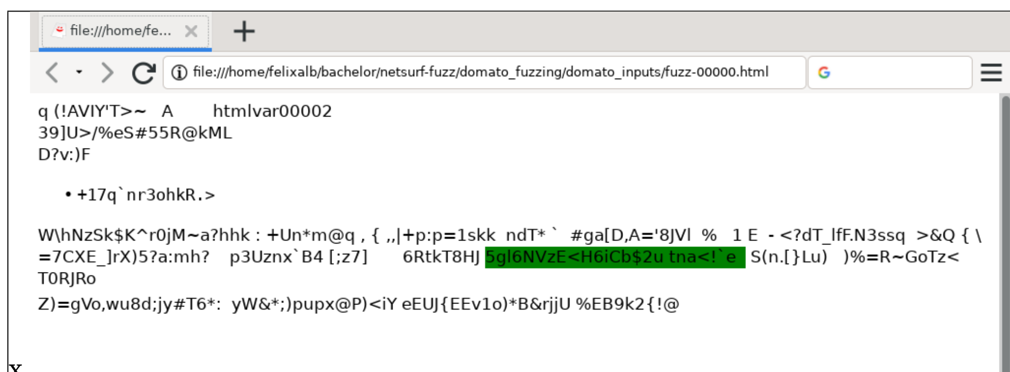


Figure 3.1: Screenshot of fuzz-00000.html in netsurf-gtk3

After running NetSurf manually on a couple of these files, we quickly decided to automate the process to handle the rest of the test cases. We used the bash script shown in Listing 3.3 to step through the files and sort them into their potential failure modes.

Code listing 3.3: Bash Script to Process the Files Generated by Domato

```
#!/usr/bin/env -S nix shell ..#netsurf-fb.browser --command bash

# Run NetSurf on each of the files in domato_inputs/, sorting them into
# valid, crashing and timeout based on their behavior.
# Requires a build of NetSurf with javascript enabled,
# and with automatic exit when the page is loaded.

mkdir dom_crash
mkdir dom_timeout
mkdir dom_valid

# Number of seconds to wait before killing the process
TIMEOUT=3

CRASH=0
TIMEOUT=0
VALID=0

for inputfile in domato_inputs/fuzz-* ; do
  echo "testing $inputfile"
  timeout 3 netsurf-fb -f ram "file://$(realpath $inputfile)" 2>&1 >/dev/null
  case $? in
    # Status code for success
    "0")
      ((VALID++))
      cp "$inputfile" dom_valid
      ;;

    # Status code when "timeout" stops the process
    "124")
      ((TIMEOUT++))
      cp "$inputfile" dom_timeout
      ;;

    # Any other status codes means that netsurf crashed for some reason
    *)
      ((CRASH++))
      cp "$inputfile" dom_crash
      ;;

  esac
done

echo "== Testing complete =="
echo " - $VALID valid"
echo " - $TIMEOUT timed out"
echo " - $CRASH crashed"
```

AFL++ Preparations

As described in Section 2.3, fuzzing with AFL++ requires us to instrument the target by compiling it with the custom compiler that comes with the fuzzer.

We made our instrumented binary build of NetSurf by using the existing build system, but replacing the usual C compiler, *GNU Compiler Collection (GCC)*, with a special AFL++ version of *clang*. AFL++ recommends the `afl-clang-lto` compiler, based on the *LLVM* toolchain, compiling C code down to machine code with embedded reporting of code coverage, errors, and other information that the fuzzer can use to analyze the program.

We first tried building NetSurf with the original build system, except swapping out the default compiler with AFL++ itself. Using a non-standard compiler is usually done by setting the CC and CXX variables when running `make`. In this case, however, it seems like make-flags could only affect compilation of the code found in `./netsurf/`, and not the supporting structure like `libdom` and `libhubbub`, which are also important when fuzzing. After building NetSurf with make, `afl-fuzz` quickly stated that every single test file caused the same execution path, and that the fuzzer could not find any new edges. This is probably because the "netsurf core" was only reading the file, finding out that it was HTML, and then passing it on to the DOM parser, renderer, CSS library and so on (without instrumentation), so every one of our test files actually behaved the same in this top level. To work around this, and actually instrument the entire tech stack bechind NetSurf, we used Nix again, with the code shown in Listing 3.4.

```
aflcc = pkgs.wrapCCWith rec {
  cc = pkgs.aflplusplus.overrideAttrs (final: prev: {
    postInstall = prev.postInstall + ''
      ln -s $out/bin/afl-clang-lto++ $out/bin/cc++
      ln -s $out/bin/afl-clang-lto $out/bin/cc
    '';
  });
  bintools = pkgs.llvmPackages_15.bintools;
};

[...]

stdenv_afl = (with pkgs; overrideCC llvmPackages_15.stdenv aflcc);
```

Code listing 3.4: AFL-based CC and stdenv Defined in netsurf-nix/default.nix

Here, we have defined a new "standard environment" (stdenv), what Nix calls the basic build environment for most packages, containing tools such as a C compiler, GNU Make, coreutils like *ls*, *cat*, *mkdir*, and tools like *patch* for working with source code. As opposed to the GCC, afl-clang-lto is built on clang, a C compiler based on LLVM. To create our new stdenv, we create a "CC" configuration, which borrows the linker, archiver, and everything else in "bintools" from LLVM, except that the actual compiler part is from the AFL++ package. In the last line of Listing 3.4, we take the entire default stdenv from llvmPackages, and keep everything except the actual compiler, which we replace. When we override the stdenv in

Listing 3.1, line 43, we enable this change for the `netsurf-fb-afl`-package only, while the other packages are built with the default toolchain, usually `gcc.stdenv` or `llvm.stdenv`.

With the package `netsurf-fb-afl.browser` defined in `flake.nix`, we are able to compile a version of NetSurf with AFL++ instrumentation and without JavaScript, that automatically exits when the page is finished loading. This is a suitable target for fuzzing, as it will process an HTML file as fast as possible, report its state back to AFL++ and exit. Fuzzing takes a long time, and an instance of AFL++ only uses a single CPU core at a time. Because our computers have several CPU cores, we can make use of AFL++'s ability to run multiple jobs in parallel, utilizing more of the available compute power for faster fuzzing.

To start running the fuzzing job on our servers, we wanted a setup that is easy to use on each machine, and decided to use Docker together with Nix to create our standardized working environment, and Docker Compose to orchestrate several containers together. We use AFL++ in two different modes of operations, one master that acts as the central controller that will keep track of the input corpus and make deterministic checks and changes while fuzzing, and several workers that receive instructions from the master instance and perform the random fuzzing [98]. Both of these types of instances run the exact same program, `afl-fuzz`, but with slightly different option flags. The AFL++ instances are defined in a simple Dockerfile shown in Appendix A.4, that only builds the Nix package described above.

Docker Compose creates the specified containers running the image described above, while also configuring the network, mounting volumes to supply the input corpus, and configuring the required settings. The "master" section of the compose file is shown in Listing 3.2, and the entire file is included in Appendix A.5.

```yaml
services:
  afl-master:
    build: .
    tmpfs:
      - /ramdisk
    environment:
      - AFL_TMPDIR=/ramdisk
    volumes:
      - type: bind
        source: ./fuzzing/inputs_x
        target: /fuzzing/inputs
      - type: bind
        source: ./fuzz-output
        target: /fuzzing/output
    network_mode: none
    stdin_open: true
    tty: true
    command: nix-shell -p aflplusplus --command "afl-fuzz -i /fuzzing/inputs -o /
        ↪ fuzzing/output -e html -M fuzz_master -- /netsurf/result/bin/netsurf-fb
        ↪ -f ram file://@@"
```

Figure 3.2: The master section of docker-compose.yml

To start fuzzing on an computer with 8 cores, we just have to install Docker, clone our repository[6], place our corpus into `./fuzzing/inputs_x`, and start fuzzing by running `docker compose up -d --build --scale afl-worker=7`. This will build the initial container image, and start one master with 7 additional workers. These instances will communicate with each other and utilize the full power of the processor for the fastest possible fuzzing.

When `afl-fuzz` is running, we can connect to its console and the command-line display shown in Figure 3.3. It shows how long the fuzzing process has been running, what technique it is currently using, and most importantly, the rate of new edges, hangs, and crashes. An *edge* is some conditional path in the program, like the result of an if-check, switch-statement, or similar. By visiting as many edges as possible, we gradually visit more of the application. As we hit new edges, there is a chance that we crash the program, for example by segmentation fault or failed assertions. When this happens, `afl-fuzz` will save the HTML file that caused the problem to a directory so we can inspect it closer.

```
american fuzzy lop ++4.08c {fuzz_master} (...f/result/bin/netsurf-fb) [fast]
┌─ process timing ────────────────────────────────────┬─ overall results ─┐
│        run time : 5 days, 18 hrs, 54 min, 57 sec    │  cycles done : 52 │
│   last new find : 0 days, 0 hrs, 1 min, 38 sec      │ corpus count : 24.0k
│last saved crash : 0 days, 0 hrs, 2 min, 25 sec      │saved crashes : 160│
│ last saved hang : 4 days, 17 hrs, 1 min, 22 sec     │  saved hangs : 500+
├─ cycle progress ─────────────────────┬─ map coverage┴───────────────────┤
│  now processing : 20.2k.2 (84.1%)    │    map density : 9.39% / 19.03%  │
│  runs timed out : 0 (0.00%)          │ count coverage : 6.49 bits/tuple │
├─ stage progress ─────────────────────┼─ findings in depth ──────────────┤
│  now trying : splice 9               │ favored items : 1015 (6.16%)     │
│ stage execs : 51/69 (73.91%)         │  new edges on : 2282 (13.86%)    │
│ total execs : 7.67M                  │ total crashes : 20 (20 saved)    │
│  exec speed : 22.89/sec (slow!)      │  total tmouts : 41.2k (0+ saved) │
├─ fuzzing strategy yields ────────────┴───────────┬─ item geometry ───────┤
│   bit flips : disabled (default, enable with -D) │    levels : 45        │
│  byte flips : disabled (default, enable with -D) │   pending : 12.8k     │
│ arithmetics : disabled (default, enable with -D) │  pend fav : 5         │
│  known ints : disabled (default, enable with -D) │ own finds : 15.1k     │
│  dictionary : n/a                                │  imported : 8886      │
│havoc/splice : 8375/3.05M, 6893/4.19M             │ stability : 57.64%    │
│py/custom/rq : unused, unused, unused, unused     ├───────────────────────┘
│    trim/eff : disabled, disabled                 │          [cpu000:100%]
└─ strategy: explore ────────── state: in progress ┘
```

Figure 3.3: AFL++ Status Output

AFL++ will not stop fuzzing on its own, but rather continue iterating over the input corpus until it is stopped manually. Over time, the fuzzer will cover different areas of the target application, potentially finding new hangs and crashes. However, as a larger portion of the possible discoverable issues are found, the rate of new hangs and crashes must go down. The diminishing returns of extended

[6] `https://github.com/felixalbrigtsen/netsurf-all`

fuzzing will make it more efficient to stop fuzzing at some point. This point can not be known beforehand, but one must use the statistics provided by AFL++, including those shown in Figure 3.3, coverage data from tools such as `afl-showmap`, and knowledge of the target to judge when fuzzing should be stopped, or when to move on to an entirely different input corpus. We will get into these specifics in Section 4.2.2.

3.3.3 Managing the Results

Compared to SAST, we found it more difficult to know what to expect from fuzzing. For this reason, it was difficult to plan the categories and severities before starting. To handle and organize the results from both types of fuzzing, we first need to narrow down where the issue originates by verifying a crash or similar problem, and building the minimal required steps to reproduce it. This will both help us in understanding the problem, and is required to file a bug report with the NetSurf developers.

From our knowledge of the different vulnerability classes described in Section 2.4, we know to be interested in crashes that can come from memory errors like buffer overflows. For example, if the fuzzer happens to crash with signal 11, it means a segmentation fault has occurred, caused by some sort of illegal memory access. That could point to a potentially dangerous vulnerability that could be used for something malicious like arbitrary code execution.

When a file causes a crash in NetSurf, we can use regular debugging tools to find out how and why NetSurf crashes. To demonstrate, we can take a look at the file “fuzz-00002.html”, which crashes with a segmentation fault, and find out why. We are running Linux, and have enabled the systemd feature that saves the state of the CPU and execution context whenever an application panics. This *core dump* can then be opened in *GNU Debugger (GDB)* to be inspected.

Code listing 3.5: GDB Backtrace After Opening fuzz-00002.html in NetSurf

```
$ netsurf-fb -f sdl file://$(pwd)/fuzz-00002.html
[1]    3159726 segmentation fault (core dumped)  netsurf-fb -f sdl file://$(pwd)/
    ↪ fuzz-00002.html
$ coredumpctl gdb
[ ... gdb output ommited ... ]
pwndbg> backtrace
#0  0x00000000005e86f4 in dom_string_byte_length ()
#1  0x00000000005e8732 in dom_string_isequal ()
#2  0x00000000005ece6c in _dom_element_get_attribute_node ()
#3  0x00000000005ec568 in _dom_element_lookup_prefix ()
#4  0x0000000000489dca in dom_node_lookup_prefix (result=0x7fffffff9358, namespace
    ↪ =<optimized out>, node=<optimized out>)
    at /nix/store/eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee-netsurf-libdom-0.4.2/include/dom
        ↪ /core/node.h:511
#5  dukky_node_lookupPrefix (ctx=0x93b950) at /build/netsurf/Node.bnd:356
#6  0x00000000004fd203 in duk__handle_call_raw (thr=thr@entry=0x93bdb0, idx_func=
    ↪ idx_func@entry=399, call_flags=<optimized out>, call_flags@entry=8) at
    ↪ content/handlers/JavaScript/duktape/duktape.c:68357
```

The GDB command `backtrace` prints out the call stack, meaning what nested functions were called leading up to the crash. Here, we can see that the actual segmentation fault happened in `dom_string_byte_length`, that was called from `dom_string_isequal`, and so on. As function call `#6`, shown at the bottom of Listing 3.5, is a general call to execute some JavaScript, we can see that the top level of JavaScript was called `lookupPrefix`. When searching in the original HTML file, we can see that out of the over 4000 lines of code, only one contains `lookupPrefix`, meaning we have probably narrowed down the search to that area. To pinpoint exactly where the potential bug is, we can now start removing parts of the HTML file, and check if the program still crashes. We can start out by removing quite big sections, even several hundred lines at a time, until NetSurf stops crashing or crashes somewhere else instead. This process requires several minutes of manual work per file, but eventually leads to a minimal reproducible example of each of the bugs. However, as we know that NetSurf actually crashes, there is no risk of this being futile work aimed at a false positive. At the end, we can replace arbitrary strings and variable names, like in this case renaming the variable `var00033` to `a`. The resulting HTML file only contains two lines of actual content:

Code listing 3.6: fuzz-00002.html Minified

```
<html>
<head>
<script>
  var a = document.createElementNS("http://www.w3.org/1998/Math/MathML", "mtr");
  a.lookupPrefix("b");
</script>
</head>
<body>
</body>
</html>
```

During our different stages of static and dynamic testing, we have also noticed that NetSurf contains a series of `assert`-statements. Assert is a special macro in C, that verifies that an expression evaluates to true before continuing. If the expression, for example “`base->components.path != NULL`” is indeed true, nothing happens and execution continues as normal. If it evaluates to false, a debug statement is printed to the standard error file, and the program is stopped with an abort signal, like shown in Figure 3.4.

```
netsurf-gtk3: utils/nsurl/parse.c:1476: nsurl_join: Assertion ‘base->components.
    ↪ path != NULL’ failed.
[1]    570343 IOT instruction (core dumped) netsurf-gtk3 file://$(pwd)/
    ↪ worker_1_crash_18.html
```

Figure 3.4: Program Stopped with Abort Signal

Because the NetSurf developers have implemented checks, in this case to verify that the component path actually exists as a non-zero pointer, it is technically *handled*. However, it is still a crash that will kill the entire browser, causing denial

of service, just like a segmentation fault would. We do consider these crashes as problematic, but not as severe as a segmentation fault, as this has practically zero potential for arbitrary code execution, leaks, or other attacks besides the denial of service.

Besides crashes, AFL++ also saves detected hangs separately. An execution is considered hanging when it has not exited after one second, the default timeout duration. As the absolute majority of our files, also ones containing several thousand lines of HTML, CSS, and JavaScript, run and exit in under 150 milliseconds, we think that taking over 6 times longer than that can be a sign of a problem.

3.4 Responsible Disclosure

In the event that we find exploitable vulnerabilities in NetSurf, we need to ensure that we disclose these vulnerabilities responsibly and correctly. We will practice what is called responsible or coordinated disclosure for the vulnerabilities we do find. This is a combination of private and full disclosure, where we will initially disclose the findings privately to the NetSurf Security Team, before making a full disclosure once a patch has been released [99]. Because the NetSurf project is maintained by volunteers in their spare time, we will suggest an embargo of 90-180 days before releasing the full details to the public.

During our possible disclosure, we will ensure that we provide as much information as we can. Providing a good report, with reproducible Proof of Concepts, makes it easier for NetSurf's developers to patch the vulnerabilities and speeds up the entire disclosure process. If applicable, we will apply for CVEs for the relevant vulnerabilities we find, to provide the public with necessary information to secure their systems. By doing this we follow ethical guidelines for responsible disclosure, and increase the security of NetSurf for everyone who uses it.

Chapter 4

Results

4.1 Source Code Analysis

In this chapter, we will present how the tools categorized the different vulnerabilities, how accurate they were, as well as presenting initial findings for each of the tools. When we started testing with SAST tools, we immediately noticed how quickly they were able to process a large code base. NetSurf consists of 587.571 LOC, and all our tools finished scanning in a few minutes. The `netsurf` component consists of a bit more than 300.000 LOC, which is reflected in the fact that most potential vulnerabilities shown in this chapter come from said component.

In Table 1, we have linked the CWEs with their official description. We also mapped the vulnerability type from each tool to their corresponding CWE, which can be see in Appendix A.1, A.2, and A.3. These appendices are used to showcase the different vulnerabilities and their types, as well as displaying the different CWEs found by each tool. As mentioned in Section 3.2.3, we will use the CWE descriptions when presenting the findings in more detail.

Comparing all the tools' findings together, dismissing CWE-398 and CWE-569, as we will explain in Section 4.1.3, our tools reported a combined total of 326 possible security vulnerabilities. After comparing the results from each tool with one another, there were in total 290 unique findings. This was not only because the tools found the same issues, but also because some of the tools reported the same issue more than once. We decided to remove these duplicates from the statistics moving forward, meaning that the percentage of useful findings will appear higher than what the tools originally reported.

4.1.1 Snyk

Snyk uncovered a total of 106 possible security vulnerabilities. As highlighted in Table 4.1, the vast majority of these potential vulnerabilities were located within the `netsurf` component, making up 104 out of the 106 possible security vulnerabilities discovered. Snyk also only assigned a single possible vulnerability with a "High" severity, as shown in Table 4.2. This is a clear difference to the other tools, as they were far more likely to assign a vulnerability a high severity rating.

An important thing to note, not only for Snyks results, but also for Semgrep, is that NetSurf includes some helper utilitities that were included by both SAST tools. These command line tools are not included in the actual build of the web browser component of NetSurf itself, but are rather used for code generation and testing. That means, that even though they are found in the component folders in our scope, potential vulnerabilities do not affect the security of the application.

Component	Amount of Possible Vulnerabilities	Percentage
netsurf	104	98.1%
libhubbub	0	0%
libsvgtiny	0	0%
libdom	0	0%
libcss	2	1.9%

Table 4.1: Overview of Possible Security Vulnerabilities - Snyk

Component	High Severity	Medium Severity	Low Severity
netsurf	1	80	23
libcss	0	2	0
Total	**1**	**82**	**23**

Table 4.2: Severity of Possible Security Vulnerabilities - Snyk

Categorizing the Initial Findings

A vast majority of the initial findings presented by Snyk were either false positives or “useless” in other ways. After thoroughly investigating each finding, and following each step presented by Snyk, we found that only 7 of the 106 findings were “useful”. This makes up for about 6.6% of all the findings, which turned out to be the lowest of all our tools. Table 4.3 shows the different useful findings found by Snyk.

