TÉCNICO LISBOA

Towards Automatic Detection of Exploitable Vulnerabilities in WebAssembly Modules of Real-World Websites

Daniel Maximiano Matos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Nuno Miguel Carvalho dos Santos
Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. José Alberto Rodrigues Pereira Sardinha
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Rodrigo Fraga Barcelos Paulus Bruno

November 2022

Acknowledgments

I want to show my gratitude to my family for their encouragement and support throughout my journey. I would also like to acknowledge my supervisors Prof. Nuno Santos and Prof. Pedro Adão, and also Prof. José Fragoso for all the advice, support, availability, and guidance that made this work possible. Next, I would like to thank Nuno Duarte for all his help in developing an essential part of this work and all his support. Lastly, I would like to thank my friends that supported me during my stay at Técnico. To each and every one of you – Thank you.

Resumo

WebAssembly (Wasm) é um formato de código binário que serve de alvo de compilação para linguagens alto nível, de forma a que possa ser executado num browser, com velocidade quase nativa. Apesar de ter múltiplas vantagens devido ao seu formato compacto, o WebAssembly permite a ocorrência de vulnerabilidades em programas Web, tal como acontece com os binários de *x86*. Dado ser uma tecnologia relativamente recente, a sua presença em websites reais ainda não foi devidamente identificada, mas a tendência é para que esteja cada vez mais predominante na Web, assim como as suas vulnerabilidades.

Nesta tese apresentamos o WebAF, uma framework escalável para análise da web que nos permite obter páginas web com submódulos de WebAssembly. Ela consegue reunir um conjunto de dados representativo de binários de WebAssembly que é util para testar a geração automática de exploits para as vulnerabilidades encontradas. Nós também propomos o WebAssault, a nossa framework de exploração automática de vulnerabilidades, e mostramos que ela consegue gerar exploits simples de prova de conceito.

Palavras-chave: WebAssembly, Geração de Exploits, Análise da Web, Vulnerabilidades

Abstract

WebAssembly (Wasm) is a binary code format that is a compilation target for high-level languages, that allows them to run on a browser with near-native performance. Besides its multiple advantages due to its compact format, WebAssembly allows for the presence of vulnerabilities on Web programs, just like in *x86* binaries. Due to being a relatively recent technology, its presence in real-world websites was not yet properly studied, but it has the tendency to be more and more prevalent on the Web, as well as its vulnerabilities.

In this thesis, we present WebAF, a scalable web crawling framework enabling us to obtain webpages with WebAssembly submodules. It can collect a representative dataset of WebAssembly binaries that are useful for testing automated exploits on the found vulnerabilities. We also present WebAssault, our automatic exploitation framework, and show that it can generate simple proof-of-concept exploits.

Keywords: WebAssembly, Exploit Generation, Web Analysis, Vulnerabilities

Contents

List of Tables

List of Figures

1

Introduction

Contents

1.1 Motivation

Due to JavaScript's poor performance when it comes to web-based applications, new languages have emerged to overcome this problem. Emerging languages also include new features and can give stronger safety guarantees. Some alternatives that have been developed to overcome this performance problem are Asm.js [1] and Google Native Client [2]. As of now, the most popular is WebAssembly [3], a compilation target for languages like C and Rust, which is a byte-code language intended for a portable virtual machine, *i.e.* uses a virtual Instruction Set Architecture (ISA). Also known as Wasm, WebAssembly can run with a near-native performance which makes it very attractive to port existing applications to it. Due to WebAssembly's benefits and popularity, the other two mentioned alternatives have been deprecated and are no longer maintained [4, 5].

Although WebAssembly code is sandboxed and features a stack-based virtual machine that is type-checked after every instruction, security vulnerabilities are still common and can sometimes be ported from the source code originally written in unsafe languages like C or C++ to the WebAssembly binary. Because WebAssembly is a virtual ISA, it can run on multiple platforms such as web browsers or server-side web containers. This means that undetected security vulnerabilities can transition to WebAssembly, e.g., buffer overflows can cripple many applications that depend on WebAssembly code running across various platforms. As a consequence of this, several tools have been developed in order to detect the possible vulnerabilities that can occur in this low-level language, being Wasmati [6] and WASP [7] the most relevant ones for a part of our work.

To better identify the possible vulnerabilities that the byte-code language may inherit from its source, the strategy used in other languages was also applied in Wasmati and WASP: recurring to *static analysis* and *symbolic execution*, respectively, to detect bugs in the code. On the static analysis side, we have Microsoft PREfast [8] that analyses C and C++ source files, JSLint [9] to check on JavaScript code, and RIPS [10] which checks the interactions between sources, sinks, and sanitizations in PHP files. Along the same vein as these tools, Wasmati employs static analysis to detect vulnerabilities in WebAssembly binaries. However, although these techniques are very efficient, they can generate a lot of false positives, requiring a great deal of manual effort to validate if the vulnerabilities flagged by the tool are true or false.