UniqueID	CWE	Description
SNYK-1	CWE-170	Allocating exactly `strlen(str)` bytes for a string, such that it cannot be properly null-terminated.
SNYK-2	CWE-170	Possible improper null termination using `strdup`, where it was difficult to track down each usage of the function.
SNYK-3	CWE-170	Possible improper null termination using `strcmp`, instead of using `strncmp`.
SNYK-4 – 6	CWE-121	Originally categorized as CWE-170, this is actually a stack buffer overflow.
SNYK-7	CWE-125	Missing error detection, function should check the return value of `snprintf` before using it as an index to write to.

Table 4.3: Useful SAST Findings - Snyk

4.1.2 Semgrep

Semgrep found a total of 137 potential security vulnerabilities. In contrast to Snyk, Semgrep categorized 103 potential security vulnerabilites as “High”, making up 75.2% of all the potential vulnerabilities found, as shown in Table 4.5. For Semgrep, about 86.9% of all the potential vulnerabilities were located in the `netsurf` component.

Component	Amount of Possible Vulnerabilities	Percentage
netsurf	119	86.9%
libhubbub	0	0%
libsvgtiny	0	0%
libdom	8	5.8%
libcss	10	7.3%

Table 4.4: Overview of Possible Security Vulnerabilities - Semgrep

Component	High Severity	Medium Severity	Low Severity
netsurf	93	22	4
libdom	2	6	0
libcss	8	1	1
Total	**103**	**29**	**5**

Table 4.5: Severity of Possible Security Vulnerabilities - Semgrep

Categorizing the Initial Findings

Similar to Snyk, most of the findings presented by Semgrep were not very valuable. Still, they showcase the potential of SAST tools and what limitations and challenges they have when it comes to utilizing these tools for security purposes. In the end, we discovered a total of 6 useful findings, some of which were multiple instances combined into single entries in Table 4.4, making up about 7.9% of the 139 initial findings. This essentially means that Semgrep was just as effective, in numbers, at finding potential vulnerabilities as Snyk, but it was slightly more accurate. Table 4.6 showcases the different useful findings that had some security implications.

Unique ID	CWE	Description
SNYK-1	CWE-131	Memory allocation of string length, causing strings of certain lengths to be improperly null-terminated (Also found by Snyk).
SEM-1	CWE-14	`memset` can be optimized out since its argument is not used later in the same function.
SEM-2	CWE-14	The call to `css_bloom_init` is optimized out, `memset_explicit` should be used instead.
SEM-3	CWE-676	`strtok` is specified unsafe to call in multi threaded programs [9, p. 127]. It is used five times in the same function.
SEM-4	CWE-416	A use after free right before the program is terminated, without any current risk of exploitation.
SEM-5	CWE-125	Memory allocation sizes can be controlled with access to the file `/etc/mimefiles`.

Table 4.6: Useful SAST Findings - Semgrep

4.1.3 Coverity

In total, Coverity uncovered a total of 174 possible security vulnerabilities. The NetSurf Browser developers have already used Coverity to search for bugs and vulnerabilities , and these results are publicly available. However, it seems that they have stopped actively using these tools many years ago, and many newer changes were never triaged. Out of the 174 total findings, 10 were already marked as false positives, and 4 were considered intentional choices. We ignored these findings, and proceeded to go through the remaining 160.

Additionally, Coverity reported security vulnerabilities for CWE-398 and CWE-569, which are according to MITRE CWE categories and “[...] must not be used to map to real-world vulnerabilities” [100], [101]. Therefore, we decided to not directly categorize the items linked with these CWEs as vulnerabilities in this thesis, but rather as issues with the code itself, and will not include them in the

statistics of the tool findings. This is based on MITRE's own explanation of why CWE Categories should not be used for mapping vulnerabilities, which states that CWE Categories are informal groupings of different weaknesses that can assist with data aggregation, navigation, and browsing, but are not weaknesses in themselves [100], [101]. CWE-398 is the CWE category for "7PK - Code Quality", and CWE-569 is the CWE category for "Expression Issues".

Removing the potential vulnerabilities linked to CWE-398 and CWE-569, we were left with 83 potential vulnerabilities that had to be investigated, where 77.1% of them were located in the `netsurf` component of NetSurf. The distribution of these findings by severity is shown in Table 4.7 and 4.8.

Component	Amount of Possible Vulnerabilities	Percentage
netsurf	64	77.2%
libhubbub	6	7.2%
libsvgtiny	4	4.8%
libdom	4	4.8%
libcss	5	6%

Table 4.7: Overview of Possible Security Vulnerabilities - Coverity

Component	High Severity	Medium Severity	Low Severity
netsurf	29	33	2
libhubbub	0	1	5
libsvgtiny	1	3	0
libdom	4	0	0
libcss	0	5	0
Total	**34**	**42**	**7**

Table 4.8: Severity of Possible Security Vulnerabilities - Coverity

Categorizing the Initial Findings

In the end, Coverity uncovered a total of 26 useful findings, which is the most out of all the SAST tools we used. These useful findings make up just over 31.3% of the 83 potential findings, not counting the discarded findings for CWE-398 and CWE-569. This is an impressive percentage of useful findings, far ahead of both Snyk and Semgrep.

Unique ID	CWE	Description
COV-1 – 2	CWE-197	Storing epoch time[1] in signed 32-bit integers will cause some functions to break on January 19th, 2038. Coverity found two instances of this.
COV-3	CWE-197	Storing epoch time in unsigned 32-bit integers will break one function in year 2106.
COV-4 – 6	CWE-476	A possible NULL pointer dereference, due to missing checks. Coverity found three instances of this.
COV-7	CWE-476	Function might return a NULL pointer that is immediately dereferenced.
COV-8 – 9	CWE-561	Unreachable code behind an impossible condition.
COV-10	CWE-561	Unreachable code, as the function has already returned.
COV-11	CWE-252	After calling `g_unlink`, the return value is not validated
COV-12	CWE-252	Missing error detection for a possible `NOMEM` error returned by `realloc`.
COV-13	CWE-561	Unreachable code after an explicit return.
COV-14	CWE-561	An unreachable call to `free`, which could cause memory leaks.
COV-15	CWE-404	Potential resource leak, malloced variable goes out of scope.
COV-16	CWE-404	Small potential resource leak after downloading a file.
COV-17	CWE-119	Out-of-bounds write after allocating the wrong size for a buffer.
COV-18	CWE-119	Out-of-bounds write after allocating the wrong size for a buffer.
COV-19 – 26	CWE-561	Checking the expression value from `lwc_string_isequal` gives an illusion of error checking and dead code.

Table 4.9: Useful SAST Findings - Coverity

4.1.4 Useful Findings

Combining the results from all of the SAST tools, we have a total of 41 unique useful findings. If we compare this number to the total amount of unique findings by all the SAST tools, this makes up for about 14% of all findings. They range from programmer mistakes like unreachable code, to big security risks like stack buffer overflows. Some findings have simple explanations, such as SEM-3, where the manual pages shows that the `strtok` function is specified as `MT_Unsafe` [9, p. 127]. Combining this with the knowledge that NetSurf is multithreaded (MT), we can determine that this is a security risk. Another example that is easy to confirm is COV-1 – 2, where time stored in the epoch format in signed 32-bit integers.

This will cause an integer overflow on January 19th, 2038, with the potential consequences described in Section 2.4.4. While it will not be a problem for many years, there is no reason to not account for this now, and avoid having to handle it in the future.

Other findings however, are more complex. In the following sections, we will take a closer look at SNYK-4 – 6, SEM-2, SNYK-1, and COV-17. SNYK-4 – 6 is a stack buffer overflow, and the most serious vulnerability found. SEM-2 highlights a potential issue where the compiler might do optimizations that alter the behavior of the code. SNYK-1 involves a memory allocation where the size is controlled by the return value of `strlen`. COV-17 is an example of a minor out-of-bounds write, which is often a serious security issue.

Stack Overflow in idna_encode

SNYK-4 – 6 was the most serious of the findings. This is a serious vulnerability, which makes it concerning that Snyk gave it a very low score, and the other tools did not detect it at all. It was categorized as improper null termination, stating that if the input buffer to `strncpy` is not properly null-terminated, it could cause vulnerabilities. To find the serious vulnerability we had to do some more investigation, finding that while the input to `strncpy` is properly null-terminated, the length of the source buffer can be controlled by the user.

The code in Listing 4.1 is copied from the `idna_encode` function in the NetSurf source code [102, l. 694-743]. The vulnerability arises from the fact that `idna_encode` fully trusts the value of the length argument, without any further checks. This means that if the function were to be called with a `len` larger than 256, the `fqdn`-buffer would be overflowed.

Code listing 4.1: C Code from idna_encode

```
1  nserror idna_encode(const char *host, size_t len, char **ace_host, size_t *ace_len)
        {
2      [...]
3      char fqdn[256];
4      [...]
5      strncpy(fqdn_p, output, output_len);
```

Snyk was able to find multiple program flows leading to where `idna_encode` was called. To reach it, the program calls `nsurl_create`, which calls `nsurl__create_from_section` which then calls `idna_encode` when parsing the host part of the URL [103]. `nsurl_create` keeps track of the correct length of the host string, but does not impose any limits on it, which can lead to the discussed overflow when `idna_encode` is called.

Because `fqdn` is a stack variable, the exploitation process becomes very similar to the one described in Section 2.4.4. When NetSurf loads a URL, either through the address bar, a link or the command line, a call is made to `nsurl_create`. We wanted to show how easily a remote threat actor could exploit this vulnerability, using an embedded link. If a website includes an anchor tag, iframe or other element with a `href` or `src` attribute, for example through an *Cross-Site Scripting*

(XSS) attack, the value is parsed with `nsurl_create`, for example triggering the exploit shown in Listing 4.2.

Code listing 4.2: HTML Code Exploiting idna_encode

```html
<html lang="en">
<head>
    <title>Stack Buffer Overflow</title>
</head>
<body>
    <a href="http://
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAA%49%48%47%46%45%44">link</a>
</body>
</html>
```

This exploit demonstrates control over the instruction pointer by setting it to `0x444546474849`, using the techniques from Section 2.4.4. The correct number of padding bytes, in this case the 312 “A”-characters, from the start of `fqdn` to the stored instruction pointer was found using *pwndbg*. When viewing this HTML file in NetSurf, the overflow is triggered, and the result is shown at the bottom of Figure 4.1, demonstrating control of the instruction pointer.

```
Thread 1 'nsgtk2' received signal SIGSEGV, Segmentation fault.
0x0000444546474849 in  ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
────────────────[REGISTERS / show-flags off / show-compact-regs off]───────────────
 RAX  0x0
 RBX  0x6161616161616161 ('aaaaaaaa')
*RCX  0x555555d90150 ◂— 0x6161616161616161 ('aaaaaaaa')
*RDX  0x555555d9020f ◂—'
    ↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    ↪ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaIHGFED'
 RDI  0x7fffffffb640 —▸ 0x555555d90150 ◂— 0x6161616161616161 ('aaaaaaaa')
*RSI  0x7fffffffb5f0 ◂— 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaIHGFED'
 R8   0x0
 R9   0x7
 R10  0x555555853b80 (no_escape) ◂— 0x0
 R11  0x45
 R12  0x6161616161616161 ('aaaaaaaa')
 R13  0x6161616161616161 ('aaaaaaaa')
 R14  0x6161616161616161 ('aaaaaaaa')
 R15  0x6161616161616161 ('aaaaaaaa')
 RBP  0x6161616161616161 ('aaaaaaaa')
 RSP  0x7fffffffb620 —▸ 0x555555f68f40 ◂— 0x6161616161616161 ('aaaaaaaa')
 RIP  0x444546474849
─────────────────────────[DISASM / x86-64 / set emulate on]─────────────────────────
Invalid address 0x444546474849
```

Figure 4.1: Pwndbg Output after a Stack Buffer Overflow

Controlling the instruction pointer can give a threat actor access to arbitrary code execution on the victims machine [22]. Depending on which security measures are implemented on the victim's machine, the exploit can also be used for further escalation of access and privileges. Security measures like *stack canaries*, and NX-bits could mitigate the control gained with the stack buffer overflow.

Memory Allocation of String Length in gui_get_clipboard

SNYK-1 was the only vulnerability reported on by more than one tool, as it was also found by Semgrep. This vulnerability is categorized as low impact, but is still a verifiable problem that was detected by multiple tools. Snyk linked it to CWE-170, which seems accurate given the code in Listing 4.3, found in `selection.c` in netsurf [104, l. 40]. The return value from `strlen` is the number of bytes read before reaching a null character, excluding the null character [82, p. 380]. This means that for there to be room for the source string and an additional null character in the allocated heap chunk, the memory allocated is required to have the size `strlen + 1`, which is not the case in Listing 4.3.

Code listing 4.3: C Code from gui_get_clipboard

```
1 static void gui_get_clipboard(char **buffer, size_t *length) {
2     [...]
3         *length = strlen(gtext);
4         *buffer = malloc(*length);
5     [...]
6     memcpy(*buffer, gtext, *length);
```

The size of any heap chunk must be divisible by 16, with a minimum size of 24 bytes of actual data plus an additional 8 bytes for metadata [6, p. 3]. If we were to create a chunk with size 32, and then move a string with length 24 into it, it would not be null-terminated, as shown in Figure 4.2[2]. Carelessly printing this string would leak the next value on the heap, `0x31`. The code in Listing 4.3 is reachable through pasting a value into a web page. Pasting 24 bytes while debugging `netsurf` in pwndbg, with a *breakpoint* set in `textarea_keypress`, gives the output shown in Figure 4.3. We pasted 24 B's and, which lead to an additional byte being appended before passing the string to `textarea_replace_text` in `textarea_keypress` [105].

```
0x555555e1f140  0x00007fffe8008b80  0x0000000000000021  ........!.......
0x555555e1f150  0x4242424242424242  0x4242424242424242  BBBBBBBBBBBBBBBB
0x555555e1f160  0x4242424242424242  0x0000000000000031  BBBBBBBB1.......
```

Figure 4.2: Heap Contents after Copying String to Heap Chunk

[2]While the sections of 8 bytes (16 nibbles) are read from left to right, the contents of each section of should be read from right to left

```
► 0x555555736b70 <textarea_keypress+3808>    call   textarea_replace_text
    ↪                  <textarea_replace_text>
rdi: 0x555555e963d0 ◂— 0x0
rsi: 0x0
rdx: 0x0
rcx: 0x555555e1f150 ◂— 'BBBBBBBBBBBBBBBBBBBBBBB1'
r8: 0x18
r9: 0x0
```

Figure 4.3: Pwndbg Output before Call to textarea_replace_text

The appended byte "1" has the ASCII value `0x31`, and contains the chunk size of the next chunk on the heap. Adding any multiple of 16 to 24, below a maximum heap chunk size limit, would result in another viable length to paste, exploiting the same vulnerability, as the only requirement is that the heap chunk aligns as shown in Figure 4.2. This means that the size allocated when exploiting the vulnerability can be controlled by the attacker, providing more control over which values could be leaked. A particularly interesting target is the size of the current top chunk, which can be used by a threat actor to plan allocation sizes of malicious sizes during heap grooming [106, p. 80]. Fortunately, the length of the pasted string is currently handled by `textarea_keypress`, and the memory is not leaked to the user [105, l. 2450]. Nevertheless, we have included it as a finding as it should be fixed to avoid future issues caused by careless string handling.

Memset Removal in css_bloom_init

SEM-3 originates when the compiler detects that a call to `memset` is the last use of a specific variable. When the compiler encounters an operation, such as the call to `memset` within `css_bloom_init`, that modifies a section of memory that does not seem to be returned or used again in any way, it may choose to optimize it away. The `memset`-function is used to clear a section of memory, usually filling it with zeros. Semgrep claimed that this was happening in many places, and was correct in this instance. This is demonstrated in Figure 4.4, where we disassemble `_chain_bloom_generate` and look at the underlying code. There, we can see the absence of a call to `css_bloom_init` and any code to replace it before the `do-while` loop starts at `_chain_bloom_generate+19`. The code adjacent to the call to `css_bloom_init` in `_chain_bloom_generate` is displayed in Listing 4.4 [107, l. 731],[108, l. 181].

Code listing 4.4: C Code in _chain_bloom_generate

```
1  static void _chain_bloom_generate(const css_selector *s, css_bloom bloom[
       CSS_BLOOM_SIZE]) {
2          css_bloom_init(bloom);
3
4      do {
5          if (s->data.comb == CSS_COMBINATOR_ANCESTOR || s->data.comb ==
               CSS_COMBINATOR_PARENT) {
6              const css_selector_detail *d = &s->combinator->data;
7              do {
8                  if (d->negate == 0) {
9                      _chain_bloom_add_detail(d, bloom);
10                 }
```

```
pwndbg> disassemble _chain_bloom_generate
Dump of assembler code for function _chain_bloom_generate:
   0x000000000024b1b0 <+0>:     pxor   xmm0,xmm0
   0x000000000024b1b4 <+4>:     mov    r8d,0x1
   0x000000000024b1ba <+10>:    movups XMMWORD PTR [rsi],xmm0
   0x000000000024b1bd <+13>:    jmp    0x24b1c9 <_chain_bloom_generate+25>
   0x000000000024b1bf <+15>:    nop
   0x000000000024b1c0 <+16>:    test   rdi,rdi
   0x000000000024b1c3 <+19>:    je     0x24b248 <_chain_bloom_generate+152>
   0x000000000024b1c9 <+25>:    movzx  eax,BYTE PTR [rdi+0x30]
   0x000000000024b1cd <+29>:    mov    rdi,QWORD PTR [rdi]
   0x000000000024b1d0 <+32>:    and    eax,0x70
   0x000000000024b1d3 <+35>:    sub    eax,0x10
   0x000000000024b1d6 <+38>:    test   al,0xe0
   0x000000000024b1d8 <+40>:    jne    0x24b1c0 <_chain_bloom_generate+16>
   0x000000000024b1da <+42>:    lea    rax,[rdi+0x18]
   0x000000000024b1de <+46>:    jmp    0x24b1ed <_chain_bloom_generate+61>
   0x000000000024b1e0 <+48>:    cmp    BYTE PTR [rax+0x18],0x0
   0x000000000024b1e4 <+52>:    lea    rdx,[rax+0x20]
   0x000000000024b1e8 <+56>:    jns    0x24b1c0 <_chain_bloom_generate+16>
   0x000000000024b1ea <+58>:    mov    rax,rdx
   0x000000000024b1ed <+61>:    test   BYTE PTR [rax+0x19],0x2
   0x000000000024b1f1 <+65>:    jne    0x24b1e0 <_chain_bloom_generate+48>
```

Figure 4.4: Assembly Code for _chain_bloom_generate Displayed in Pwndbg

The compiler was correct that the argument "bloom" was not going to be used later in `css_bloom_init`. However, "bloom" was passed to `css_bloom_init` as a memory address, and will be used again in `css_bloom_generate`, at line 9 of Listing 4.4. It should therefore not be optimized out. The impact of this vulnerability is unclear considering the vast amount of possible flows to reach the code in Listing 4.4. MITRE describes the possible results as disclosure of potentially sensitive information as data is not zeroed as expected [16]. To prevent the compiler from optimizing it out, the call to `memset` can be changed to a call to `memset_explicit` to guarantee safety for the previously used data [82, p. 379].

Out-of-Bounds Write in fire_dom_keyboard_event

COV-17 is an off-by-one programmer error where a buffer has been allocated one byte less than it should. The mistake can be seen in Listing 4.5, taken from `html.c` [109, l. 176]. The `utf8_from_ucs4` function returns the length of the string placed in the variable `utf8`, with a maximum value of 6 [110, l. 56].