On the symbolic execution side, we have tools like Manticore [11] for smart contracts and binaries, WANA [12] for WebAssembly, and KLEE [13] for C/C++ and Rust, which are all symbolic execution engines that find bugs in their specific area. Likewise, WASP is a robust concolic execution tool for WebAssembly. The potential of these tools is that they can accurately prove the presence of a given vulnerability by generating an input that can trigger the vulnerable code path. However, a central drawback of symbolic execution, in general, is scalability and performance. In general, tools of this nature have to make simplifications in the code exploration algorithms in order to deal with path explosion problems. This is necessary because symbolic execution takes significant time to execute, and programs generally have many execution paths that need to be explored.

In this work, we aim to investigate the possibility of combining the best of both worlds in order to improve vulnerability detection capabilities for WebAssembly code. In theory, we can prune out the

false positives generated by Wasmati by manually checking if the detected vulnerabilities are exploitable or not. However, testing the exploitability of vulnerabilities may not be achievable by hand because they might be numerous, and exploits are time-consuming to write. To speed up this process, our idea is to employ automatic exploit generation, as first proposed in [14]. With this approach, we strive to simultaneously identify vulnerabilities and test their exploitability by software in a fully automated fashion. This way, developers can prioritize the vulnerabilities that must be patched first.

Toward this end, however, we need to obtain a representative dataset of WebAssembly binaries so that we can evaluate the effectiveness of our technique in finding security vulnerabilities. Unfortunately, as of the time of this writing, no such dataset has yet been collected by the research community. Moreover, there is no prior study focused only on investigating how prevalent Wasm binaries have become on the Internet. As a result, we cannot be sure how common WebAssembly vulnerabilities are in real-world web applications: if they are present on a small number of websites or if tech giants are also affected. Therefore, to test the viability of our vulnerability detection technique we must first collect such a dataset which in itself requires the development of a scalable web crawling framework for fetching webpages containing subcomponents written in WebAssembly.

1.2 Objectives

In light of the context clarified above, this thesis has three main goals:

- Obtain an up-to-date dataset of real-world webpages, containing HTML, JavaScript, and WebAssembly, allowing us to perform multiple types of analyses and queries on the obtained data.
- Characterize the obtained dataset according to the presence of WebAssembly and JavaScript data types being used and shared between the two languages.
- Generate proof-of-concept exploits for WebAssembly vulnerabilities in the dataset, showing the viability of static analysis and symbolic execution as a precursor to automatic exploit generation.

1.3 Contributions

This thesis makes the following main contributions. Firstly, we developed an analysis framework capable of navigating the web and scanning any Wasm binaries for security vulnerabilities they could contain. More specifically, our framework is called WebAF and it can scale easily if we want it to span an enormous amount of computing power while keeping it reliable and simple to use. The framework contains a web crawler and it has an extension system, allowing us to extend it with any plugin we want, to perform various tasks in a website response, that can be later saved and analyzed.

The second contribution which resulted from our efforts is a web page dataset that is up-to-date, meaning it has contents that are more recent than other datasets that may already exist and contain features that may not exist in others; is representative, in the sense that the websites that are present

make a good representation of what can be found by randomly navigating the web; and that is extensive, in order to allow performing multiple studies.

Finally, the third contribution of this thesis consists of proof-of-concept exploits generated by combining Wasmati and WASP. This is a preliminary step toward the future development of WebAssault, an automatic vulnerability detector and exploit generator for the WebAssembly language. WebAssault will receive as input a WebAssembly program and will use it as input for the static analyzer Wasmati [6] to detect possible vulnerabilities. With the obtained report, it will try to rewrite the program in a way that the symbolic execution engine WASP [7] will be able to check only the execution paths that lead to vulnerabilities. In a nutshell, by leveraging static analysis, we are able to get an overview of the possible vulnerabilities that might be present. Then, using concolic execution (which is faster than classic symbolic execution) we can discard the false positives, ending with a set of vulnerabilities that are effectively present in the code and also with the inputs that can trigger them. To make this technique work in practice, there are multiple challenges regarding the pruning of the execution paths and the generation of symbolic arrays that WebAssault will overcome with a set of novel techniques. These challenges will be addressed in future work and therefore fall outside the scope of this thesis.

1.4 Thesis Outline

This thesis is structured as follows:

- Chapter 2 provides an overview of Wasm vulnerabilities, focusing on how they are inherited from the source language, and introduces WASP and Wasmati, the tools we are going to use.
- Chapter 3 discusses all the relevant investigation work relevant for this thesis.
- Chapter 4 describes how our framework is composed.
- Chapter 5 explains how we obtained the dataset that we will analyze and enumerates the analyses performed by the framework and the results obtained from the gathered dataset.
- Chapter 6 starts by illustrating our solution for automatic exploit analysis, shows how WebAssault works and explains its limitations.
- Chapter 7 sums up our work and points directions for future work based on this project.

2

Background

Contents

This chapter provides some background for this thesis. In Section 2.1 we start by introducing the WebAssembly language. Next, Section 2.2 takes an overview of how WebAssembly vulnerabilities are inherited from the source language. We take a brief look at Wasmati in Section 2.3, a static analysis tool, and also at WASP in Section 2.4, a symbolic execution engine, both being a dependency of WebAssault. Finally, we wrap up the chapter with a summary.

2.1 WebAssembly Overview

WebAssembly, which was introduced in [3], is a compilation target that offers a compact binary representation, with low-level control over the memory and instructions closely mapped to the ones in hardware, having a great performance, compared to JavaScript. Despite being a binary format, it has a language syntax and structure allowing developers to write WebAssembly by hand.

A WebAssembly binary is distributed in *modules*, which have the definitions of *functions*, *memories*, *tables* and *globals*. It runs on a stack machine, where instructions can push or pop values to the stack, according to their semantics. There are four primitive types in WebAssembly—`i32`, `f32`, `i64` and `f64`—that occupy either 32 or 64 bits, so we need a place to store more complex data structures, like strings and C structs. In order to achieve this, WebAssembly has a *linear memory*, which is simply a continuous buffer where we can access its positions with an index. Dynamically allocated memory (heap), local variables, and globals are also stored in this memory, and it can be increased with the instruction `grow`.

The WebAssembly VM has two separate stacks: the *unmanaged stack*, which is where the local variables are stored (*i.e.* in the linear memory); and the execution stack where the values that are operated by the instructions are saved. Return addresses exist but are not accessible through code, in order to protect against control flow attacks when there are buffer overflows in the stack or heap.

2.2 WebAssembly Vulnerabilities

WebAssembly is a language that can be run anywhere and is not intended to be written by hand, so it is used as a compilation target from many other languages, like C/C++, Rust, Python, and Java. Some of these languages are unsafe, *i.e.*, they do not guarantee that the program will behave well when given certain inputs. By using WebAssembly as a compilation target, from those languages, some vulnerabilities are transferred to the WebAssembly binary, so they can also be triggered in a web application by Javascript code [15].

Some of the vulnerabilities that are present in WebAssembly binaries are:

- *Buffer overflows*: Since the current toolchain for WebAssembly does not introduce canaries in linear memory, the usage of unsafe functions in the source, such as `gets`, `strcpy` and `strcat`, which do not perform bounds checking of the destination, may incur in a buffer overflow. Adding to this, if the programmer performs a mistake and introduces a value bigger than the length of the buffer that is being used, there is no way to detect if data has been written out of bounds

and so, execution cannot be terminated, as it does in native binaries. If this type of vulnerability occurs in the unmanaged stack, it can change the values of local values of functions, even outside the current frame, which poses a big threat to security. This type of overflow can also occur in dynamically allocated memory, which also resides in the linear memory, but with the extra that its metadata can be easily changed since it is stored on the same spot and the indexes are going to be deterministic.

- *Format strings*: This type of vulnerability occurs when a function that expects a format string as one of its arguments is passed a non-constant string, *i.e.*, a user-controlled value. This vulnerability in WebAssembly is inherited from the source language; the programmer may have forgotten to introduce a constant value for the format string or might not be aware that one should be used at a predefined position. In WebAssembly, such vulnerability is able to change any value that is present in the linear memory.

- *Integer overflows*: As stated above, WebAssembly's primitive types occupy either 32 or 64 bits. Integer overflows arise due to JavaScript representing both integer and floating point values using the `Number` type which can only take values between $-(2^{53}-1)$ and $2^{53}-1$ [16], *i.e.*, values that use 53 bits. An overflow happens when the values close to the maximum value that can be stored in a 32-bit integer $(2^{32}-1)$ are passed to a WebAssembly module that is going to perform arithmetic operations on it, such that the value instead of increasing (as we supposed should happen) is going to decrease since the value cannot fit in 32 bits. When this value is then passed back to JavaScript, it will not be the expected value, due to the lost bits.

In Figure 2.1 we provide an example of a buffer overflow vulnerability in C code, a real-world vulnerability, that is going to be moved to the underlying WebAssembly binary. This vulnerability was present in the popular image manipulation library *libpng*(CVE-2018-14550) [17] and occurs because we are writing to the `token` buffer (line 23) without checking if we are out-of-bounds (lines 19-24). Consequently, one can overwrite memory outside the `token` buffer space. If the given file content is larger than the size of the buffer and does not contain a blank space, the buffer will be filled and will cause the buffer to overflow to other memory regions. If this library is then used on the web, it can generate abnormal behavior on the page, like XSS, or trigger some other web vulnerability.