Code listing 4.5: C Code in fire_dom_keyboard_event

```
1 bool fire_dom_keyboard_event(dom_string *type, dom_node *target, bool bubbles, bool
      cancelable, uint32_t key) {
2     [...]
3     char utf8[6];
4     size_t length = utf8_from_ucs4(key, utf8);
5     utf8[length] = '\0';
```

When `utf8[6]` is initialized, 6 bytes are reserved on the stack. Since character arrays are *zero-indexed*, this makes the first byte accessible with `utf8[0]`, and the last byte accessible with `utf8[5]`. In the case that `utf8_from_ucs4` were to return 6, the last line in Listing 4.5 would set `utf8[6]` to 0, causing an out-of-bounds write. While this is just one byte outside of the allocated bytes, it could influence the value of short integers or booleans stored there, and potentially change to flow of the program. As with the vulnerability related to memory allocation of string length earlier in this section, we have not found any serious consequences from this small issue. However, it is still a problem that should be fixed, for example by simply making the `utf8` buffer one byte larger.

4.1.5 Useless Findings

While there were many findings that could be used in a penetration test, there were many false positives and other useless findings. Out of the 290 unique findings, 249 of them were what we categorized as "useless", making up the remaining 86% of all the findings. As we will demonstrate with a few examples, one strikingly common issue was the tools not understanding preprocessor statements. Another issue was overreporting, with many different findings being alleged from the same reported vulnerability.

Misunderstanding Preprocessor Statements

When compiling source code written in a language like C or C++, the compiler will first run a *preprocessor step*, that will evaluate statements like `#define` and `#ifdef`, that are only executed during build time and never at runtime. This step can create powerful macros and conditional code that can be changed before each build, leading to faster and more compact executables in the end [111]. However, code that is managed by the preprocessor can be confusing to both humans and software, like the SAST tools we are using, because the source code can look quite different to the code that is actually running.

Many examples of SAST tools misunderstanding these can be found in Coveritys reporting on `duktape.c`. On 12 occasions, Coverity claimed to have found an uninitialized scalar variable, potentially leading to inconsistency or modification of control flow [36]. However, in each instance preprocessor statements had been used to initialize this variable shortly before. One example can be seen in Listing 4.6 [112], where Coverity claims `tv_tmp` is uninitialized on line 78727.

Code listing 4.6: C Code in duk__handle_break_or_continue

```
78724 duk_tval tv_tmp;
78725
78726 DUK_TVAL_SET_U32(&tv_tmp, (duk_uint32_t) label_id);
78727 duk__handle_finally(thr, &tv_tmp, lj_type);
```

In this example `tv_tmp` has been initialized by `DUK_TVAL_SET_U32`, which is a preprocessor definition shown in Listing 4.7. The address of `tv_tmp` is placed in `duk__tv`, which is then used to initialize it. All of Coverity's 12 findings with uninitialized scalar variables in `duktape.c` were related to these definitions, with `DUK_TVAL_SET_OBJECT`, `DUK_TVAL_SET_STRING`, and `DUK_TVAL_SET_BUFFER` being just a few examples. How this affected our process operating with SAST tools will be discussed in Chapter 5 Discussion.

Code listing 4.7: C Code in DUK_TVAL_SET_U32

```
1 #define DUK_TVAL_SET_U32(tv, val) \
2         do { \
3                 duk_tval *duk__tv; \
4                 duk__tv = (tv); \
5                 duk__tv->t = DUK_TAG_FASTINT; \
6                 duk__tv->v.fi = (duk_int64_t) (val); \
7         } while (0)
```

To demonstrate that this issue is not unique to Coverity, we will also discuss an example with use after free findings in Snyk. Snyk claims that there is a use after free happening on line 9 and 12 of Listing 4.8. This happens twice in `hlcache.c`, leading to four false positives [113, l. 745].

Code listing 4.8: C Code in hlcache_handle_release

```
1   } else {
2      RING_ITERATE_START(struct hlcache_retrieval_ctx, hlcache->retrieval_ctx_ring,
            ictx) {
3           if (ictx->handle == handle && ictx->migrate_target == false) {
4               llcache_handle_abort(ictx->llcache);
5               llcache_handle_release(ictx->llcache);
6               RING_REMOVE(hlcache->retrieval_ctx_ring, ictx);
7               free((char *) ictx->child.charset);
8               free(ictx);
9               RING_ITERATE_STOP(hlcache->retrieval_ctx_ring, ictx);
10          }
11     } RING_ITERATE_END(hlcache->retrieval_ctx_ring, ictx);
12  }
```

The reason these are not actual use after free vulnerabilities is that `RING_ITERATE_STOP` and `RING_ITERATE_END` are not functions, but preprocessor macros that

do not actually use their arguments at runtime. They can both be seen in Listing 4.9 [114, l. 133]. Snyk reports that they are use after free vulnerabilities because `ictx` looks like it is passed to a function after it is freed, on line 8 of Listing 4.8. However, because `ictx` is passed to a preprocessor definition, this type of logic never reaches compilation. Instead, `RING_ITERATE_STOP(hlcache->retrieval_ctx_ring, ictx)` is replaced with `goto iteration_end_ring_ictx`, before compilation, and the address or value of `ictx` is not actually used when "calling" `RING_ITERATE_STOP`.

Code listing 4.9: C Code for RING_ITERATE_STOP and END

```
1  #define RING_ITERATE_STOP(ring, iteratorptr)    \
2          goto iteration_end_ring##_##iteratorptr
3  #define RING_ITERATE_END(ring, iteratorptr)            \
4                          } while (false);               \
5                          iteratorptr = iteratorptr->r_next;      \
6                  } while (iteratorptr != ring);         \
7          }                                              \
8          iteration_end_ring##_##iteratorptr:
```

Overreporting

Contributing to the large amount of useless findings were an unnecessary amount of findings being reported on the same alleged vulnerabilities. Many examples of this can be found in Semgrep's reporting of use after free vulnerabilities. For example, Semgrep reported 30 use after free vulnerabilities in `plot_alpha_bitmap`, based on the assumption that the variable `bitmap` could have been freed before the function was called. The code snippet in Listing 4.10 triggered 5 separate findings, one for each usage of `bitmap`. The overall result of this was Semgrep reporting 81 use after free findings in total, making the output cluttered, regardless of whether the findings were actually correct or not. Better methods for reporting this will be discussed in Chapter 5 Discussion.

Code listing 4.10: C Code in plot_alpha_bitmap

```
1  static nserror plot_alpha_bitmap(HDC hdc, struct bitmap *bitmap, int x, int y, int
       width, int height) {
2      [...]
3      if ((bitmap->width != width) ||
4          (bitmap->height != height)) {
5          NSLOG(plot, DEEPDEBUG, "scaling from %d,%d to %d,%d",
6                        bitmap->width, bitmap->height, width, height);
7          bitmap = bitmap_scale(bitmap, width, height);
```

Another common issue can for example be seen regarding the vulnerability described in Section 4.1.4, where many different paths leading to the same root weakness are reported as individual findings. While this was a somewhat[3] useful finding, Snyk reported on it multiple times. One finding was reported for each

[3]The reported line was part of a serious vulnerability but not to the extent or for the reason Snyk claimed.

place it found a call to `nsurl_create`, potentially leading to the vulnerable code being executed. This lead to an even larger number of reported findings per vulnerability, further cluttering the output and increasing our workload.

4.2 Fuzzing

Our fuzzing results are naturally divided into the results from a one-time execution of effectively thousands of tests with Domato, and the continuous process of binary fuzzing with AFL++. As opposed to the static analysis tools, we are not concerned with false positives or unlikely scenarios, as each of the reported findings, from both types of fuzzing, are actually confirmed and reproducible hangs or crashes. This means that our main focus will be on considering the type and impact of each vulnerability, rather than verifying its validity.

4.2.1 Domato

As described in Section 3.3.2, our fuzzing process began by generating a range of input files and then manually running and inspecting the first few files that were generated by Domato. That revealed that a surprisingly large portion of our files crashed NetSurf, resulting in segmentation faults. To quantify the scale of this problem, we ran the script as defined and explained in Listing 3.3 on a dataset of one thousand HTML files, yielding the results shown in Listing 4.11.

Code listing 4.11: Result Statistics from Testing with Domato

```
== Testing complete ==
 - 315 valid
 - 75 timed out
 - 610 crashed
```

As we can see, only 31.5% of our original input files were successfully rendered by NetSurf in under 3 seconds without crashing.

Looking into the new folder of files that cause crashes, we can try to see what type of errors are generated. We have gone through a sample of these files, and narrowed them down from around 4000 lines each until we get the smallest files that will reproduce the same issues with the same backtrace in GDB. As the files are randomly and independently generated, our selection of the files with the lowest identifiers are equivalent to a random selection.

In the same vein as Listing 3.5 and 3.6, respectively showing the backtrace and minimal cause of the crash occurring in `fuzz-00002.html`, we have looked into some of these files to get a better understanding of the cause. We found that most of these problems boiled down to issues in obscure JavaScript functions like `lookupPrefix`.

Code listing 4.12: Minimal Script that Causes dom_crash/fuzz-00003.html

```
var a = document.createElementNS("http://www.w3.org/2000/svg", "a");
var b = a.classList;
var c = b.value;
```

Code listing 4.13: Minimal Script that Causes dom_crash/fuzz-00016.html

```
var a = document.createElement("th");
var b = a.cellIndex;
```

When going through the backtrace of the various crashes, we can see that almost every single one is a variation on just a few different JavaScript functions. `Node.lookupPrefix`, `Node.classList` (as shown in Listing 4.12), and the different operations on a `HTMLTableCellElement` (like the `cellIndex` in Listing 4.13) seem to be the most common causes.

As over 60% of our tests lead to crashes, it might indicate that there are many, potentially dangerous, memory problems in NetSurf. However, we do not believe that is actually the case. For one, our Domato testing does not reveal 600 different vulnerabilities, but rather five to ten different bugs encountered many times. Even though these problems lead to crashes, we have not been able to exploit any of them beyond the denial of service effect that naturally follows a crash. The problems usually consists of null pointer exceptions or a read-loop where illegal memory is accessed. This means an attacker can use XSS or other initial attack vectors to crash your browser, potentially losing data, like form data open in another tab. This is still not a severe issue like remote code execution, and we don't think it's possible for an attacker to take control of your computer, steal your information, or cause any other type of damage.

Another import aspect to note in this context, is that JavaScript support in NetSurf is still experimental. Their documentation pages state that

> At present, NetSurf has only primitive and incomplete support for JavaScript, which we disable by default. Without JavaScript, NetSurf is able to provide access to most of sites on the World Wide Web. Some sites, however, will not display correctly or be unworkable due to heavy reliance on this standard. [115]

With this in mind, we deem these problems lower risk, as JavaScript has to be actively enabled.

We wanted to verify that the vast majority of these problems come from the Duktape JavaScript engine, an experimental feature, by doing the same tests without JavaScript. We ran the same 1000 HTML files on a build of NetSurf without JavaScript, with the results shown in Listing 4.14.

Code listing 4.14: Result Statistics From Testing with Domato, Without JavaScript

```
== Testing complete ==
 - 999 valid
 - 1 timed out
 - 0 crashed
```

Except for one timeout, we can see that none of the actual crashes originate from the parts of NetSurf that deal with file loading, HTML parsing, running the DOM, or rendering, but actually from the Duktape JavaScript engine.

4.2.2 AFL++

After we figured out multi-threaded fuzzing with AFL++, we began fuzzing on a few of our servers as described in Section 3.3.2, each running a different input corpus. In total, we have been fuzzing with AFL++ for between 1500 and 2000 core hours, on several servers running Intel Xeon processors from around 2012, with an estimated average rate of 20-23 executions of NetSurf per second per core. This is the equivalent of running a quad core CPU of similar speed at full utilization for two to three weeks.

When fuzzing, we monitored the output from AFL++, and saw a gradual decrease in the number of new crashes after a couple of days on each host. Using the utility `afl-showmap` bundled with AFL++, we found that this decrease occurred somewhere between 15% and 20% total coverage, where the rate of new edges started dropping from around 40 to around 20 new edges per hour. A run of `afl-showmap` is shown in Listing 4.15. After gathering and evaluating this data from our first few runs, we stopped subsequent rounds at around 15% coverage, before starting a new round with a different input corpus.

Code listing 4.15: Example Output from afl-showmap

```
[*] Reading from directory '/fuzzing/output/'...
[*] Scanning '/fuzzing/output/'...
[*] Scanning '/fuzzing/output/fuzz_0f9550a6487c'...
[*] Scanning '/fuzzing/output/fuzz_0f9550a6487c/crashes'...
[*] Scanning '/fuzzing/output/fuzz_0f9550a6487c/hangs'...
[*] Scanning '/fuzzing/output/fuzz_0f9550a6487c/queue'...
[...]
[*] Scanning '/fuzzing/output/fuzz_master'...
[*] Scanning '/fuzzing/output/fuzz_master/crashes'...
[*] Scanning '/fuzzing/output/fuzz_master/hangs'...
[*] Scanning '/fuzzing/output/fuzz_master/queue'...
[+] Captured 18786 tuples (map size 98464, highest value 255, total values
    47694564875) in '/dev/null'.
[+] A coverage of 18786 edges were achieved out of 98496 existing (19.07%) with
    171823 input files.
```

When analyzing the output from each of our instances of AFL++, we saw a total of 725 crashes and 6290 hangs. The crashes are presented in Table 4.10, where we break down the number of crashes by cause. The majority of these crashes come from failed assertions, meaning that the situation is handled by NetSurf, but a significant number of crashes come from severe issues like double frees and segmentation faults.

Count	Type	Cause
402	Abort	Assertion failed in nsurl_join
282	Abort	Buffer overflow, detected by libc
16	Abort	Assertion failed in box_textarea_create_textarea
12	Abort	Double free, detected by libc
12	Segmentation Fault	Illegal memory accesses described in Section 4.2.3
1	Abort	Assertion failed in bitmap_get_width

Table 4.10: Breakdown of the Crashes Discovered by AFL++

Many crashes were duplicates of the same actual problems, for example the stack buffer overflow in `idna_encode` when creating a URL with `nsurl_join` caused more than 100 of the buffer overflows, prompting us to restart that AFL++ process with a new input corpus to start a new path.

4.2.3 Findings

As described in the previous sections, there were many findings from AFL++ and Domato, and as they were found with fuzzing, we knew that none were false positives. The most severe vulnerability found by the SAST tools, the stack buffer overflow in `idna_encode`, was also found by AFL++ as described in Section 4.1.4. When AFL++ finds a crash or a hang, the input file that caused it is placed in the corresponding output folder. After minimizing and analyzing each of these files, we got a closer look at how these different problems work. In this section, we will go into a few of these crashes and why they can occur.

Double Free in box_normalise_table

The first vulnerability AFL++ found was a double free in `box_normalise_table`. It is possible that `col_info.spans` is first freed within `box_normalise_table_row_group`, before it is then freed again in line 4 of Listing 4.16 [116, l. 651].

Code listing 4.16: C Code in box_normalise_table

```
1  static bool box_normalise_table(struct box *table, const struct box *root,
       html_content * c) {
2      [...]
3      if (box_normalise_table_row_group(child, root, &col_info, c) == false) {
4          free(col_info.spans);
5          return false;
6      }
```

The function called in Listing 4.16, `box_normalise_table_row_group`, calls `box_normalise_table_row`, which subsequently calls `calculate_table_row`. Inside of `calculate_table_row`, we find the code in Listing 4.17 [116, l. 83]. From Listing 4.16, we see that if `box_normalise_table_row_group` returns false, `col_info.spans` will be freed, so it is important that it has not been freed previously.

Code listing 4.17: C Code in calculate_table_row

```
1  static bool calculate_table_row(struct columns *col_info, unsigned int col_span,
       unsigned int row_span, unsigned int *start_column, struct box *cell) {
2      [...]
3      cell_end_col = cell_start_col + col_span;
4      if (col_info->num_columns < cell_end_col) {
5          spans = realloc(col_info->spans, sizeof *spans * (cell_end_col + 1));
6          if (spans == NULL)
7                  return false;
```

The double free can happen because of what happens when `realloc` is called with its second argument, size, equal to 0. When this is the case, `realloc` becomes equivalent to `free` on the first argument, which is `col_info.spans` [9, p. 49]. The developers might not be aware of this issue, as the if-check before the call to `realloc` makes it seem impossible that `cell_end_col + 1` evaluates to zero, as long as `col_info->num_columns` is at least zero. This is not the case however, as an integer overflow can happen, as described in Section 2.4.4. If `cell_end_col` is `0xffffffff` on line 4 of Listing 4.17, adding 1 inside the if-check will overflow it to zero, causing `col_info->spans` to be freed and the return value to be false, causing another free. The described integer overflow can be caused by the input file shown in Listing 4.18. 4294967295 is the decimal representation of `0xffffffff`, the highest value that can be represented by a 32-bit integer.

Code listing 4.18: HTML Code Exploiting box_normalise_table

```
<html lang="en">
<head>
    <title>Double Free</title>
</head>
<body>
    <table><th colspan="4294967295"></th></table>
</body>
</html>
```

When opening this file in netsurf on Linux, libc detects a double free and kills the process with a *SIGABRT*. On systems with less protections, the consequences can be more severe, as described in Section 2.4.4.

Integer Overflow Causing an Out-of-Bounds Write

AFL++ found another integer overflow, with one of them leading to an out-of-bounds write. This integer overflow originates in `table_calculate_column_types` in `table.c` [117, l. 811], shown in Listing 4.19. If i is “1” and `cell->columns` is `0xffffffff`, the integer overflow will occur, causing a nearly infinite loop.

Code listing 4.19: C Code in table_calculate_column_types

```
1  bool table_calculate_column_types(const css_unit_ctx *unit_len_ctx, struct box *
       table) {
2      [...]
3      i = cell->start_column;
4          for (j = i; j < i + cell->columns; j++) {
5          col[j].positioned = false;
6          }
```

This vulnerability can be exploited very similarly to the previous double free, with `colspan=4294967295`. The only differences are some extra tags avoiding the specific flow leading to the `realloc`, as seen in Listing 4.20. Since `j` is incremented almost infinitely, line 5 of Listing 4.19 will eventually access memory outside of the `col`-array, causing a segmentation fault.

Code listing 4.20: HTML Code Exploiting table_calculate_column_types

```
<html lang="en">
<head>
    <title>Out-of-bounds write</title>
</head>
<body>
    <svg><tr><><th colspan="4294967295">
</body>
</html>
```

Integer Underflow Causing Out-of-Bounds Read

A similar problem to integer overflows are integer underflows, where a subtraction causes a value to drop below the lowest representable value, and become very large instead. AFL++ found one of these underflows, which caused an out-of-bounds read in `current_node`, shown in Listing 4.21 [118, l. 1166]. Here, `context.current_node` could have been underflowed after a call to `handle_in_row`, where its value is decremented by 4[119, l. 78].

Code listing 4.21: C Code in current_node

```
1166 element_type current_node(hubbub_treebuilder *treebuilder) {
1167     return treebuilder->context.element_stack[treebuilder->context.current_node].
             type;
1168 }
```

The minimized exploit to find this vulnerability was perhaps the most obscure one, and can be seen in Listing 4.22. Even if the SAST tools would have found this vulnerability, it is unlikely that we would be able to manually implement a working exploit, highlighting an advantage of fuzzing. The consequences of this and the previous out-of-bounds read can be leakage of potentially sensitive data [24].

Code listing 4.22: HTML Code Exploiting current_node

```
<html lang="en">
<head>
    <title>Out-of-bounds read</title>
</head>
<body>
    <table><svg><th><html></b><td></th><tr>
</body>
</html>
```

Infinite Loop in textarea_reflow_multiline

This was the only vulnerability found by Domato after disabling JavaScript. It is less serious than the other vulnerabilities in this section, as it only leads to a hang, and can not be developed into further control like an out-of-bounds access potentially could. The vulnerability can be triggered by creating a `textarea`-tag with at least 12 bytes of content and its `rows`-attribute set to 0. When this is the case, netsurf will expect there to be a row, leading to `textarea_reflow_multiline` being called. Parts of the code can be seen in Listing 4.23 [120].