2.3 Wasmati Overview

Wasmati [6], whose architecture we can observe in Figure 2.2, is a static analysis tool for WebAssembly that analyses either binary code or WABT and generates a Code Property Graph (CPG) [18] of the code. CPG is a graph-based data structure that combines information from Call Graph, Abstract Syntax Tree, Control Flow Graph, and Program Dependency Graph. Wasmati can help us detect various types of vulnerabilities, for instance: use-after-free, double-free, buffer overflow, format string, integer overflow, and taint-style vulnerabilities. These vulnerabilities can be detected with a low false

```c
void get_token(FILE *pnm_file, char *token) {
  int i = 0;
  int ret;

  /* remove white-space and comment lines */
  do {
    ret = fgetc(pnm_file);
    if (ret == '#') {
      /* the rest of this line is a comment */
      do {
        ret = fgetc(pnm_file);
      } while ((ret != '\n') && (ret != '\r') && (ret != EOF));
    }
    if (ret == EOF) break;
    token[i] = (unsigned char) ret;
  } while ((token[i] == '\n') || (token[i] == '\r') || (token[i] == ' '));

  /* read string */
  do {
    ret = fgetc(pnm_file);
    if (ret == EOF) break;
    i++;
    token[i] = (unsigned char) ret;
  } while ((token[i] != '\n') && (token[i] != '\r') && (token[i] != ' '));

  token[i] = '\0';

  return;
}
```

Figure 2.1: Example of buffer overflow vulnerability in libpng

negative rate, but Wasmati still generates many false positives as a result of being a static analyzer and not having all of the information to discard some of the detected vulnerabilities.

More specifically, with this tool, we get a serialized CPG graph that can be later queried in order to detect security vulnerabilities that may come from the C language, the source of most modules. Running Wasmati with our example from Figure 2.1 produces the output we can see in Figure 2.3. It detects that a buffer overflow occurs inside the second `do-while` loop because no bounds-checking is performed inside of it. The file `pnm_file`, which is controlled by the user, could be bigger than the space allocated for `token` since it is stored on the disk which has more space than `token` can possibly have, so an attack could be launched by a buffer overflow in this function. Unfortunately, although Wasmati can identify this vulnerability, this tool cannot tell us which are the concrete values of each variable in order to trigger the vulnerability. As a result, we can only infer that the vulnerability may exist, but not be certain about what kind of concrete input needs to be given to the application in order to reach the vulnerable code.

2.4 WASP Overview

WASP [7] is a symbolic execution engine that performs concrete and symbolic execution—concolic execution—over WebAssembly programs without needing to access the source code. Symbolic execu-

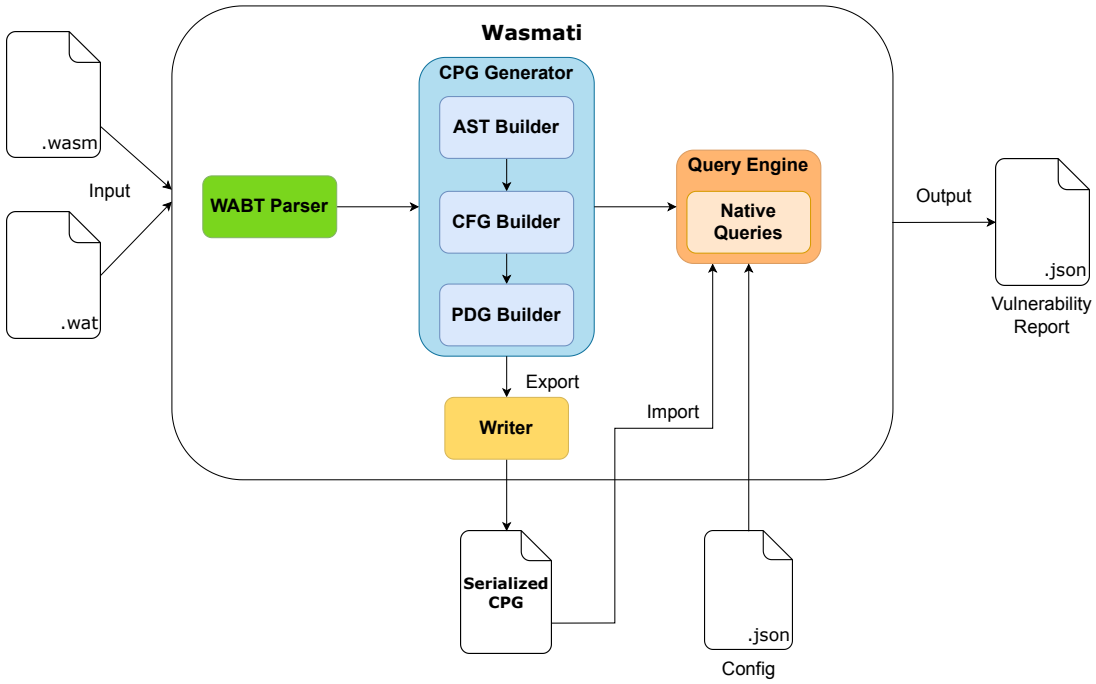


Figure 2.2: Wasmati architecture.

```
{
    "vulnerabilities": [
        {
            "description": "In loop $L4: a buffer is assigned without bound check.",
            "function": "$get_token",
            "type": "Buffer Overflow"
        }
    ]
}
```

Figure 2.3: Output of Wasmati for buffer overflow in libpng.

tion runs a program determining which inputs run which parts of the program, without assuming their actual values. Concolic execution is a special type of symbolic execution, where a symbolic analysis is run over a concrete execution, *i.e* over an execution where we have actual values for our inputs. This tool "concolically" runs the program for multiple rounds. For each round, it feeds the program a set of inputs, which allow the tool to explore a given path. Then, as the execution unfolds, constraints are generated and passed to a constraint solver, which will generate a new input that will explore a new path. With WASP, we are able to obtain concrete values for each of the variables in the program in case a vulnerability is reached. It uses Z3 [19], an efficient constraint solver, to obtain concrete inputs to use for the concolic execution. Z3 is called by WASP once per path, so it is very efficient.