Code listing 4.23: C Code in textarea_reflow_multiline

```
1  static bool textarea_reflow_multiline(struct textarea *ta, const size_t b_start,
       const int b_length, struct rect *r) {
2      [...]
3      do {
4          [...]
5          if (line > scroll_lines && ta->bar_y == NULL) {
6              int h = ta->vis_height - 2 * ta->border_width;
7              if (scrollbar_create(false, h, h, h,
8                      ta, textarea_scrollbar_callback,
9                      &(ta->bar_y)) != NSERROR_OK) {
10                 return false;
11             }
12             [...]
13             restart = true;
14         } else if (line <= scroll_lines && ta->bar_y != NULL) {
15             scrollbar_destroy(ta->bar_y);
16             [...]
17             restart = true;
18         }
19     } while (restart);
```

Initializing the `textarea`-tag with enough content and the`rows`-attribute set to 0, causes a scrollbar to be created, as there is not enough room to display the content in the given number of rows. A side effect of this is `scroll_lines` being decremented from 0, causing an integer underflow to `0xffffffff`. This guarantees that the if-statement on line 14 of Listing 4.23 evaluates to true on the next iteration, causing the reverse effect, destroying the scrollbar. Both when a `scrollbar` is created and jdestroyed, `restart` is set to true, causing the loop to repeat. The eventual result is `scroll_lines` switching back and forth between 0 and `0xffffffff`, causing alternating calls to `scrollbar_create` and `scrollbar_destroy` in an infinite loop. The program remains unresponsive, pinning a CPU core at full utilization, until it is killed with a SIGKILL signal.

The vulnerability appears to be handled in NetSurf with GTK3, meaning that we have only confirmed that it is exploitable when using the framebuffer frontend. The exploit also requires the font size on both the `html`- and `body`-tag to be lowered to “smaller”, as shown in Listing 4.24.

Code listing 4.24: HTML Code Exploiting current_node

```
<html>
<head>
<style>
  html,body {font-size: smaller;}
</style>
</head>
<body>
  <textarea rows="0">aaa-bbb-ccc-</textarea>
</body>
</html>
```

4.3 Disclosure of Discovered Vulnerabilities

Given that we found multiple vulnerabilities in NetSurf, we proceeded with responsible disclosure of the findings, as planned in Section 3.4. We provided the NetSurf security team with the Proof of Concepts presented in Section 4.1.4 and 4.2.3, as well as a *Dockerfile*, which can be seen in appendix A.6. The Dockerfile can be used to create a reproducible and debuggable build of `netsurf`, which could be used on any computer and operating system with support for Docker. We also informed them that we were going to apply for CVEs for the most serious vulnerabilities, and asked if they had any preferences for how we went about this process. NetSurf's security team did not want us to embargo this thesis, allowing us to make it public right away. We are in the process of reporting these vulnerabilities with the proper documentation to MITRE, so they can be assigned official CVE records.

Chapter 5

Discussion

5.1 Sources of Error

Throughout our research, we have answered our research questions and problem statement found in Chapter 1 Introduction. We have tried to keep an objective view on our research, but there are still more factors to consider when reading our results and conclusions.

5.1.1 NetSurf Characteristics

While we have used a wide array of different tools and techniques in our testing, the application, NetSurf, has stayed the same. That means that our process and findings might be biased towards the specifics of NetSurf. For example, software can be written in a myriad of different programming languages, with wildly different characteristics and threat models. While NetSurf is written in C, similar programs could also be written in a more modern systems programming language like Rust. Such a large difference would drastically change both the type of vulnerabilities we are looking for, and what tools we would use. For this reason, our methods and findings are highly applicable to NetSurf, with the potential deviation gradually increasing the further you stray from NetSurf.

Some other aspects of NetSurf also affect the efficiency of our SAST tools, like the abundant use of preprocessor statements. In many areas of NetSurf, these macros make the source code appear very different than the compiled results, making it more difficult for static analysis tools to understand what will happen when the program is running. This problem is described in detail in Section 4.1.5, with examples from NetSurf. This does not affect the fuzzing results at all, since it operates on the compiled binary, with no regard for how the source code looks. For other programs written in C, limiting the use of these preprocessor statements can give the SAST tools a greater advantage with fewer false positives and better detection of actual vulnerabilities.

When considering whether you should use static analysis or something like AFL++ for your project, it is also important to consider the amount of work re-

quired to get started with different tools. NetSurf reads URLs and option flags from the command line, and proceeds to fetch content either from the network or from a file on disk. This makes it a simple target to fuzz, as we only had to modify the code to make it exit automatically when the file is loaded and rendered. For some applications, making the target fuzzable would require a major rewrite. For example, this would be the case if you are trying to generate network traffic, or if the application requires a lot of user interaction. Even for NetSurf, we would probably get a major speed increase by writing a test harness that loaded the file and passed it through the DOM renderer as fast as possible, instead of spending time loading settings, bookmarks, browsing history, and so on for every execution. With NetSurf, it was relatively simple to get a viable fuzzing target, so we did not go the much more difficult route of writing a new target mode for efficient fuzzing. For many applications, you would have to create some sort of wrapper or test harness, increasing the effort required to start fuzzing, potentially by several orders of magnitude.

5.1.2 Manual Analysis

Working with both SAST tools and fuzzing requires some manual analysis. Fuzzing requires us to minimize the input files leading to crashes, and determine the source code causing the vulnerabilities. This process is quite forgiving, as if you remove the wrong part of the input file, the application will cease crashing. It is also easier to confirm that our conclusion is correct after determining the location of a vulnerability.

While fuzzing usually results in clear findings with reproducible crashes, findings from SAST tools may require a lot more work and deeper understanding of the target code to actually prove and reproduce a bug or vulnerability. This results in a larger probability of human error leading to uncaught vulnerabilities. After going through hundreds of false positives and other useless findings, we see the possibility that we become biased towards dismissing subsequent findings from SAST tools, even when we actively try to remain objective. This could potentially have skewed the results further in the direction of SAST tools being ineffective.

5.1.3 Usage of SAST Tools

To provide specific examples on how our usage differs from the typical use cases for SAST tools, we can emphasize that SAST is often implemented into a CI/CD pipeline that runs tests automatically. However, we have manually run each of these tools a single time, in their respective cloud services or locally on our own computers. Because we have been running one-time intensive runs, rather than a continuous process to aid development, we found it important to use several SAST tools instead of choosing one. This choice has shown that there were large and valuable differences between the different tools and their findings, meaning we have found considerably more vulnerabilities and problems with this combined

technique. This can be seen throughout Section 4.1 and in the appendices, listing all the findings from each tool.

As described in Section 3.2, we had some problems with the tools reporting many findings in files outside our scope. This shows that there is a low signal-to-noise ratio with the default configurations of these SAST tools, requiring a lot of tuning and filtering with ignore-files to be useful.

With Coverity, a majority of these exclusions were done automatically, as it does not work by blindly analyzing all the files it can find, but by compiling and building the project to focus on code that is actually in use. This avoids reporting useless vulnerabilities in for example test files, dead code files, and other unnecessary materials.

5.1.4 Usage of Fuzzing Tools

When applying our results for fuzzing other applications, one must remember that NetSurf is a fairly small application, at least for a web browser. As described in Section 5.1.1, NetSurf can natively read its input files from disk, making it very simple to automate the process of running NetSurf on a generated file. A consequence of this was that we could run the entire, unmodified application rather than writing a complicated wrapper. Writing a custom frontend to interact with NetSurf internals would increase the maximum fuzzing rate, and speed up the process, but would require a lot more manual labor to set up.

Our fuzzing with Domato was unconventional in the sense that it is not a fuzzer along the same lines as AFL++. After generating our input files as described in Section 3.3.2, we opened them in NetSurf and inspected the outputs, instead of allowing a fuzzer to automatically investigate crashes and search for new edges. This was extremely efficient at finding JavaScript errors and testing with as much coverage as possible in one go. However, once JavaScript was disabled, we only found one hang and zero crashes, and there was no way for Domato to learn from its findings to increase future efficiency. These results have been very useful to us when testing a web browser, but might not be easily transferable to other types of target applications.

In contrast to Domato, AFL++ was able to adapt based on findings, which might mean that we could have given it even more time than what we described in Section 4.2.2. It is a continuous process, meaning it could go on for very long time, on the scale of several weeks or months. Over time, the potential yield will diminish as the percentage of already covered edges increase, while the cost, in the form of intensive computing power, remains high. Over time, we will also see an increased amount of duplicate findings, as the same bugs and vulnerabilities will be reached in a large number of cases. This also applies to Domato, as it uses a limited set of grammar rules to create a large number of input files, meaning there is a large probability of overlapping test cases.

5.2 Exploring the Problem Statement

In this section, we will explore and approach an answer to each of our research questions from Section 1.5. The combined results of these will lead to the answer of our problem statement, which we will present in Chapter 6 Conclusion.

5.2.1 Performance and Accuracy of SAST Tools

> **RQ1:** How effective are source code analysis tools at detecting and describing security vulnerabilities in NetSurf?

Advantages of working with SAST Tools

Our SAST tools found a total of 41 useful findings, where Coverity reported 63% of them. One of the main advantages we experienced while working with SAST tools, especially Coverity, is that they found a lot of code that did not follow best practices. Pointing out programming mistakes that are not technically errors, and are not caught by the compiler, but are discouraged due to style guides, security concerns, and code quality, is very important to developers. One such example was COV-19 – 26, where potential errors in a preprocessor macro looks to be handled, but this is not actually the case, and the error handling code is unreachable. While this is not currently an exploitable vulnerability, it allows writing unsafe code in the future, leading to new vulnerabilities. Since fuzzing does not have any insight in the code, it cannot detect these issues, which gives SAST tools a clear advantage here. If SAST tools are used in a CI/CD context, they can detect and preemptively avoid some vulnerabilities.

Another advantage is the short runtime, and large number of findings. While they were not all useful, the SAST tools found significant vulnerabilities almost instantly, while fuzzing took considerably longer. The process of setting up and running the SAST tools, as described in Section 3.2, was also quite simple, making the tools and results more accessible.

Disadvantages of working with SAST Tools

The first disadvantage we noticed was that working with SAST tools consisted of more heavy manual work than we expected. This was mostly due to the amount of useless findings that needed to be investigated and removed. The cluttered output we described in Section 4.1.5 contributed to the high workload, and should probably be avoided by grouping together related vulnerabilities in a more structured manner. In terms of efficiency, a higher workload is negative as it took more hours to find the vulnerabilities we found, and a majority of our time was spent investigating false positives and other useless findings.

Another characteristic leading to more false positives and lower efficiency was preprocessor statements in the source code confusing our SAST tools, as mentioned in Section 4.1.5. The Semgrep team explains that it can be quite simple

to create a parser specifically for the C preprocessor statements, however "developing a parser that simultaneously understands both C/C++ code and its preprocessor directives is challenging" [121]. Semgrep mentions that "in real-world scenarios programmers tend to use preprocessor directives in a highly disciplined manner" [121]. This is not the case in NetSurf, as preprocessor statements are a very common occurrence across the code base, making it more difficult for source code analysis tools.

5.2.2 Performance and Accuracy of Fuzzing

> **RQ2:** How effective is fuzzing at uncovering security vulnerabilities in NetSurf, in comparison to source code analysis tools? Are the benefits of fuzzing worth the additional costs?

Advantages of Working with Fuzzing

With fuzzing, we were able to identify 6 unique vulnerabilities that are definitely exploitable, and many other potential problems. Contrary to when we were working with SAST tools, we were surprised by how little manual effort was required to process our results after actually running our fuzzers. Each crash or hang report was simple to minimize, and given our access to source code and debug symbols, we could track them down to specific lines rather quickly. The biggest positive contributor, however, was the absence of false positives, with each finding guaranteed to be real, allowing us to prioritize categorizing findings and digging deeper into their cause. Although we found a relatively low number of exploitable vulnerabilities through fuzzing, each of the findings are real problems with high impact.

Disadvantages of Working with Fuzzing

When compared to the SAST tools we have been using, AFL++ and Domato both required a lot more effort to configure. We needed extensive tooling to get our fuzzers up and running, including a custom build system with Nix, a scalable Docker configuration and a lot of automation scripts.

After the initial setup and tuning, most of the manual labor required for fuzzing was complete. However, we still needed to dedicate a lot of hardware and power to actually running the fuzzers, which could potentially be quite expensive.

Overall Effectiveness of Fuzzing

Despite requiring many hours of work, deep understanding of NetSurf and its build system, and powerful hardware, we have definitely found issues and vulnerabilities that we would not be able to find without fuzzing.

Fuzzing has provided unique, exploitable findings that would be nearly impossible to discover using automated or manual source code analysis, proving it to be an essential tool for finding complex vulnerabilities. Particularly AFL++ has provided us a list of findings entirely without false positives, and a very high average impact.

5.2.3 General Security of the NetSurf Project

> **RQ3:** Are we able to find software vulnerabilities in the chosen parts of the NetSurf web browser project?

We were able to find software vulnerabilities using both SAST tools and fuzzing. Combined, our methods found a total of 47 useful findings, each taking advantage of unique vulnerabilities in netsurf. They ranged from best practice violations to serious vulnerabilities potentially enabling arbitrary code execution.

5.3 Quality of Research

We are able to answer all of our research questions, allowing for a clear conclusion which can be used as a base for future research. To attempt to find useful results that are applicable to a wide array of similar programs, we chose a sufficiently large project to cover many different characteristics of C programs. Another contributing factor to the reliability of our research is our usage of three different SAST tools as well as two vastly different fuzzing tools. When working with fuzzing, we also used multiple different input corpuses in an attempt to exhaust as many paths and edges as possible.

We have found a lot of important technical findings in NetSurf. However, as this type of software projects includes a large number of different styles, choices, traits, and characteristics, we understand that our process and results might not be generally applicable. This means an improvement on our research could have been made by analyzing more programs, preferably with few similarities to NetSurf. With more time and resources, we could also have used more SAST and especially fuzzing tools, to better represent the two categories.

5.4 Future Work

Based on our results and considerations, we believe that our research holds true for NetSurf, but is not necessarily applicable to other software in general. To expand on our results, we think the best direction forward would be to perform similar penetration tests on other software projects. By including other targets, fuzzers, SAST tools, and methods in our testing, we could gather valuable information that would give a better understanding of how these tools and techniques can be used together in general.

If we had more time to work on this project, we might have expanded our horizons by looking at another piece of software. By testing one or more additional applications, preferably with very different characteristics and technologies than NetSurf, we believe our results could have a greater and more reliable impact regarding the flexibility and usability of the tools and techniques in question.

Chapter 6

Conclusion

Our goal has been to use our experiences and findings to answer our research questions to finally shed some light on our problem statement. Any conclusions described here should be considered alongside the reflections and conditions described in Chapter 5 Discussion. The problem statement, as described in Section 1.4 was to investigate if "source code analysis, fuzzing or a combination of the two can be effective at finding real vulnerabilities in software written in a systems programming language". We have determined that each tool can be used effectively in different circumstances to actually find vulnerabilities, when used appropriately.

When considering the different vulnerabilities , bugs, and other problems we have found through our testing, we found it increasingly important to filter out which issues actually had the largest potential impact. When specifically looking for dangerous vulnerabilities , as opposed to inconvenient bugs, we have found fuzzing to be the best tool for the job for an application like NetSurf. We make this conclusion based on the fact that our fuzzing more efficiently and accurately found the most serious vulnerability reported by our SAST tools, in addition to other severe findings that the static analysis failed to uncover. The findings presented by fuzzing were given with more precision than any SAST tools we utilized, and all the findings were on average more severe. SAST tools were more efficient in finding noncompliance with best practices, which help prevent future mistakes and vulnerabilities . This suggests that SAST tools can be used in a CI/CD context for this purpose, continuously helping software developers.

In Section 1.1, we described how we could not find any other academic resources describing our specific problem statement. Through our research into this field, we believe that our methods can be utilized by developers and penetration testers as an effective step towards finding new vulnerabilities . This combination has proved very useful for us, and we hope that it can address an existing gap in academic research as well as being a useful building block in practical applications of these tools.

Finally, we conclude that a combination of these techniques can yield better results than each of them individually. Our research suggests that a high level of security can be achieved by continuously running and following advice from SAST tools, while periodically using fuzzing to catch more serious vulnerabilities too complex for the SAST tools to detect.

Bibliography

[1] R. Stallman, R. Pesch, and S. Shebs et al., “Debugging with GDB,” Free Software Foundation Inc., Tech. Rep., Jan. 2002. [Online]. Available: https://hamblen.ece.gatech.edu/2036/handouts/Fall2012_handouts/gdb-reference.pdf.

[2] Free Software Foundation Inc. “core(5) – Linux manual page.” Accessed May 19th, 2024. (Dec. 22, 2023), [Online]. Available: https://man7.org/linux/man-pages/man5/core.5.html.

[3] Docker. “Docker overview.” Accessed May 8th, 2024. (2024), [Online]. Available: https://docs.docker.com/get-started/overview/.

[4] M. Heuse, A. Fioraldi, D. Maier, D. Zhang, and A. Crump, “Fuzz everything, everywhere, all at once,” Accessed January 10th, 2024, Chaos Communication Congress, Dec. 2023.

[5] Free Software Foundation Inc. “GNU Make.” Accessed May 16th, 2024. (Feb. 26, 2023), [Online]. Available: https://www.gnu.org/software/make/.

[6] G. Lettieri. “Heap exploitation.” Accessed May 3rd, 2024. (Nov. 15, 2023), [Online]. Available: https://lettieri.iet.unipi.it/hacking/heap.pdf.

[7] A. Fioraldi and D. Maier. “The LibAFL Fuzzing Library - Corpus.” Accessed May 7th, 2024. (2024), [Online]. Available: https://aflplus.plus/libafl-book/core_concepts/corpus.html.

[8] A. Tolmach, “Notes on x86-64 programming,” Centre national de la recherche scientifique, Tech. Rep., 2012, Accessed May 14th, 2024. [Online]. Available: https://usr.lmf.cnrs.fr/~jcf/ens/compil/x86-64.pdf.

[9] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper, “The GNU C Library Reference Manual,” Free Software Foundation Inc., Tech. Rep., Aug. 29, 2023. [Online]. Available: https://www.gnu.org/software/libc/manual/pdf/libc.pdf.

[10] The LLVM Admin Team. “The LLVM compiler infrastructure.” Accessed May 20th, 2024. (2024), [Online]. Available: https://llvm.org/.

[11] D. Czarnota. “Pwndbg.” Accessed May 8th, 2024. (Mar. 23, 2024), [Online]. Available: https://github.com/pwndbg/pwndbg.

[12] Z. Riggle. "Pwntools." Accessed May 8th, 2024. (Apr. 24, 2024), [Online]. Available: `https://github.com/Gallopsled/pwntools`.

[13] M. Lemmens, "Stack Canaries Gingerly Sidestepping the Cage," *SANS Cyber Security Blog*, Feb. 4, 2021, Accessed May 8th, 2024. [Online]. Available: `https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/`.

[14] O. Lawlor. "The stack, cs 301 lecture." Accessed May 14th, 2024. (2010), [Online]. Available: `https://www.cs.uaf.edu/2010/fall/cs301/lecture/10_06_the_stack.html`.

[15] MITRE. "CWE Glossary." Accessed April 24th, 2024. (Nov. 16, 2022), [Online]. Available: `https://cwe.mitre.org/documents/glossary/index.html`.

[16] MITRE. "CWE-14: Compiler Removal of Code to Clear Buffers." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/14.html`.

[17] MITRE. "CWE-20: Improper Input Validation." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/20.html`.

[18] MITRE. "CWE-22: Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/22.html`.

[19] MITRE. "CWE-23: Relative Path Traversal." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/23.html`.

[20] MITRE. "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/119.html`.

[21] MITRE. "CWE-120: Buffer Copy without Checking Size of Input (Classic Buffer Overflow)." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/120.html`.

[22] MITRE. "CWE-121: Stack-based Buffer Overflow." Accessed 29th of April 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/121.html`.

[23] MITRE. "CWE-122: Heap-based Buffer Overflow." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/122.html`.

[24] MITRE. "CWE-125: Out-of-bounds Read." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/125.html`.

[25] MITRE. “CWE-131: Incorrect Calculation of Buffer Size.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/131.html.

[26] MITRE. “CWE-170: Improper Null Termination.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/170.html.

[27] MITRE. “CWE-190: Integer Overflow or Wraparound.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/190.html.

[28] MITRE. “CWE-197: Numeric Truncation Error.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/197.html.

[29] MITRE. “CWE-252: Unchecked Return Value.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/252.html.

[30] MITRE. “CWE-328: Use of Weak Hash.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/328.html.

[31] MITRE. “CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/367.html.

[32] MITRE. “CWE-369: Divide By Zero.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/369.html.

[33] MITRE. “CWE-404: Improper Resource Shutdown or Release.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/404.html.

[34] MITRE. “CWE-415: Double Free.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/415.html.

[35] MITRE. “CWE-416: Use After Free.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/416.html.

[36] MITRE. “CWE-457: Use of Unititialized Variable.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/457.html.

[37] MITRE. “CWE-467: Use of sizeof() on a Pointer Type.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/467.html.

[38] MITRE. “CWE-476: NULL Pointer Dereference.” Accessed April 23rd, 2024. (2024), [Online]. Available: https://cwe.mitre.org/data/definitions/476.html.

[39] MITRE. "CWE-484: Omitted Break Statement in Switch." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/484.html`.

[40] MITRE. "CWE-561: Dead Code." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/561.html`.

[41] MITRE. "CWE-563: Assignment to Variable without Use." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/563.html`.

[42] MITRE. "CWE-606: Unchecked Input for Loop Condition." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/606.html`.

[43] MITRE. "CWE-676: Use of Potentially Dangerous Function." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/676.html`.

[44] MITRE. "CWE-775: Missing Release of File Descriptor or Handle after Effective Lifetime." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/775.html`.

[45] MITRE. "CWE-787: Out-of-bounds Write." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/787.html`.

[46] MITRE. "CWE-916: Use of Password Hash With Insufficient Computational Effort." Accessed April 23rd, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/916.html`.

[47] Stack Overflow. "Stack Overflow Developer Survey 2023." Accessed March 8th, 2024. (2023), [Online]. Available: `https://survey.stackoverflow.co/2023/`.

[48] A. K. Lab. "What is cybersecurity? types, threats and cyber safety tips." Accessed May 8th, 2024. (2024), [Online]. Available: `https://www.kaspersky.com/resource-center/definitions/what-is-cyber-security`.

[49] L. B. Furstenau, M. K. Sott, A. J. O. Homrich, L. M. Kipper, A. A. Al Abri, T. F. Cardoso, J. R. López-Robles, and M. J. Cobo, "20 years of scientific evolution of cyber security: A science mapping," in *International conference on industrial engineering and operations management*, IEOM Society International, 2020, pp. 314–325.

[50] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds., Cham: Springer International Publishing, 2017, pp. 26–47, ISBN: 978-3-319-66332-6.

[51] W. Wang, D. Tian, R. Ma, H. Wei, Q. Ying, X. Jia, and L. Zuo, "Shfuzz: A hybrid fuzzing method assisted by static analysis for binary programs," *China Communications*, vol. 18, no. 8, pp. 1–16, 2021. DOI: 10.23919/JCC.2021.08.001.

[52] P. Shields, "Hybrid testing: Combining static analysis and directed fuzzing," Ph.D. dissertation, Massachusetts Institute of Technology, 2023.

[53] R. Scandariato, J. Walden, and W. Joosen, "Static analysis versus penetration testing: A controlled experiment," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 451–460. DOI: 10.1109/ISSRE.2013.6698898.

[54] Greg, David and Biggar, Paul, "Static analysis of dynamic scripting languages," Aug. 17, 2009. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1c25bd0d46db26d43a45008655bf0bd115fd6947.

[55] S. Sakharkar, "Systematic review: Analysis of coding vulnerabilities across languages," *Journal of Information Security*, vol. 14, Oct. 2023. DOI: 10.4236/jis.2023.144019. [Online]. Available: https://www.scirp.org/journal/paperinformation?paperid=128108.

[56] NTNU, *Structure in a empirical thesis*, Accessed February 29th, 2024. [Online]. Available: https://i.ntnu.no/academic-writing/strukture-in-a-empirical-thesis.

[57] Synopsys. "What is Penetration Testing and How Does It Work?" Accessed May 2nd, 2024. (2024), [Online]. Available: https://www.synopsys.com/glossary/what-is-penetration-testing.html.

[58] M. Denis, C. Zena, and T. Hayajneh, "Penetration testing: Concepts, attack methods, and defense strategies," in *2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, IEEE, 2016, pp. 1–6.

[59] P. Engebretson, *The basics of hacking and penetration testing: ethical hacking and penetration testing made easy*. Elsevier, 2013.

[60] H. Poston, "What are black box, grey box, and white box penetration testing?" *Infosec*, Aug. 11, 2020, Accessed February 19th, 2024. [Online]. Available: https://resources.infosecinstitute.com/topics/penetration-testing/what-are-black-box-grey-box-and-white-box-penetration-testing/.

[61] IBM. "What is penetration testing?" Accessed May 2nd, 2024. (Apr. 4, 2024), [Online]. Available: https://www.ibm.com/topics/penetration-testing.

[62] PTES. "The Penetration Testing Execution Standard." Accessed Februrary 25th, 2024. (Aug. 16, 2014), [Online]. Available: http://www.pentest-standard.org/index.php/Main_Page.

[63] PTES. "Intelligence Gathering." Accessed Februrary 25th, 2024. (Oct. 6, 2014), [Online]. Available: http://www.pentest-standard.org/index.php/Intelligence_Gathering.

[64] OWASP Foundation. "Source Code Analysis Tools." Accessed February 19th, 2024. (2024), [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools.

[65] Snyk. "Static Application Security Testing (SAST) Tools." Accessed March 19th, 2024. (2024), [Online]. Available: https://snyk.io/learn/application-security/static-application-security-testing/.

[66] M. Horvath, D. Gardner, M. Bhat, R. Chugh, and A. Zhao, "Magic quadrant for application security testing," Gartner, Inc., Tech. Rep., May 17, 2023. [Online]. Available: https://www.synopsys.com/software-integrity/engage/gartner-mq-auto/gartner-mq-appsec.

[67] M. Milev, J. P. d. A. Sierra, and G. Kamathe. "Shifting security left through collaboration." Accessed April 16th, 2024. (Nov. 22, 2023), [Online]. Available: https://redhat.com/en/blog/shifting-security-left-through-collaboration.

[68] Fortinet. "What Is Shift Left Security?" Accessed April 16th, 2024. (2024), [Online]. Available: https://www.fortinet.com/resources/cyberglossary/shift-left-security.

[69] Synposys. "Static Application Security Testing." Accessed May 8th, 2024. (2024), [Online]. Available: https://www.synopsys.com/glossary/what-is-sast.html.

[70] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, pp. 18–33, 2015, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2015.08.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584915001366.

[71] S. Dechand. "SAST, DAST, IAST and Feedback-Based Fuzzing." Accessed April 17th, 2024. (Apr. 28, 2020), [Online]. Available: https://www.code-intelligence.com/blog/what-is-fast.

[72] Check Point Software Technologies. "How does Static Application Security Testing (SAST) work?" Accessed April 17th, 2024. (2024), [Online]. Available: https://www.checkpoint.com/cyber-hub/cloud-security/what-is-static-application-security-testing-sast/.

[73] J. Fell, "A review of fuzzing tools and methods," *PenTest Magazine*, 2017. [Online]. Available: https://wcventure.github.io/FuzzingPaper/Paper/2017_review.pdf.

[74] M. Zalewski. "American fuzzy lop." Accessed May 21st, 2024. (2020), [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[75] AFLplusplus. "AFL++ Overview." Accessed May 7th 2024. (2024), [Online]. Available: https://github.com/AFLplusplus/AFLplusplus.

[76] AFLplusplus. "Fuzzing with afl++." Accessed May 7th, 2024. (2024), [Online]. Available: https://aflplus.plus/docs/fuzzing_in_depth/.

[77] A. Fioraldi and D. Maier. "The LibAFL Fuzzing Library - Mutator." Accessed May 7th, 2024. (2024), [Online]. Available: https://aflplus.plus/libafl-book/core_concepts/mutator.html.

[78] MITRE and CVE Working Groups. "CVE Record Lifecycle." Accessed April 25th, 2024. (2024), [Online]. Available: https://www.cve.org/About/Process.

[79] National Institute of Standards and Technology (NIST). "Vulnerabilities." Accessed April 26th, 2024. (Aug. 3, 2023), [Online]. Available: https://nvd.nist.gov/vuln.

[80] MITRE. "About CWE." Accessesed April 24th, 2024. (Mar. 22, 2024), [Online]. Available: https://cwe.mitre.org/about/index.html.

[81] CWE Community. "Frequently Asked Questions (FAQ)." Accessed April 24th, 2024. (Mar. 22, 2024), [Online]. Available: https://cwe.mitre.org/about/faq.html.

[82] ISO/IEC JTC1/SC22/WG14, "ISO/IEC 9899:2023," ISO/IEC, Tech. Rep., Apr. 1, 2023, Accessed April 24th, 2024. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3088.pdf.

[83] E. Gupta. "Memory Layout in C." Accessed April 24th, 2024. (Jan. 16, 2024), [Online]. Available: https://www.scaler.com/topics/c/memory-layout-in-c/.

[84] BugTraq, r00t, and Underground.Org, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, 49 Nov. 8, 1996, Accessed April 24th, 2024. [Online]. Available: http://phrack.org/issues/49/14.html.

[85] krloer. "Veal and Car's first baby (heap)." Accessed April 25th, 2024. (2023), [Online]. Available: https://ctf.krloer.com/writeups/ept/baby_heap/.

[86] P. S. Foundation. "patchelf." Accessed April 25th, 2024. (Feb. 26, 2023), [Online]. Available: https://pypi.org/project/patchelf/.

[87] M. A. Butt, Z. Ajmal, Z. I. Khan, M. Idrees, and Y. Javed, "An in-depth survey of bypassing buffer overflow mitigation techniques," *Applied Sciences*, vol. 12, Jul. 2022. DOI: 10.3390/app12136702.

[88] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security - TISSEC*, vol. 15, pp. 1–34, Mar. 2012. DOI: 10.1145/2133375.2133377.

[89] Arm Limited, "Learn the architecture - providing protection for complex software," pp. 10–18, 1 Dec. 11, 2023, Accessed April 26th, 2024. [Online]. Available: `https://developer.arm.com/documentation/102433/0200/Return-oriented-programming`.

[90] Arm Limited, "Learn the architecture - providing protection for complex software," pp. 19–22, 1 Dec. 11, 2023, Accessed April 26th, 2024. [Online]. Available: `https://developer.arm.com/documentation/102433/0200/Jump-oriented-programming`.

[91] MITRE. "CWE-134: Use of Externally-Controlled Format String." Accessed May 20th, 2024. (2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/134.html`.

[92] Wikipedia contributors, *Sigreturn-oriented programming — Wikipedia, the free encyclopedia*, Accessed April 26th, 2024, 2023. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Sigreturn-oriented_programming&oldid=1167065775`.

[93] Snyk. "Snyk Code." Accessed March 19th, 2024. (2024), [Online]. Available: `https://snyk.io/product/snyk-code/`.

[94] Gartner and Reviewers. "Snyk Code Reviews." Accessed April 3rd, 2024. (2024), [Online]. Available: `https://www.gartner.com/reviews/market/application-security-testing/vendor/snyk/product/snyk-code`.

[95] Semgrep. "Docs home | Semgrep." Accessed April 9th, 2024. (2024), [Online]. Available: `https://semgrep.dev/docs/`.

[96] Y. Padioleau. "Semgrep: A static analysis journey." Accessed May 20th, 2024. (Nov. 9, 2024), [Online]. Available: `https://semgrep.dev/blog/2021/semgrep-a-static-analysis-journey`.

[97] Synopsys. "Frequently Asked Questions (FAQ) - What is Coverity Scan?" Accessed April 5th, 2024. (2024), [Online]. Available: `https://scan.coverity.com/faq#what-is-coverity-scan`.

[98] AFL++. "Tips for parallel fuzzing." Accessed May 20th, 2024. (2015), [Online]. Available: `https://aflplus.plus/docs/parallel_fuzzing/`.

[99] OWASP Foundation. "Vulnerability Disclosure Cheat Sheet." Accessed May 16th, 2024. (2024), [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html`.

[100] MITRE. "CWE CATEGORY: 7PK - Code Quality." Accessed April 23rd, 2024. (Feb. 29, 2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/398.html`.

[101] MITRE. "CWE CATEGORY: Expression Issues." Accessed April 23rd, 2024. (Feb. 29, 2024), [Online]. Available: `https://cwe.mitre.org/data/definitions/569.html`.

[102] NetSurf. "idna.c." Accessed May 4th, 2024. (Jun. 17, 2023), [Online]. Available: https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/utils/idna.c.

[103] NetSurf. "parse.c." Accessed May 4th, 2024. (Jun. 17, 2023), [Online]. Available: https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/utils/nsurl/parse.c.

[104] NetSurf. "selection.c." Accessed May 3rd, 2024. (Sep. 21, 2019), [Online]. Available: https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/frontends/gtk/selection.c.

[105] NetSurf. "textarea.c." Accessed May 3rd, 2024. (Jan. 23, 2022), [Online]. Available: https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/desktop/textarea.c.

[106] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for k(h)eaps: Understanding and improving linux kernel exploit reliability," *Proceedings of the 31st USENIX Security Symposium*, Aug. 12, 2022. [Online]. Available: https://www.usenix.org/system/files/sec22-zeng.pdf.

[107] NetSurf. "hash.c." Accessed May 3rd, 2024. (Aug. 28, 2022), [Online]. Available: https://github.com/netsurf-browser/libcss/blob/b88595eb72302cf40e13b78e2d2917c7e98b66c4/src/select/hash.c.

[108] NetSurf. "bloom.h." Accessed May 3rd, 2024. (Mar. 16, 2022), [Online]. Available: https://github.com/netsurf-browser/libcss/blob/b88595eb72302cf40e13b78e2d2917c7e98b66c4/src/select/bloom.h.

[109] NetSurf. "html.c." Accessed May 4th, 2024. (Jun. 17, 2023), [Online]. Available: https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/content/handlers/html/html.c.

[110] NetSurf. "utf8.c." Accessed May 4th, 2024. (Jun. 17, 2023), [Online]. Available: https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/utils/utf8.c.

[111] Free Software Foundation Inc. "Top (The C Preprocessor): Ifdef." Accessed May 16th, 2024. (2024), [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/Ifdef.html.

[112] NetSurf. "duktape.c." Accessed May 16th, 2024. (May 29, 2022), [Online]. Available: https://raw.githubusercontent.com/netsurf-browser/netsurf/466361cb148e301213b8e8aa3b488bb4242827f2/content/handlers/javascript/duktape/duktape.c.

[113] NetSurf. "hlcache.c." Accessed May 6th, 2024. (Jun. 17, 2023), [Online]. Available: https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/content/hlcache.c.

[114] NetSurf. "ring.h." Accessed May 6th, 2024. (Nov. 10, 2019), [Online]. Available: `https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/utils/ring.h`.

[115] NetSurf. "User information." Accessed May 5th, 2024. (), [Online]. Available: `https://www.netsurf-browser.org/documentation/info.html`.

[116] NetSurf. "box_normalise.c." Accessed May 7th, 2024. (Oct. 29, 2022), [Online]. Available: `https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/content/handlers/html/box_normalise.c`.

[117] NetSurf. "table.c." Accessed May 7th, 2024. (Oct. 29, 2022), [Online]. Available: `https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/content/handlers/html/table.c`.

[118] NetSurf. "treebuilder.c." Accessed May 7th, 2024. (May 27, 2021), [Online]. Available: `https://github.com/netsurf-browser/libhubbub/blob/873ed6e236f7669afd3ef44259c34addc6dc95b6/src/treebuilder/treebuilder.c`.

[119] NetSurf. "treebuilder.c." Accessed May 7th, 2024. (Jul. 26, 2011), [Online]. Available: `https://github.com/netsurf-browser/libhubbub/blob/873ed6e236f7669afd3ef44259c34addc6dc95b6/src/treebuilder/in_row.c`.

[120] NetSurf. "textarea.c." Accessed May 7th, 2024. (Jan. 23, 2022), [Online]. Available: `https://github.com/netsurf-browser/netsurf/blob/466361cb148e301213b8e8aa3b488bb4242827f2/desktop/textarea.c`.

[121] Semgrep. "Semgrep code brings modern static analysis to c/c++." Accessed May 7th, 2024. (Feb. 27, 2024), [Online]. Available: `https://semgrep.dev/blog/2024/modernizing-static-analysis-for-c`.