We can take a look at its high-level architecture in Figure 2.4. WASP uses the WebAssembly reference interpreter to parse WebAssembly binaries and adds new instructions to it, allowing us to control the paths that are going to be explored by WASP. We have available the following instructions:

- `sym_assume`. Checks the value at the top of the stack and if it is equal to *0*, the current concrete execution is discarded. The negation of the symbolic expression associated with the value on top

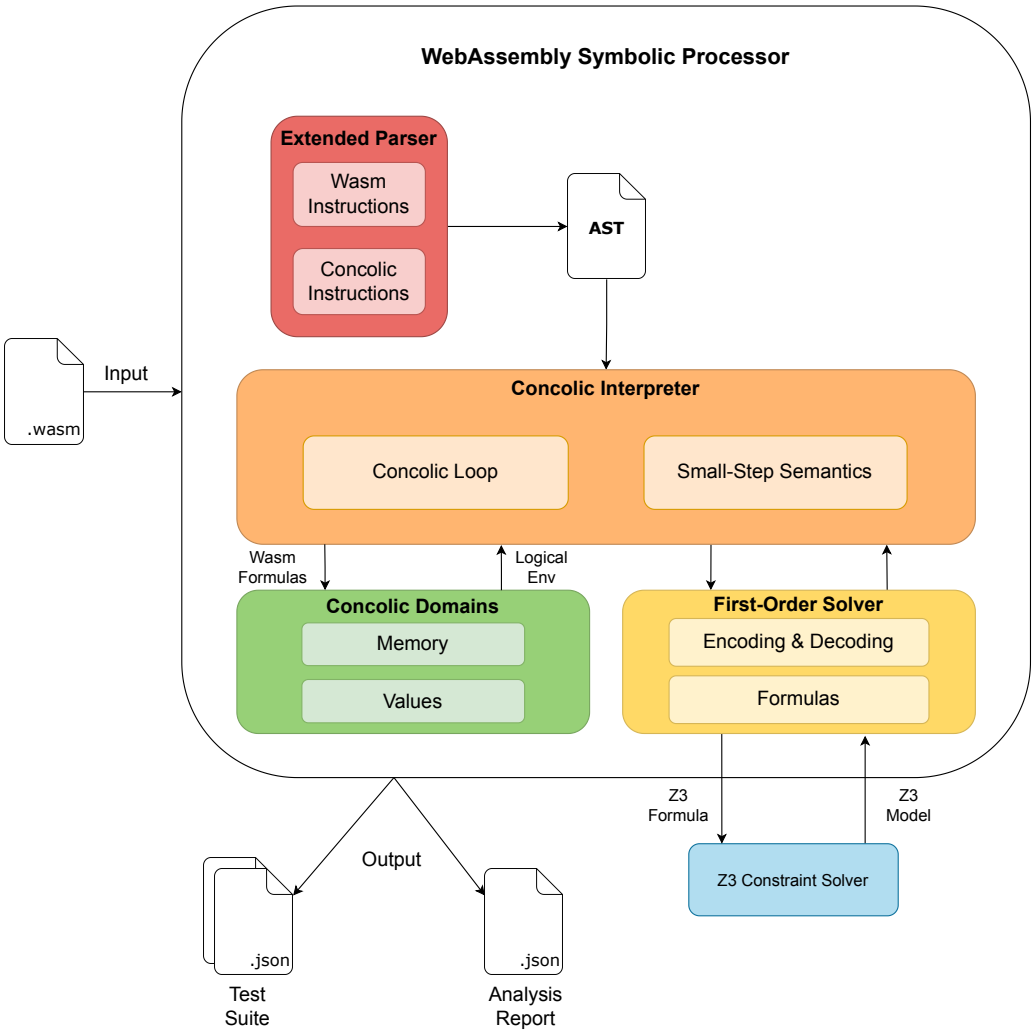


Figure 2.4: WASP architecture.

of the stack is negated and added to the path condition in order to get a new concrete execution. By adding this instruction, we can improve the speed of the concolic execution by adding constraints to the initial path condition, allowing us to get one correct input, to explore a given path more quickly.

- `sym_assert`. Checks the top of the stack and if the value is equal to *0*, the concolic execution stops with an error. Otherwise, it checks if the negation of the symbolic expression of the top of the stack in conjunction with the path condition is satisfiable, and if not, execution also stops. This is useful when we want to ensure that the execution follows strict rules, allowing us to end the computation earlier.