Appendix A

Additional Material

A.1 Initial Findings - Snyk

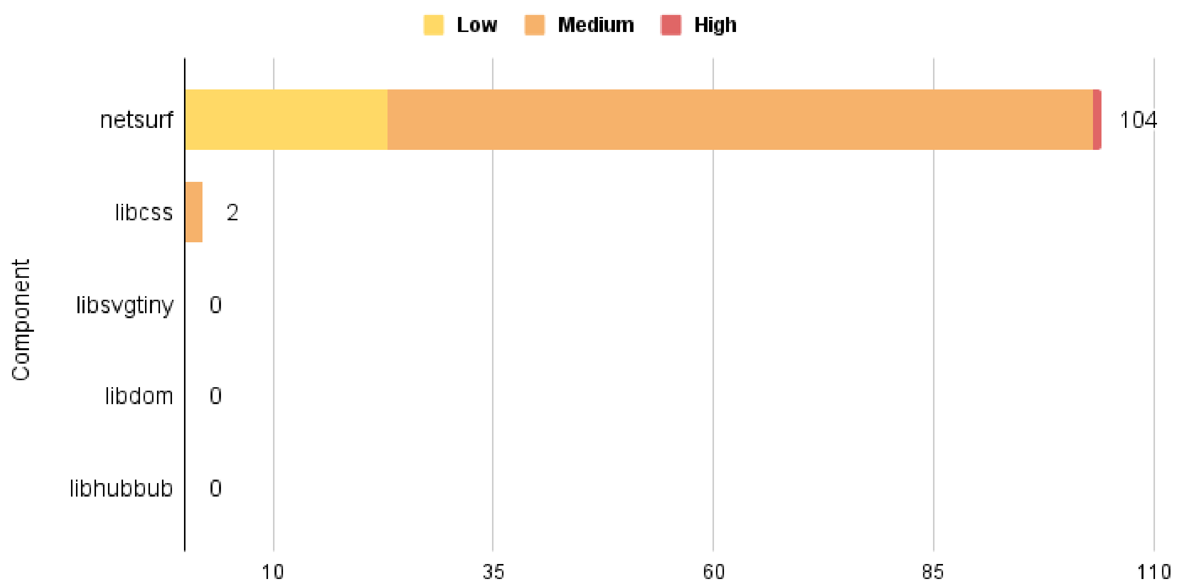


Figure A.1: Potential Vulnerabilities per Component - Snyk

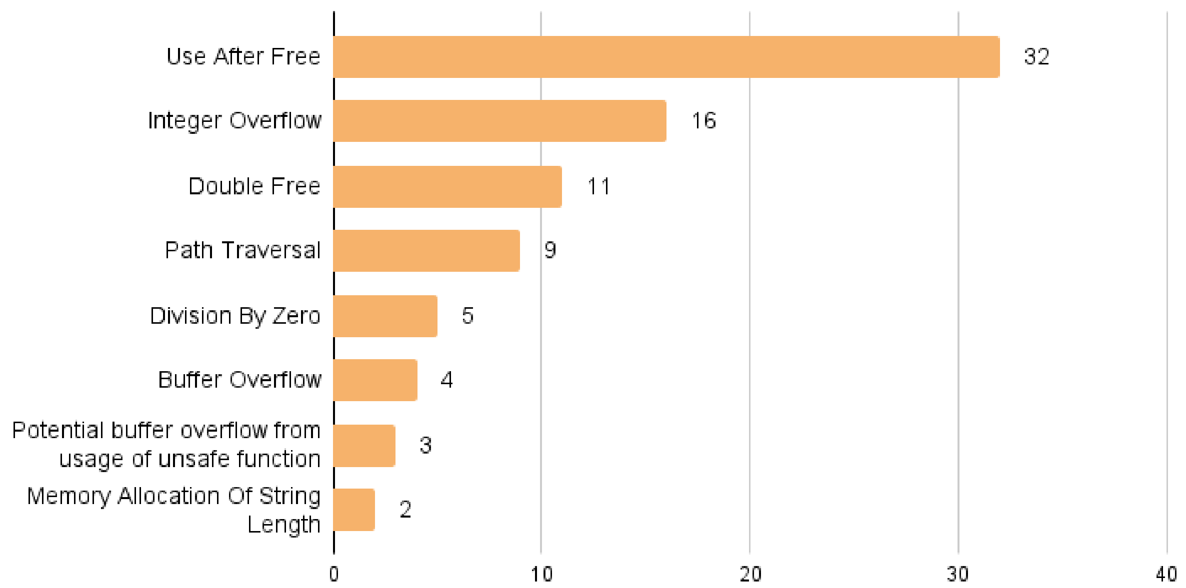


Figure A.2: Potential Medium Vulnerabilities - Snyk

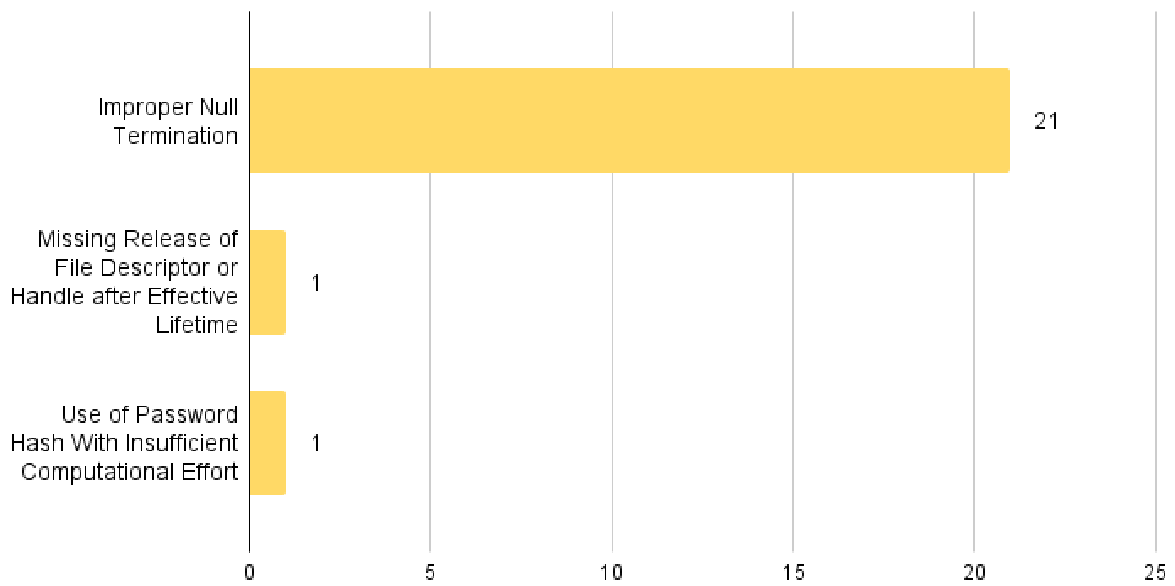


Figure A.3: Potential Low Vulnerabilities - Snyk

CWE	Vulnerabiltiy Type	Severity	File:Line
CWE-23	Path Traversal	Medium	netsurf/tools/convert_font.c:1020
CWE-23	Path Traversal	Medium	netsurf/tools/xxd.c:80
CWE-23	Path Traversal	Medium	netsurf/tools/convert_image.c:230
CWE-23	Path Traversal	Medium	netsurf/tools/convert_font.c:316
CWE-23	Path Traversal	Medium	netsurf/tools/xxd.c:93
CWE-23	Path Traversal	Medium	netsurf/tools/convert_image.c:253
CWE-23	Path Traversal	Medium	libcss/src/parse/properties/css_property_parser_gen.c:538
CWE-23	Path Traversal	Medium	netsurf/tools/split-messages.c:508
CWE-23	Path Traversal	Medium	netsurf/tools/split-messages.c:515
CWE-122	Buffer Overflow	Medium	netsurf/frontends/framebuffer/findfile.c:96
CWE-122	Buffer Overflow	Medium	netsurf/frontends/framebuffer/findfile.c:103
CWE-122	Buffer Overflow	Medium	netsurf/frontends/windows/findfile.c:103
CWE-122	Buffer Overflow	Medium	netsurf/frontends/windows/findfile.c:136
CWE-122	Potential buffer overflow from usage of unsafe function	Medium	netsurf/frontends/windows/findfile.c:121
CWE-122	Potential buffer overflow from usage of unsafe function	Medium	netsurf/frontends/windows/findfile.c:140
CWE-122	Potential buffer overflow from usage of unsafe function	Medium	netsurf/frontends/gtk/download.c:388
CWE-125	Potential Negative Number Used as Index	High	netsurf/utils/ssl_certs.c:249
CWE-170	Memory Allocation Of String Length	Medium	netsurf/frontends/windows/clipboard.c:63
CWE-170	Memory Allocation Of String Length	Medium	netsurf/frontends/gtk/selection.c:55
CWE-170	Improper Null Termination	Low	netsurf/desktop/browser_history.c:74
CWE-170	Improper Null Termination	Low	netsurf/utils/idna.c:832
CWE-170	Improper Null Termination	Low	netsurf/desktop/browser_history.c:129
CWE-170	Improper Null Termination	Low	netsurf/frontends/windows/file.c:164
CWE-170	Improper Null Termination	Low	netsurf/frontends/windows/file.c:167
CWE-170	Improper Null Termination	Low	netsurf/desktop/page-info.c:310
CWE-170	Improper Null Termination	Low	netsurf/desktop/page-info.c:342
CWE-170	Improper Null Termination	Low	netsurf/desktop/page-info.c:354
CWE-170	Improper Null Termination	Low	netsurf/desktop/page-info.c:359
CWE-170	Improper Null Termination	Low	netsurf/contents/fs_backing_store.c:1162
CWE-170	Improper Null Termination	Low	netsurf/utils/nsoption.c:271
CWE-170	Improper Null Termination	Low	netsurf/utils/nsoption.c:299
CWE-170	Improper Null Termination	Low	netsurf/utils/nsoption.c:305
CWE-170	Improper Null Termination	Low	netsurf/utils/nsoption.c:311
CWE-170	Improper Null Termination	Low	netsurf/utils/nsoption.c:318
CWE-170	Improper Null Termination	Low	netsurf/utils/nsoption.c:334
CWE-170	Improper Null Termination	Low	netsurf/utils/nsoption.c:311
CWE-170	Improper Null Termination	Low	netsurf/desktop/browser_window.c:1188
CWE-170	Improper Null Termination	Low	netsurf/desktop/textarea.c:2012
CWE-170	Improper Null Termination	Low	netsurf/desktop/searchweb.c:270
CWE-170	Improper Null Termination	Low	netsurf/utils/nsurl/parse.c:923
CWE-190	Integer Overflow	Medium	libcss/src/parse/properties/css_property_parser_gen.c:47
CWE-190	Integer Overflow	Medium	netsurf/frontends/framebuffer/findfile.c:96
CWE-190	Integer Overflow	Medium	netsurf/frontends/framebuffer/findfile.c:103
CWE-190	Integer Overflow	Medium	netsurf/contents/fs_backing_store.c:1162
CWE-190	Integer Overflow	Medium	netsurf/contents/fs_backing_store.c:1824
CWE-190	Integer Overflow	Medium	netsurf/contents/urldb.c:2274
CWE-190	Integer Overflow	Medium	netsurf/contents/urldb.c:2278
CWE-190	Integer Overflow	Medium	netsurf/contents/urldb.c:2291
CWE-190	Integer Overflow	Medium	netsurf/contents/urldb.c:2296
CWE-190	Integer Overflow	Medium	netsurf/contents/urldb.c:4420
CWE-190	Integer Overflow	Medium	netsurf/frontends/gtk/fetch.c:157
CWE-190	Integer Overflow	Medium	netsurf/frontends/gtk/fetch.c:162
CWE-190	Integer Overflow	Medium	netsurf/contents/fs_backing_store.c:1825
CWE-190	Integer Overflow	Medium	netsurf/contents/urldb.c:2304
CWE-190	Integer Overflow	Medium	netsurf/frontends/framebuffer/gui.c:2259
CWE-190	Integer Overflow	Medium	netsurf/frontends/framebuffer/gui.c:1033
CWE-369	Division By Zero	Medium	netsurf/frontends/framebuffer/fbtk/osk.c:159
CWE-369	Division By Zero	Medium	netsurf/frontends/framebuffer/fbtk/osk.c:165
CWE-369	Division By Zero	Medium	netsurf/frontends/framebuffer/fbtk/osk.c:166
CWE-369	Division By Zero	Medium	netsurf/frontends/framebuffer/fbtk/osk.c:167
CWE-369	Division By Zero	Medium	netsurf/frontends/framebuffer/fbtk/osk.c:168

CWE	Vulnerabiltiy Type	Severity	File:Line
CWE-415	Double Free	Medium	netsurf/content/handlers/html/form.c:1224
CWE-415	Double Free	Medium	netsurf/content/handlers/html/form.c:1236
CWE-415	Double Free	Medium	netsurf/content/handlers/html/form.c:1244
CWE-415	Double Free	Medium	netsurf/content/handlers/html/form.c:1235
CWE-415	Double Free	Medium	netsurf/content/handlers/html/form.c:1235
CWE-415	Double Free	Medium	netsurf/utils/idna.c:720
CWE-415	Double Free	Medium	netsurf/utils/idna.c:727
CWE-415	Double Free	Medium	netsurf/utils/idna.c:800
CWE-415	Double Free	Medium	netsurf/utils/idna.c:732
CWE-415	Double Free	Medium	netsurf/utils/idna.c:806
CWE-415	Double Free	Medium	netsurf/utils/file.c:374
CWE-416	Use After Free	Medium	netsurf/content/hlcache.c:762
CWE-416	Use After Free	Medium	netsurf/content/hlcache.c:813
CWE-416	Use After Free	Medium	netsurf/content/hlcache.c:764
CWE-416	Use After Free	Medium	netsurf/content/hlcache.c:815
CWE-416	Use After Free	Medium	netsurf/content/handlers/html/form.c:1232
CWE-416	Use After Free	Medium	netsurf/content/handlers/html/form.c:1232
CWE-416	Use After Free	Medium	netsurf/content/handlers/html/form.c:1242
CWE-416	Use After Free	Medium	netsurf/content/handlers/html/form.c:1242
CWE-416	Use After Free	Medium	netsurf/content/handlers/html/form.c:2242
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:106
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:263
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:267
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:274
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:277
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:287
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:292
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:302
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:306
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:420
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:420
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:427
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:467
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:477
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:438
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:445
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:455
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:463
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:451
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:386
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:805
CWE-416	Use After Free	Medium	netsurf/utils/idna.c:731
CWE-416	Use After Free	Medium	netsurf/utils/talloc.c:808
CWE-775	Missing Release of File Descriptor or Handle after Effective Lifetime	Low	netsurf/frontends/windows/main.c:151
CWE-916	Use of Password Hash With Insufficient Computational Effort	Low	netsurf/content/fetchers/about/certificate.c:1052