- `sym_int32` ψ, `sym_int64` ψ, `sym_float32` ψ, `sym_float64` ψ. Create a symbolic variable named ψ with the given type. These instructions let the concolic executor know that it must find a concrete value for the given variable.

For our example in Figure 2.1, we are able to obtain the concrete values for our input, in order to trigger the vulnerability in our example. However, because we are only relying on a concolic execution

engine, it will explore *all* the possible paths, which may take a long time to compute. Our goal is to combine Wasmati and WASP and draw the strengths of each tool to improve the overall accuracy and scalability of vulnerability detection for WebAssembly code.

Summary

This chapter provides some necessary background for this thesis. First, we looked at WebAssembly language, taking a look at some of its advantages. Next, we introduced the vulnerabilities that can be present in WebAssembly due to bugs in the source language that originated the code. Finally, we presented the two vulnerability scanners we are going to use: Wasmati, a static analyzer, and WASP, a concolic executor.

3

Related Work

Contents

As explained previously, the growing interest in WebAssembly and the difficulty of identifying exploitable bugs in the code motivates our work, whose goals are to identify how prevalent Wasm binaries are in the wild and build adequate tools for automatically detecting and exploiting vulnerabilities in this language. In this chapter, we discuss the related work. We start by taking a look at the overall security of WebAssembly and also at some of the efforts to detect vulnerabilities in WebAssembly binaries. Then, we present some of the existing tools and methods that have been used for automatic exploit generation. Finally, we review some of the existing web crawlers and web datasets used in previous studies.

3.1 WebAssembly (In)Security

WebAssembly was designed with safety and security in mind, being highlighted in both the initial publication [3] and in the official website [20]. In fact, security is a top priority in this language, since the virtual machine will run untrusted code both on the client and server sides. However, due to its semantics and implementation, vulnerabilities that are not present in native binaries can now be exploited in WebAssembly. Due to its simplicity, we can also find vulnerabilities like buffer and integer overflows, that are present in unsafe source languages, namely C and C++, and are inherited by the generated WebAssembly code. We will start by analyzing some types of protections in the binaries.

Protection	WebAssembly	Native
Stack Canaries		X
Address Space Layout Randomization (ASLR)		X
Data Execution Prevention (NX)		X
Protected Call Stack	X	
Type-Checked Indirect Calls	X	
Function Calls Restricted to a Subset	X	

Table 3.1: Protection mechanisms present in native and WebAssembly executables.

As we can see from Table 3.1, WebAssembly binaries lack stack canaries and NX protection. As stated in [21], these protections are not needed by WebAssembly programs, because return addresses are stored in a separate location—the *protected stack*—and cannot be affected by buffer overflows. Also, control flow integrity is preserved by the type system at runtime, checking if an indirect call can be executed and by using only an index to select the function to be executed, from all the ones that are present in the module. The NX protection is not of much use in WebAssembly since there is no instruction pointer and code cannot be modified, and also because there are no permissions for the linear memory, there is no way to implement such behavior.

As stated in Section 2.1, in WebAssembly there are only four primitive types and they are all numeric, so strings are stored in the linear memory. Since the unmanaged stack and the heap are stored in linear memory, this linear memory needs to be writable and readable. However, string literals that our program may use are also stored there. Moreover, since this is a continuous buffer allocated all on the same page, there is no way to create a read-only section to prevent tampering with these literals. So, a buffer overflow vulnerability can be leveraged for Cross-Site Scripting (XSS) attacks if a function that would return a constant value, can now return a value that has been changed to an XSS payload. This

behavior has been first identified in [15] and can also be performed using a format string vulnerability, but it is much harder to execute.

According to [15], XSS can also be mounted by exploiting ill-used indirect function calls. This risk may occur if the source code takes advantage of the Emscripten functions, like `emscripten_run_script`, that allow WebAssembly modules to execute arbitrary JavaScript code, behaving exactly as the function `eval`. This can be done by overwriting the argument given to these functions, which is stored in the linear memory and consequently can be changed, letting an attacker execute any JavaScript code and allowing an XSS attack.

Another relevant defense mechanism used in other platforms is Address Space Layout Randomization (ASLR). Besides not being present in WebAssembly yet, the official website states that ASLR may come in the future [21]. However, brute-forcing 32 bits is something feasible nowadays. As a result, ASLR will not introduce a big challenge for the attacker to defeat.

In WebAssembly, by type-checking indirect function calls and by restricting the functions a module can call to the ones that have been defined in its own module or the ones imported, the number of functions an attacker is able to call is significantly reduced when compared to the native binaries. In x86 when we leak an address from a library, such as the GNU C Library[1], we can find the addresses of all the other functions in it, even if the binary does not use them, allowing us to call any function the library has. Still, a lot of functions receive strings or pointers as arguments, which are simply indexes in the linear memory and have the type `i32`. Therefore, it may still be easy to choose a function to call with a crafted argument since we just need to choose one that matches these characteristics. According to [15], Emscripten being an SDK can place a lot of functions in a module, facilitating the task of an attacker by increasing the number of possible functions that can be called. According to [22], from the analyzed sample, 49,2% of the functions can be indirectly called and 48,1% can be reached by simply changing an index stored in the linear memory. Thus, the type-checking system is not very effective and consequently, indirect calls still pose a significant threat to the overall security of WebAssembly modules.