A.2 Initial Findings - Semgrep

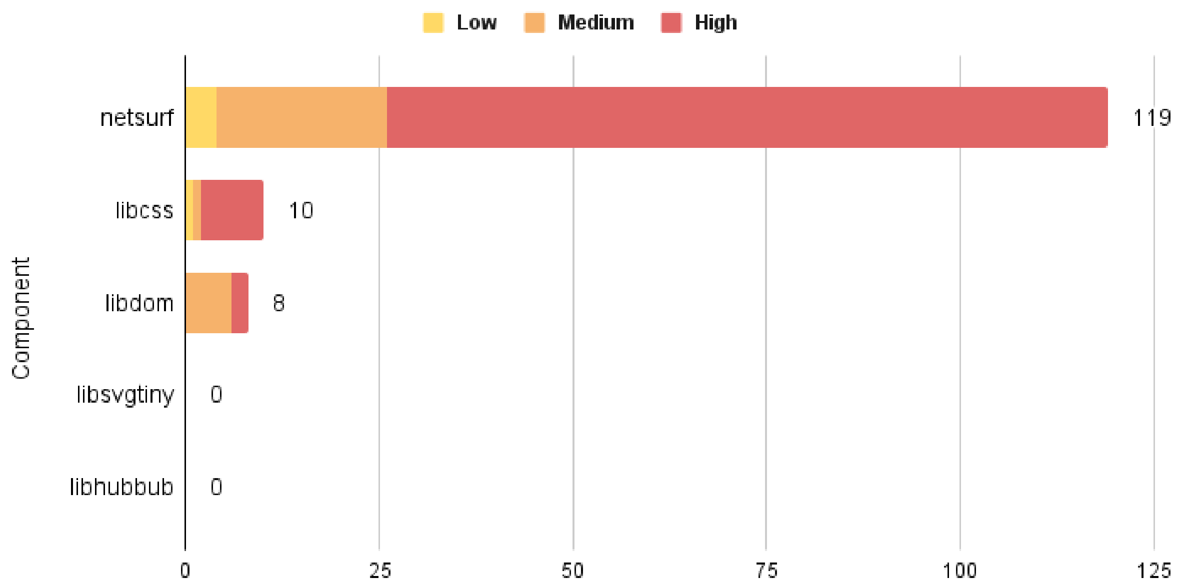


Figure A.4: Potential Vulnerabilities per Component - Semgrep

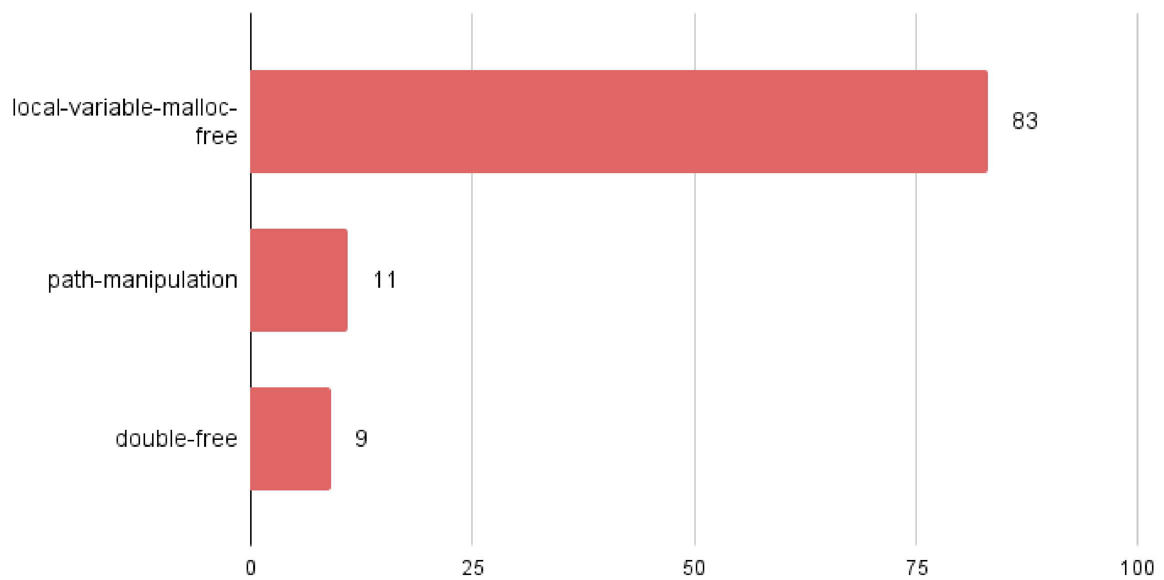


Figure A.5: Potential High Vulnerabilities - Semgrep

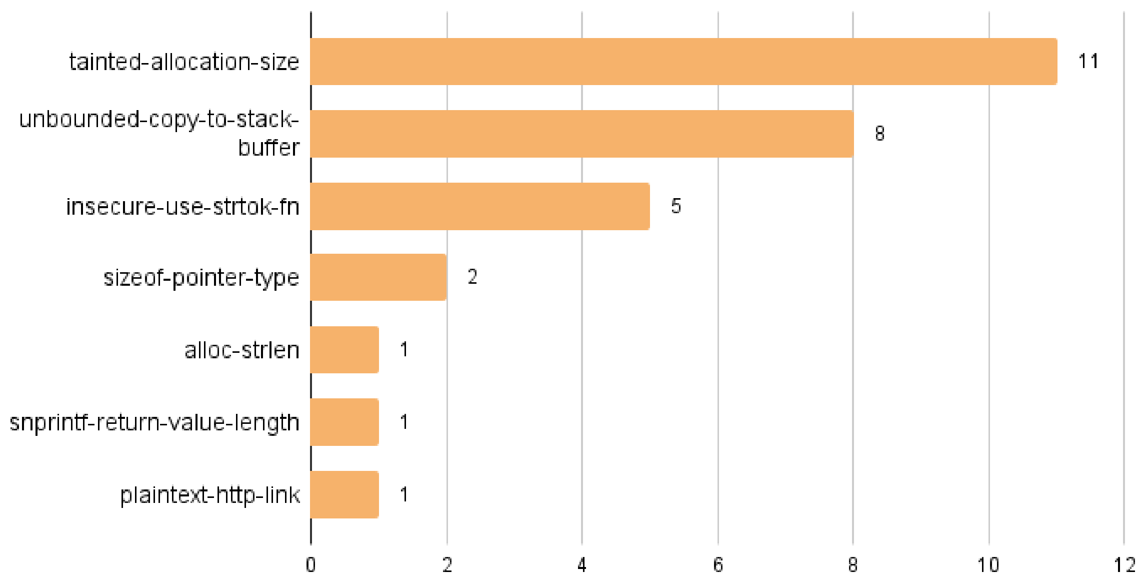


Figure A.6: Potential Medium Vulnerabilities - Semgrep

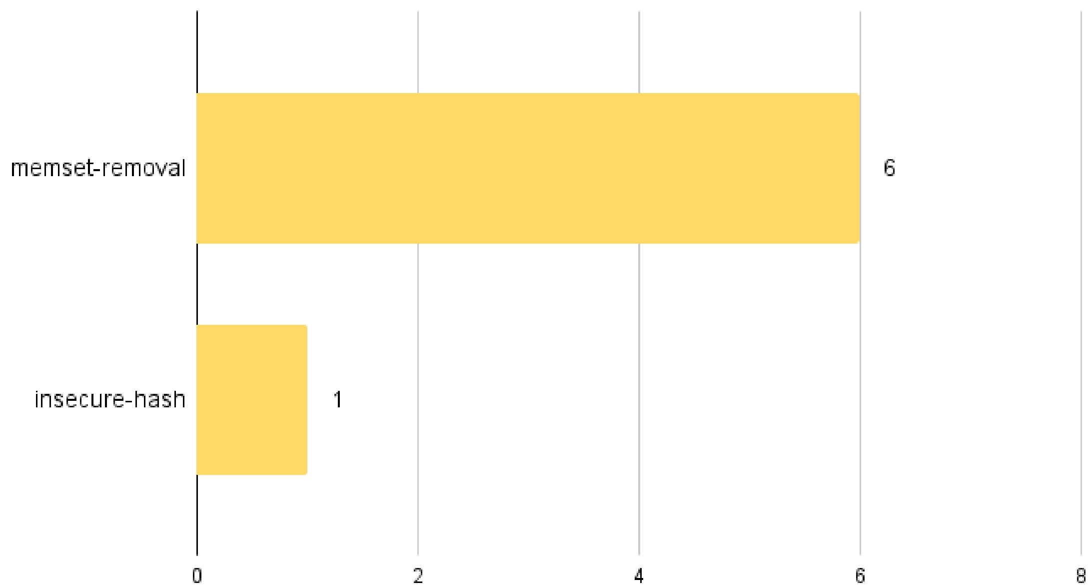


Figure A.7: Potential Low Vulnerabilities - Semgrep

CWE	Vulnerability Type	Severity	File:Line
CWE-14	memset-removal	Low	libcss/src/select/bloom.h:183
CWE-14	memset-removal	Low	netsurf/tools/convert_font.c:803
CWE-14	memset-removal	Low	netsurf/utils/talloc.c:1095
CWE-14	memset-removal	Low	netsurf/utils/utf8.c:452
CWE-22	path-manipulation	High	libcss/src/parse/properties/css_property_parser_gen.c:538
CWE-22	path-manipulation	High	netsurf/frontends/gtk/gui.c:1009
CWE-22	path-manipulation	High	netsurf/frontends/gtk/viewdata.c:656
CWE-22	path-manipulation	High	netsurf/frontends/gtk/viewdata.c:722
CWE-22	path-manipulation	High	netsurf/tools/convert_font.c:316
CWE-22	path-manipulation	High	netsurf/tools/convert_font.c:1020
CWE-22	path-manipulation	High	netsurf/tools/convert_image.c:230
CWE-22	path-manipulation	High	netsurf/tools/convert_image.c:253
CWE-22	path-manipulation	High	netsurf/tools/xxd.c:80
CWE-22	path-manipulation	High	netsurf/tools/xxd.c:93
CWE-22	path-manipulation	High	netsurf/utils/log.c:219
CWE-120	unbounded-copy-to-stack-buffer	Medium	libdom/bindings/xml/libxml_xmlparser.c:718
CWE-120	unbounded-copy-to-stack-buffer	Medium	libdom/bindings/xml/libxml_xmlparser.c:810
CWE-120	unbounded-copy-to-stack-buffer	Medium	libdom/src/utils/hashtable.c:413
CWE-120	unbounded-copy-to-stack-buffer	Medium	libdom/src/utils/hashtable.c:423
CWE-120	unbounded-copy-to-stack-buffer	Medium	libdom/src/utils/hashtable.c:414
CWE-120	unbounded-copy-to-stack-buffer	Medium	libdom/src/utils/hashtable.c:424
CWE-120	unbounded-copy-to-stack-buffer	Medium	netsurf/frontends/gtk/download.c:388
CWE-120	unbounded-copy-to-stack-buffer	Medium	netsurf/frontends/windows/download.c:107
CWE-125	tainted-allocation-size	Medium	libcss/src/parse/properties/css_property_parser_gen.c:47
CWE-125	tainted-allocation-size	Medium	netsurf/content/urldb.c:3084
CWE-125	tainted-allocation-size	Medium	netsurf/frontends/gtk/gui.c:1016
CWE-125	tainted-allocation-size	Medium	netsurf/frontends/gtk/viewdata.c:606
CWE-125	tainted-allocation-size	Medium	netsurf/frontends/gtk/viewdata.c:614
CWE-125	tainted-allocation-size	Medium	netsurf/frontends/gtk/viewdata.c:651
CWE-125	tainted-allocation-size	Medium	netsurf/frontends/gtk/viewdata.c:717
CWE-125	tainted-allocation-size	Medium	netsurf/tools/xxd.c:38
CWE-125	tainted-allocation-size	Medium	netsurf/utils/filepath.c:254
CWE-125	tainted-allocation-size	Medium	netsurf/utils/filepath.c:283
CWE-125	tainted-allocation-size	Medium	netsurf/utils/hashtable.c:321
CWE-131	alloc-strlen	Medium	netsurf/frontends/gtk/selection.c:55
CWE-319	plaintext-http-link	Medium	netsurf/resources/nl/welcome.html:46
CWE-328	insecure-hash	Low	netsurf/content/fetchers/about/certificate.c:1052
CWE-415	double-free	High	libcss/src/stylesheet.c:772
CWE-415	double-free	High	netsurf/content/content_factory.c:67
CWE-415	double-free	High	netsurf/content/handlers/html/imagemap.c:191
CWE-415	double-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:801
CWE-415	double-free	High	netsurf/content/urldb.c:2780
CWE-415	double-free	High	netsurf/content/urldb.c:2786
CWE-415	double-free	High	netsurf/frontends/framebuffer/schedule.c:83
CWE-415	double-free	High	netsurf/frontends/gtk/viewdata.c:957
CWE-415	double-free	High	netsurf/frontends/windows/plot.c:253

CWE	Vulnerability Type	Severity	File:Line
CWE-416	local-variable-malloc-free	High	libcss/src/parse/language.c:1908
CWE-416	local-variable-malloc-free	High	libcss/src/stylesheet.c:760
CWE-416	local-variable-malloc-free	High	libcss/src/stylesheet.c:764
CWE-416	local-variable-malloc-free	High	libcss/src/stylesheet.c:765
CWE-416	local-variable-malloc-free	High	libcss/src/stylesheet.c:769
CWE-416	local-variable-malloc-free	High	libcss/src/stylesheet.c:771
CWE-416	local-variable-malloc-free	High	libdom/bindings/xml/libxml_xmlparser.c:226
CWE-416	local-variable-malloc-free	High	libdom/bindings/xml/libxml_xmlparser.c:226
CWE-416	local-variable-malloc-free	High	netsurf/content/content_factory.c:62
CWE-416	local-variable-malloc-free	High	netsurf/content/content_factory.c:63
CWE-416	local-variable-malloc-free	High	netsurf/content/content_factory.c:65
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:785
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:787
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:788
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:791
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:797
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:832
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:834
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:835
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:835
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:836
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:1673
CWE-416	local-variable-malloc-free	High	netsurf/content/handlers/javascript/duktape/dukky.c:1681
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2719
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2720
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2723
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2724
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2727
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2728
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2730
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2732
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2734
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2779
CWE-416	local-variable-malloc-free	High	netsurf/content/urldb.c:2785
CWE-416	local-variable-malloc-free	High	netsurf/desktop/download.c:211
CWE-416	local-variable-malloc-free	High	netsurf/frontends/framebuffer/schedule.c:76
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/bitmap.c:251
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/bitmap.c:257
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/bitmap.c:257
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/bitmap.c:260
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:130
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:136
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:136
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:153
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:163
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:164

CWE	Vulnerability Type	Severity	File:Line
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:166
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:166
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:167
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:175
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:175
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:181
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:181
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:190
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:190
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:194
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:195
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:199
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:199
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:206
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:211
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:212
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:213
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:214
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:215
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:246
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:246
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:247
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:251
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:252
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:303
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:306
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:307
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:315
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:316
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:325
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:325
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:326
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:327
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:340
CWE-416	local-variable-malloc-free	High	netsurf/frontends/windows/plot.c:955
CWE-416	local-variable-malloc-free	High	netsurf/utils/talloc.c:808
CWE-416	local-variable-malloc-free	High	netsurf/utils/talloc.c:811
CWE-467	sizeof-pointer-type	Medium	netsurf/content/handlers/html/redraw.c:1797
CWE-467	sizeof-pointer-type	Medium	netsurf/content/handlers/html/redraw.c:1808
CWE-676	insecure-use-strtok-fn	Medium	netsurf/content/handlers/html/imagemap.c:396
CWE-676	insecure-use-strtok-fn	Medium	netsurf/content/handlers/html/imagemap.c:419
CWE-676	insecure-use-strtok-fn	Medium	netsurf/content/handlers/html/imagemap.c:438
CWE-676	insecure-use-strtok-fn	Medium	netsurf/content/handlers/html/imagemap.c:448
CWE-676	insecure-use-strtok-fn	Medium	netsurf/content/handlers/html/imagemap.c:472
CWE-787	snprintf-return-value-length	Medium	netsurf/frontends/gtk/resources.c:386

A.3 Initial Findings - Coverity

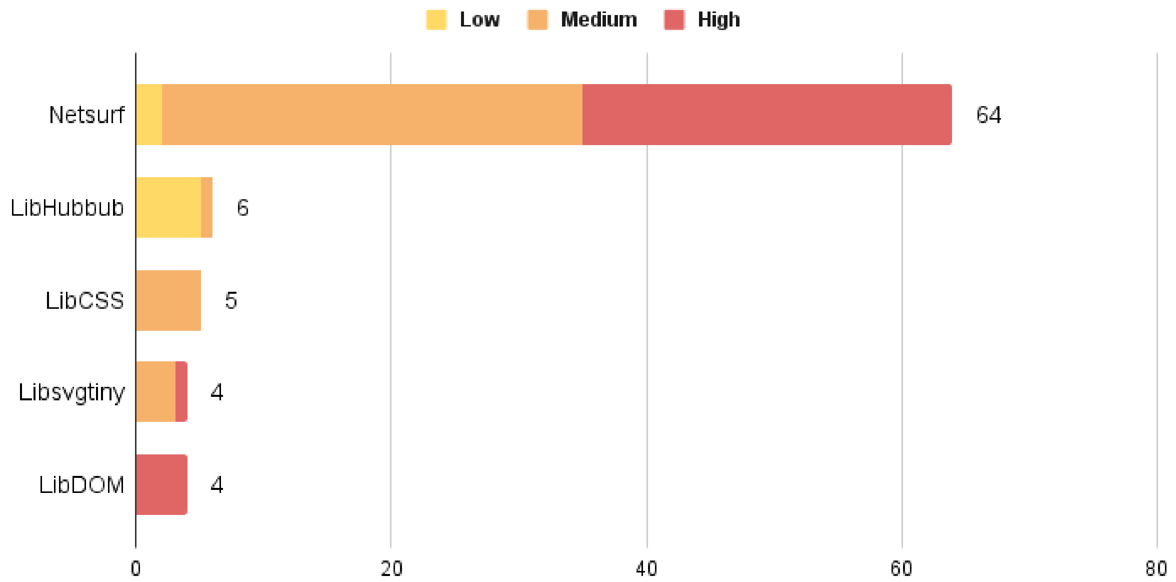


Figure A.8: Potential Vulnerabilities per Component - Coverity

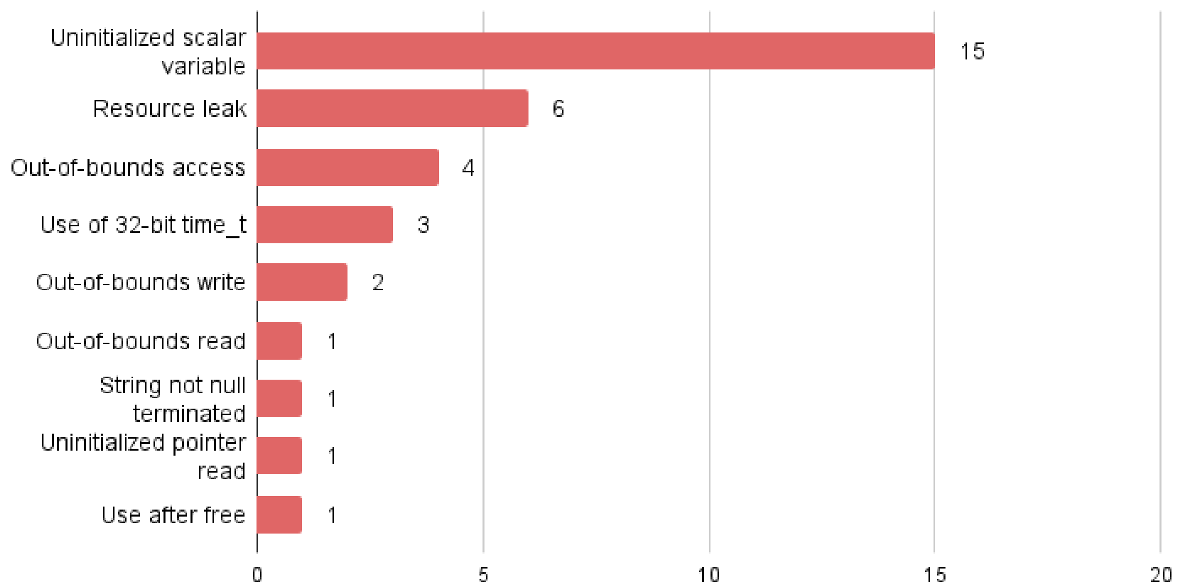


Figure A.9: Potential High Vulnerabilities - Coverity

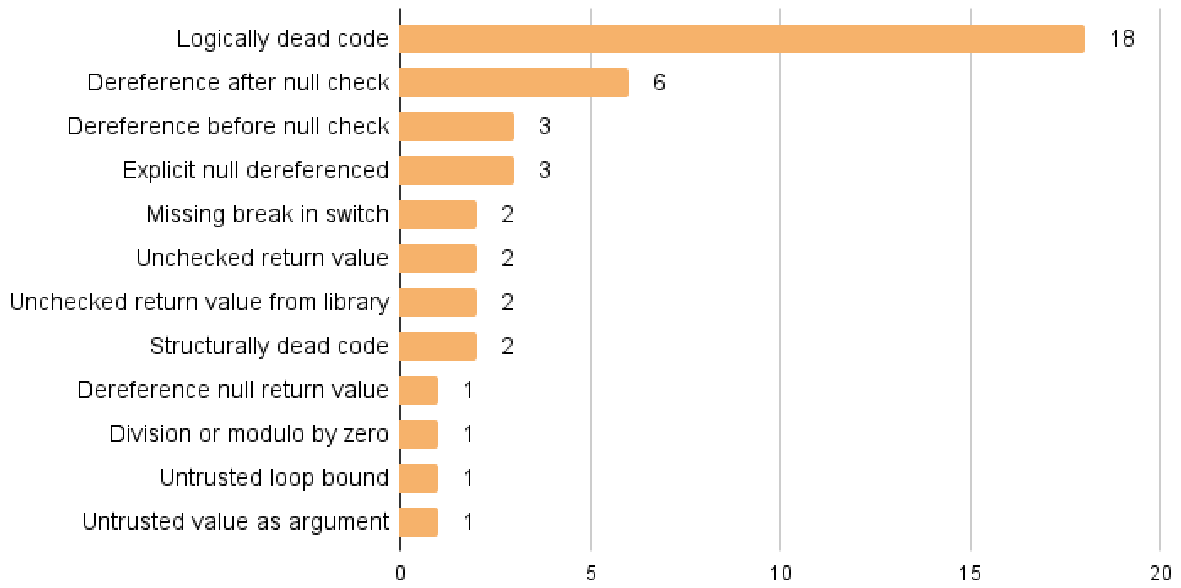


Figure A.10: Potential Medium Vulnerabilities - Coverity

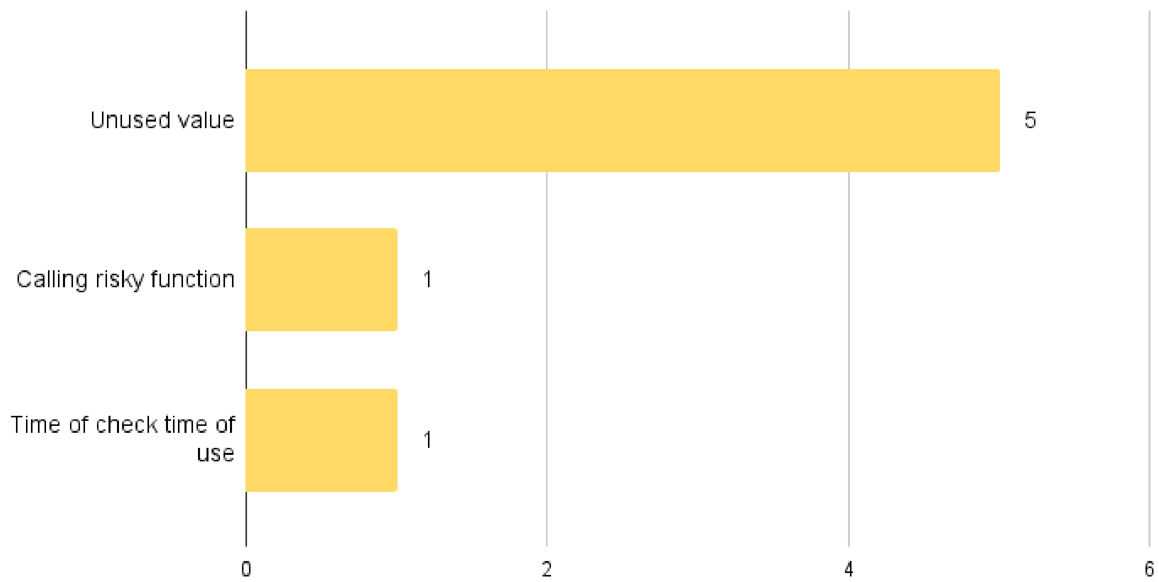


Figure A.11: Potential Low Vulnerabilities - Coverity

CWE	Vulnerability Type	Severity	File:Line
CWE-20	Untrusted value as argument	Medium	netsurf/content/fs_backing_store.c:Various
CWE-119	Out-of-bounds write	High	netsurf/content/handlers/html/html.c:178
CWE-119	Out-of-bounds access	High	netsurf/content/handlers/javascript/duktape/duktape.c:Various
CWE-119	Out-of-bounds access	High	netsurf/content/handlers/javascript/duktape/duktape.c:16374
CWE-119	Out-of-bounds access	High	netsurf/content/handlers/javascript/duktape/duktape.c:Various
CWE-119	Out-of-bounds write	High	netsurf/desktop/textarea.c:2476
CWE-119	Out-of-bounds access	High	netsurf/content/handlers/javascript/duktape/duktape.c:16409
CWE-125	Out-of-bounds read	High	netsurf/frontends/gtk/toolbar.c:Various
CWE-170	String not null terminated	High	netsurf/content/fs_backing_store.c:1162
CWE197	Use of 32-bit time_t	High	netsurf/content/urldb.c:339
CWE197	Use of 32-bit time_t	High	netsurf/content/urldb.c:Various
CWE197	Use of 32-bit time_t	High	libdom/src/events/event.c:259
CWE-252	Unchecked return value from library	Medium	netsurf/frontends/gtk/download.c:576
CWE-252	Unchecked return value from library	Medium	netsurf/frontends/gtk/download.c:870
CWE-252	Unchecked return value	Medium	netsurf/content/handlers/text/textplain.c:Various
CWE-252	Unchecked return value	Medium	libcss/src/parse/properties/content.c:417
CWE-367	Time of check time of use	Low	netsurf/frontends/gtk/gui.c:1009
CWE-369	Division or modulo by zero	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:41211
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1852
CWE-398	Copy-paste error	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:87683
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:749
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:874
CWE-398	Copy-paste error	Medium	netsurf/content/handlers/html/object.c:335
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1235
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1671
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1504
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1281
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1773
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1387
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1817
CWE-398	Big parameter passed by value	Low	libsvgtiny/src/svgtiny.c:1587
CWE-404	Resource leak	High	netsurf/content/handlers/javascript/duktape/dukky.c:1632
CWE-404	Resource leak	High	netsurf/frontends/gtk/download.c:698
CWE-404	Resource leak	High	netsurf/content/handlers/html/css_fetcher.c:319
CWE-404	Resource leak	High	netsurf/frontends/gtk/toolbar.c:2856
CWE-404	Resource leak	High	libsvgtiny/src/svgtiny_gradient.c:Various
CWE-404	Resource leak	High	netsurf/frontends/gtk/toolbar.c:2901
CWE-416	Use after free	High	libdom/src/core/node.c:124
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:14935
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:85015
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:85177
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:Various
CWE-457	Uninitialized scalar variable	High	libdom/src/core/node.c:1220
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:85177
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:Various
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:49947
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:24590
CWE-457	Uninitialized pointer read	High	netsurf/content/fetchers/curl.c:694
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:24559

CWE	Vulnerability Type	Severity	File:Line
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:78727
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:Various
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/html/table.c:544
CWE-457	Uninitialized scalar variable	High	libdom/src/core/node.c:1742
CWE-457	Uninitialized scalar variable	High	netsurf/content/handlers/javascript/duktape/duktape.c:24580
CWE-476	Dereference before null check	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:30141
CWE-476	Dereference after null check	Medium	netsurf/content/handlers/html/layout.c:4021
CWE-476	Dereference after null check	Medium	netsurf/frontends/gtk/throbber.c:116
CWE-476	Explicit null dereferenced	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:36523
CWE-476	Dereference after null check	Medium	netsurf/content/handlers/html/form.c:2146
CWE-476	Explicit null dereferenced	Medium	netsurf/content/handlers/html/box_construct.c:1007
CWE-476	Dereference before null check	Medium	netsurf/content/handlers/html/layout.c:3128
CWE-476	Dereference null return value	Medium	libcss/src/parse/mq.c:983
CWE-476	Dereference after null check	Medium	netsurf/content/handlers/html/interaction.c:Various
CWE-476	Dereference after null check	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:37145
CWE-476	Dereference after null check	Medium	netsurf/content/handlers/html/layout.c:1061
CWE-476	Explicit null dereferenced	Medium	netsurf/content/handlers/html/imagemap.c:396
CWE-476	Dereference before null check	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:30430
CWE-484	Missing break in switch	Medium	libhubbub/src/treebuilder/element-type.gperf:122
CWE-484	Missing break in switch	Medium	libsvgtiny/src/colors.gperf:74
CWE-561	Logically dead code	Medium	libcss/src/select/hash.c:366
CWE-561	Logically dead code	Medium	netsurf/content/llcache.c:866
CWE-561	Logically dead code	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:79240
CWE-561	Logically dead code	Medium	netsurf/content/fetchers/about/testament.c:114
CWE-561	Logically dead code	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:32528
CWE-561	Logically dead code	Medium	netsurf/desktop/browser_window.c:4367
CWE-561	Structurally dead code	Medium	libsvgtiny/src/svgtiny.c:1696
CWE-561	Logically dead code	Medium	netsurf/desktop/cw_helper.c:60
CWE-561	Logically dead code	Medium	netsurf/desktop/treeview.c:3967
CWE-561	Structurally dead code	Medium	libsvgtiny/src/svgtiny.c:594
CWE-561	Logically dead code	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:38626
CWE-561	Logically dead code	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:31294
CWE-561	Logically dead code	Medium	libcss/src/select/hash.c:441
CWE-561	Logically dead code	Medium	netsurf/content/llcache.c:862
CWE-561	Logically dead code	Medium	netsurf/content/llcache.c:858
CWE-561	Logically dead code	Medium	netsurf/content/fetchers/curl.c:176
CWE-561	Logically dead code	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:31938
CWE-561	Logically dead code	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:31775
CWE-561	Logically dead code	Medium	libcss/src/select/hash.c:523
CWE-561	Logically dead code	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:30597
CWE-563	Unused value	Low	libhubbub/src/treebuilder/in_body.c:790
CWE-563	Unused value	Low	libhubbub/src/treebuilder/in_body.c:1368
CWE-563	Unused value	Low	libhubbub/src/treebuilder/in_body.c:807
CWE-563	Unused value	Low	libhubbub/src/treebuilder/in_body.c:2045
CWE-563	Unused value	Low	libhubbub/src/treebuilder/in_body.c:990
CWE-569	Operands don't affect result	Medium	libcss/src/select/mq.h:99
CWE-569	Operands don't affect result	Medium	netsurf/desktop/treeview.c:1281
CWE-569	Operands don't affect result	Medium	netsurf/desktop/hotlist.c:Various
CWE-569	Operands don't affect result	Medium	netsurf/desktop/treeview.c:1352

CWE	Vulnerability Type	Severity	File:Line
CWE-569	Operands don't affect result	Medium	netsurf/content/llcache.c:1790
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:Various
CWE-569	Operands don't affect result	Medium	netsurf/desktop/browser_window.c:4690
CWE-569	Operands don't affect result	Medium	netsurf/desktop/treeview.c:1332
CWE-569	Operands don't affect result	Medium	netsurf/content/fetchers/curl.c:170
CWE-569	Operands don't affect result	Medium	netsurf/content/llcache.c:2357
CWE-569	Operands don't affect result	Medium	netsurf/content/content.c:531
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:3444
CWE-569	Operands don't affect result	Medium	netsurf/content/handlers/html/box_inspect.c:619
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:121
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:1387
CWE-569	Operands don't affect result	Medium	netsurf/content/llcache.c:860
CWE-569	Operands don't affect result	Medium	netsurf/content/llcache.c:2352
CWE-569	Wrong sizeof argument	Medium	netsurf/content/handlers/javascript/duktape/duktape.c:92624
CWE-569	Operands don't affect result	Medium	netsurf/content/fetchers/about/about.c:641
CWE-569	Operands don't affect result	Medium	netsurf/desktop/treeview.c:1427
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:3784
CWE-569	Operands don't affect result	Medium	libcss/src/select/arena.c:83
CWE-569	Operands don't affect result	Medium	libcss/src/parse/language.c:875
CWE-569	Operands don't affect result	Medium	netsurf/content/fetchers/curl.c:1311
CWE-569	Operands don't affect result	Medium	libcss/src/parse/font_face.c:Various
CWE-569	Operands don't affect result	Medium	libcss/src/select/arena.c:60
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:145
CWE-569	Operands don't affect result	Medium	netsurf/content/llcache.c:1818
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:137
CWE-569	Operands don't affect result	Medium	netsurf/desktop/hotlist.c:504
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:113
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:153
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:1398
CWE-569	Operands don't affect result	Medium	netsurf/desktop/browser_window.c:4704
CWE-569	Operands don't affect result	Medium	netsurf/content/fetch.c:192
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:3373
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:129
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/parse.c:1307
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:783
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:1335
CWE-569	Operands don't affect result	Medium	netsurf/content/handlers/css/select.c:724
CWE-569	Operands don't affect result	Medium	libcss/src/select/mq.h:134
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:4174
CWE-569	Operands don't affect result	Medium	netsurf/desktop/treeview.c:1443
CWE-569	Operands don't affect result	Medium	netsurf/content/fetchers/resource.c:353
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:161
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:4209
CWE-569	Operands don't affect result	Medium	libcss/src/select/mq.h:124
CWE-569	Operands don't affect result	Medium	netsurf/utils/nsurl/nsurl.c:169
CWE-569	Operands don't affect result	Medium	libdom/src/core/string.c:1019
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:Various
CWE-569	Operands don't affect result	Medium	libcss/src/select/select.c:1801
CWE-569	Operands don't affect result	Medium	netsurf/content/fetchers/curl.c:175

CWE	Vulnerability Type	Severity	File:Line
CWE-569	Operands don't affect result	Medium	netsurf/content/fetch.c:183
CWE-569	Operands don't affect result	Medium	libdom/src/core/element.c:1222
CWE-569	Operands don't affect result	Medium	netsurf/content/llcache.c:864
CWE-569	Operands don't affect result	Medium	netsurf/content/fetch.c:136
CWE-569	Operands don't affect result	Medium	netsurf/content/content.c:508
CWE-569	Operands don't affect result	Medium	libcss/src/parse/language.c:928
CWE-569	Operands don't affect result	Medium	libcss/src/select/select.c:2356
CWE-569	Operands don't affect result	Medium	libcss/src/select/mq.h:145
CWE-569	Operands don't affect result	Medium	netsurf/desktop/browser_window.c:4712
CWE-569	Operands don't affect result	Medium	netsurf/content/llcache.c:856
CWE-569	Operands don't affect result	Medium	netsurf/content/urldb.c:3187
CWE-606	Untrusted loop bound	Medium	netsurf/content/urldb.c:Various
CWE-676	Calling risky function	Low	netsurf/content/handlers/image/image_cache.c:314

A.4 Ignore Files

Code listing A.1: .snyk File

```
# Snyk (https://snyk.io) policy file
exclude:
 global:
   - buildsystem/**
   - inst-gtk2/share/netsurf-buildsystem/**
   - libnsbmp/**
   - libnsfb/**
   - libnslog/**
   - libnspsl/**
   - libnsutils/**
   - libparserutils/**
   - libpencil/**
   - librosprite/**
   - librufl/**
   - libutf8proc/**
   - libwapcaplet/**
   - netsurf-nix/**
   - nsgenbind/**
   - Makefile
   - env.sh
   - flake.lock
   - flake.nix
   - flawfinder_out
   - netsurf/frontends/amiga/**
   - netsurf/frontends/atari/**
   - netsurf/frontends/riscos/**
   - netsurf/frontends/beos/**
   - netsurf/frontends/monkey/**
   - netsurf/test/**
   - libcss/test/**
   - libdom/test/**
   - libhubbub/test/**
   - libhubbub/perf/**
   - libsvgtiny/test/**
   - libdom/examples/**
   - libsvgtiny/examples/**
   - libhubbub/examples/**
```

Code listing A.2: .semgrepignore File

```
buildsystem/
inst-gtk2/share/netsurf-buildsystem/
libnsbmp/
libnsfb/
libnslog/
libnspsl/
libnsutils/
libparserutils/
libpencil/
librosprite/
librufl/
libutf8proc/
libwapcaplet/
netsurf-nix/
nsgenbind/
Makefile
env.sh
flake.lock
flake.nix
flawfinder_out
```

```
21 netsurf/frontends/amiga/
22 netsurf/frontends/atari/
23 netsurf/frontends/riscos/
24 netsurf/frontends/beos/
25 netsurf/frontends/monkey/
26 netsurf/test/
27 libcss/test/
28 libdom/test/
29 libhubbub/test/
30 libhubbub/perf/
31 libsvgtiny/test/
32 libhubbub/examples/
33 libsvgtiny/examples/
34 libdom/examples/
```

A.5 Nix Flake

```
{
  description = "Flake for debugging and fuzzing NetSurf";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-23.11";
  };

  outputs = { self, nixpkgs, ... }@inputs:
  let
    systems = [
      # GDB and AFL are only available for linux
      "x86_64-linux"
      "aarch64-linux"

      # ... but everything else should work on mac as well
      "x86_64-darwin"
      "aarch64-darwin"
    ];
    forAllSystems = f: nixpkgs.lib.genAttrs systems (system: f system);
  in {
    packages = forAllSystems (system: rec {
      pkgs = import nixpkgs { inherit system; };

      netsurf-gtk3 = with pkgs; (lib.recurseIntoAttrs (callPackage \
            ./netsurf-nix { })).overrideScope’ (final: pref: {
        ui = "gtk3";
        enableDebugging = true;
      });

      netsurf-fb = with pkgs; (lib.recurseIntoAttrs (callPackage \
            ./netsurf-nix { })).overrideScope’ (final: pref: {
        ui = "framebuffer";
        enableDebugging = true;
        forceEnableScripting = true;
      });

      netsurf-fb-afl = with pkgs; if (lib.hasSuffix "linux" system) then
        ((lib.recurseIntoAttrs (callPackage ./netsurf-nix { })).overrideScope’ \
            (final: prev: {
          ui = "framebuffer";
          enableDebugging = false;
          exitWhenDone = true;
          forceEnableScripting = false;
          stdenv = prev.stdenv_afl;
        })) else { };
    });
```

```
47
48     devShells = forAllSystems (system: rec {
49       pkgs = import nixpkgs { inherit system; };
50
51       # "$ coredumpctl gdb" always runs "gdb" from your path.
52       pwndbg-with-alias = pkgs.pwndbg.overrideAttrs ({ installPhase ? "", ... }: {
53         installPhase = installPhase + ''
54           ln -s $out/bin/pwndbg $out/bin/gdb
55         '';
56       });
57
58       default = pkgs.mkShell {
59         nativeBuildInputs = with pkgs; with self.packages.${system}; [
60           netsurf-fb.browser
61           netsurf-gtk3.browser
62
63           aflplusplus
64         ] ++ lib.optionals stdenv.isLinux [
65           pwndbg-with-alias
66         ];
67       };
68     });
69   };
70 }
```

Code listing A.3: Nix Flake

A.6 Docker Configuration

Code listing A.4: Dockerfile to Run AFL++ on NetSurf

```
FROM nixos/nix:latest AS builder

RUN nix-build '<nixpkgs>' -A aflplusplus

COPY . /netsurf
WORKDIR /netsurf

RUN nix \
    --extra-experimental-features "nix-command flakes" \
    --option filter-syscalls false \
    build ".#netsurf-fb-afl.browser"

ENV AFL_AUTORESUME=1
ENV AFL_NO_AFFINITY=1

# These settings should be set on your host for increased performance, but they are not required.
ENV AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1
ENV AFL_SKIP_CPUFREQ=1

CMD ["nix-shell", "-p", "aflplusplus", "--command", "afl-fuzz -i /fuzzing/inputs -o /fuzzing/output \
-e html -M fuzz01 -- /netsurf/result/bin/netsurf-fb -f ram file://@@"]
```

Code listing A.5: Docker Compose Configuration for Running Multiple Instances of afl-fuzz

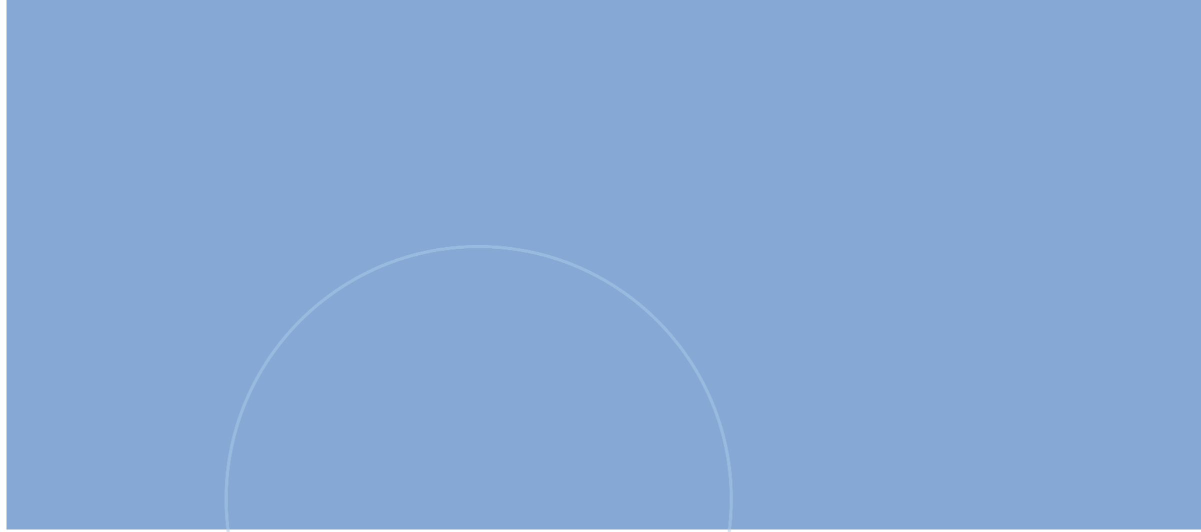
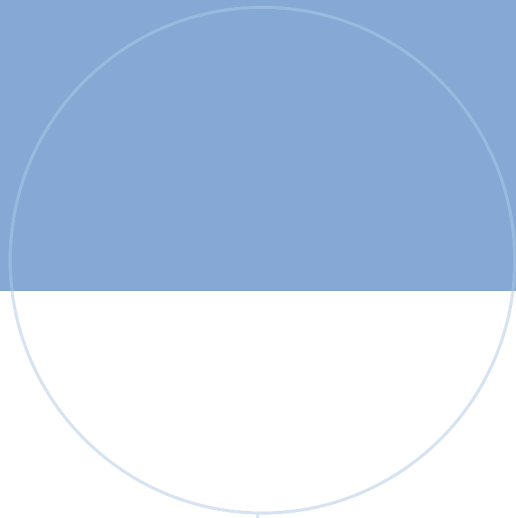
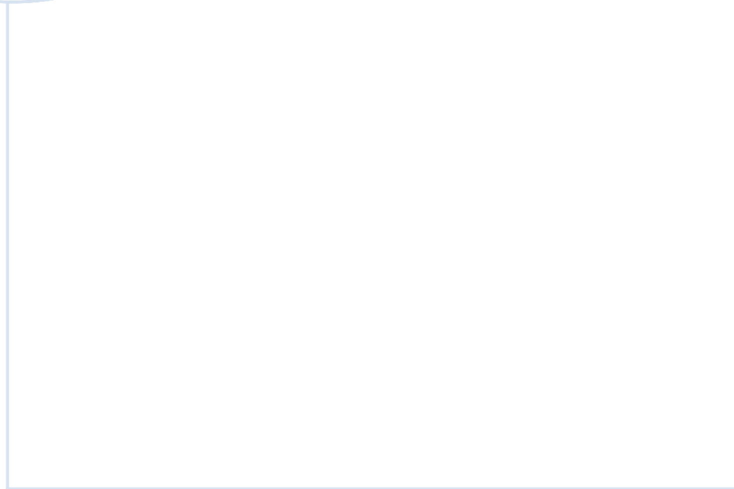
```
services:
  afl-master:
    build: .
    tmpfs:
      - /ramdisk
    environment:
      - AFL_TMPDIR=/ramdisk
    volumes:
      - type: bind
        source: ./fuzzing/inputs_x
        target: /fuzzing/inputs
      - type: bind
        source: ./fuzz-output
        target: /fuzzing/output
    network_mode: none
    stdin_open: true
    tty: true
    command: nix-shell -p aflplusplus --command "afl-fuzz -i /fuzzing/inputs -o /fuzzing/output \
    -e html -M fuzz_master -- /netsurf/result/bin/netsurf-fb -f ram file://@@"

  afl-worker:
    build: .
    tmpfs:
      - /ramdisk
    environment:
      - AFL_TMPDIR=/ramdisk
    volumes:
      - type: bind
        source: ./fuzzing/inputs_x
        target: /fuzzing/inputs
      - type: bind
        source: ./fuzz-output
        target: /fuzzing/output
    network_mode: none
    stdin_open: true
    tty: true
```

```
37        command: nix-shell -p aflplusplus --command "afl-fuzz -i /fuzzing/inputs -o /fuzzing/output \
38        -e html -S fuzz_$(cat /etc/hostname) -- /netsurf/result/bin/netsurf-fb -f ram file://@@"
39
40
41    # Note:
42    # Preparation:
43    # - Choose a set of inputs, e.g. "3".
44    # - cp ./fuzzing/inputs_3 into ./fuzzing/inputs_x
45    # - mkdir fuzz-output
46    #
47    # Running:
48    # - docker compose up --detach --scale afl-worker=X
49    #   - "X" is the number of extra workers (in addition to the master). 4 CPU Cores -> X=3
50    # - See the status:
51    #   - docker logs -f --tail 50 netsurf-all-afl-master-1
52    #   - docker logs -f --tail 50 netsurf-all-afl-worker-X
53    #
54    # The "logs" show the status, and can be opened and closed(^C) at any time.
55    # When exiting, your shell might be confused and look weird. If so, run ‘reset‘.
```

Code listing A.6: Basic Dockerfile for Debugging an Unmodified NetSurf

```
1     # docker build . -t netsurf_report
2     # docker run --security-opt seccomp=unconfined -it netsurf_report
3     FROM debian@sha256:1aadfee8d292f64b045adb830f8a58bfacc15789ae5f489a0fedcd517a862cb9
4
5     ENV DEBIAN_FRONTEND noninteractive
6     RUN apt update && apt upgrade
7     RUN apt install -y \
8         make \
9         git \
10        bison \
11        flex \
12        gperf \
13        libcurl3-gnutls \
14        libcurl3-nss \
15        libcurl4 \
16        libcurl4-openssl-dev \
17        libpng-dev \
18        libjpeg-dev
19
20    RUN git clone https://github.com/pwndbg/pwndbg
21    WORKDIR /pwndbg
22    RUN /pwndbg/setup.sh
23
24    WORKDIR /
25    COPY netsurf-all-3.11.tar.gz /
26    RUN tar -xvf /netsurf-all-3.11.tar.gz
27
28    WORKDIR /netsurf-all-3.11
29    RUN make TARGET=framebuffer
30    ADD inputs /html_inputs
31    ENV NETSURFRES=/netsurf-all-3.11/netsurf/frontends/framebuffer/res
32
33    CMD ["/bin/bash"]
```

Norwegian University of
Science and Technology