In [15] and [23], the authors show that besides being designed with the security of the sandbox environment in mind, WebAssembly still allows for vulnerabilities to exist in the binaries, due to problems in its specification. Also, [22] collected a large dataset of binaries from repositories, package managers, live and archived websites, and through manual search, to study the existence of vulnerabilities in the wild, using real-world programs. More than 50% of the analyzed code is believed to be generated from C/C++, a memory-unsafe language, so they are possibly the most vulnerable ones. 38.6% were shipped with a custom heap allocator, which increases the risk of shipping a vulnerable one, and 21.2% import potentially dangerous APIs from their host environment, allowing compromised programs to execute malicious actions. By looking at these numbers, we can see that there is an abundance of buggy code currently being shipped to production, which poses a serious risk to end-users.

[1] `https://www.gnu.org/software/libc`

3.2 Vulnerability Scanning Tools for WebAssembly Programs

Apart from Wasmati and WASP, we survey other tools that have been developed to detect vulnerabilities specifically for WebAssembly and to help developers avoid mistakes when writing their code.

Wasabi: Wasabi [24] is a framework developed for dynamic analysis of WebAssembly programs through binary instrumentation. It allows us to develop analysis functions in JavaScript that will be called by the instrumented binary through the usage of hooks which will receive these functions as a callback. It uses on-demand "monomorphization" to handle polymorphic instructions, in order to avoid wasting memory by storing a lot of code that is not going to be used. It is useful for call graph analysis, dynamic taint analysis, crypto miner detection, and memory access tracing, as well as studying branch and instruction coverage and profiling basic blocks and instructions. Yet, this tool still needs a human to program the analysis functions that are going to be executed. Besides having to modify the binary, the program does not change its semantics, *i.e.* the behavior of the instrumented program does not differ from the unmodified one.

Wassail: In [25], the authors' proposed Wassail, a static taint analysis tool that uses compositional analysis. It works by creating a call graph of the module and then identifying the Strongly Connected Components (SCCs) of the generated graph. It generates an information flow summary of a function based on how the information propagates from the function's parameters and global variables to the function return value and globals after the execution. Next, it analyses the SCCs in topological order so that when a function is analyzed it will either use a function that has already been fully analyzed or another function in the same SCC and, in the latter case, it is rescheduled for analysis. In the end, we obtain the resume of the module, and, according to the results in the paper, this tool has 64% of precision when using the PolyBenchC benchmark. The issue with this tool is that it will report a large number of false positives, which we can observe due to its precision on the PolyBenchC benchmark. So, it is not a very good option to determine whether or not vulnerabilities are present in code.

Fuzzm: Fuzzm [26] is a grey box fuzzer, based on Google's AFL fuzzer[2] that works directly with WebAssembly binaries. In order to identify stack overflows, it instruments every function in the program with code that inserts a canary, on function entry, onto the current stack frame in linear memory, and that checks its integrity upon exit. For the heap, a chunk is surrounded by canaries that are going to be checked when it gets deallocated, which may not be ideal since overflows can occur without being detected if a chunk is not freed. These canaries work as oracles in order to guide fuzzing toward exploring all paths in a program and finding crashing inputs. The program is also instrumented such that all return instructions are rewritten to a jump to the code that validates the canary. In [26], the authors also evaluated the effectiveness of the canaries in preventing exploitation and concluded that they harden existing programs and even prevent them from attacks of known exploits. The overhead of introducing the canaries in binaries is small, showing that they can secure binaries without affecting

[2] `https://github.com/google/AFL`

much of the original performance. The use of canaries to prevent buffer overflows in WebAssembly will imply an overhead that may affect performance-critical applications and an increase in memory usage. If there are many small buffers in the program, the canaries may occupy more space than the buffers themselves, increasing the memory footprint of the program.

3.3 Automatic Exploit Generation

Detecting bugs in programs is very important to prevent unwanted behavior in them, especially security-critical ones. There are many tools whose goal is to find them. However, a program can contain a huge number of bugs, and deciding which ones to fix first may be a very complicated issue.

In order to fix this issue, Automatic Exploit Generation (AEG) [14] was proposed in 2011 as a way to automatically detect bugs, determine which ones are exploitable, and then generate an exploit that spawns a shell. However, this type of automatic analysis is not currently available for WebAssembly, being mostly applied to x86 binaries. AEG analyses the source code of a program and uses KLEE [13] as the backend for symbolic execution, with some modifications that implement their techniques and heuristics in order to find exploitable bugs more quickly and pruning from the space state inputs that are not large enough to cause an overflow. After finding a bug that crashes the program, the binary is instrumented and stops when the vulnerable function is called so that the stack can be analyzed in order to obtain the location of the return address and the location of the buffer containing the shellcode generated by AEG. These values are then passed to STP[3], a constraint solver, in order to solve the exploit constraints and concretize the input if the given constraints are satisfiable.

Even before AEG, Automatic Patch-based Exploit Generation (APEG) [27] was proposed as a method to automatically generate exploits by receiving a program with a possibly unknown vulnerability and the version of the same program with the bug fixed. APEG was the first work to propose and demonstrate the feasibility of a combined static and dynamic analysis to generate formulas for a constraint solver, in this case, STP [28], to determine if an input is a candidate exploit or not. However, this tool assumes we already know where the vulnerability is located, as it requires an already patched version. On the positive side, it works directly with the binary which is something AEG [14] was not built for.

Following these pioneering works, multiple solutions that only work for x86 binaries emerged with the goal of improving what has already been proposed. We present and discuss some of them below:

MAYHEM: MAYHEM [29] is an extension of the AEG tool for binary code, a hybrid symbolic execution engine that combines offline execution (concolic execution) and online execution—forks an executor at each branching point—to tackle vulnerabilities. It uses a client-server architecture, where the client is a concrete executor running the binary and the server is a symbolic executor which manages the symbolic execution environment and which paths the client will execute. It can generate exploits for buffer overflow and format string vulnerabilities and uses Z3 to decide whether an exploit is satisfiable or not.

[3] https://github.com/stp/stp

CRAX: CRAX [30] is a framework designed to be used as a backend for static and dynamic program analysis. Based on obtained crash reports, it is able to generate exploits for large programs, which is a big improvement over other tools since scalability is a huge issue due to the number of paths that need to be explored. It works as a fuzzer to identify crashing inputs and then uses concolic execution to explore the path generated by them. This tool uses pseudo-symbolic memory in order to handle symbolic pointers, that can later be concretized when generating an exploit. It is able to generate exploits for format strings (in Linux only) and for heap and buffer overflows in both Linux and Windows systems.

ANGR: In [31], a binary analysis framework that gathers state-of-the-art techniques for binary analysis was proposed. ANGR is able to perform multiple types of dynamic and static analysis to binary, namely a modified version of Value-Set Analysis which performs better when applied to real-world binaries. In order to automatically generate exploits, it resorts to concolic execution on the found crashing inputs. Concolic execution stops where the program crashed in order to examine the symbolic state and measure exploitability. It is able to generate both ROP and shellcode exploits by overwriting the saved instruction pointer. In [32], ANGR is used as the symbolic engine to automatically generate exploits for buffer overflow vulnerabilities while also gathering information about the system, in order to bypass protections and harden the exploit, namely ASLR and non-executable stack. However, none of the herein surveyed solutions have been or can directly be applied to WebAssembly, which is our goal.

3.4 Web Crawling Systems

A web crawler, sometimes known as a web spider, is a computer software that searches the Internet by accessing websites. Crawling is primarily used to create an index of the websites visited by the tool.

Web scraping can be performed by web crawlers because it is well interconnected with the crawling activity since it consists in extracting data from the accessed sites. Web scraping is widely performed by search engines, like Google [33] or Bing [34] since they need to store the contents of the pages to fulfill the queries made by the users. Due to this, both activities are almost always done simultaneously.

Web crawling can be easily performed by utilizing tools like curl [35] or Wget [36], two very simple and popular tools that allow us to make HTTP/HTTPS requests. Starting with an initial list, we would be able to make all of the requests by creating a loop iterating through all of them and one of the previously mentioned tools to access the contents. This would be a naive approach but would do the job, however, it will only load one request and the other resources the website may reference like images, JavaScript, CSS, etc. While this may work for very simple analyses, a web browser-like tool would better suit the job of acquiring all the requests, like Selenium [37] or Puppeteer [38].

There are multiple challenges when crawling the Internet, so when implementing one of our own we need to detail:

- **Initial set of websites.** We need to define from which websites we are going to start the crawling. Ideally, we want websites that reference a lot of other websites so we can broaden our search more quickly and easily.

- **Selection policy.** From all the websites gathered we need to decide which ones we are going to select and which ones are going to be discarded.
- **Paralellization strategy.** A strategy needs to be outlined so we can maximize the analysis rate without repeating websites, so we also need to establish a policy that will prevent the same website from being accessed by different workers.
- **Deepness of the crawl.** We need to establish a limit on how many websites, counting from the root (the one from the initial set), we want to access. This is an important parameter to establish since if we go too deep, we are most certainly accessing a repeated website.

We now present some of the current web crawlers, their architectures, and their limitations.

curl: Curl [35] is a command line tool for transferring data. It supports multiple protocols, like HTTP, HTTPS, FTP, GOPHER, and SMTP. This tool allows a user to override the User-Agent parameter it uses so that websites that block the default one think the request was made by a browser, for example. While being very simple, although complete, the data it can retrieve during crawling is also very basic since it treats every request as static, meaning, for example, that if a page has JavaScript that performs any kind of request, those will not be processed, as it cannot process the obtained data, it only returns it as is.

Wget: This command line tool is similar to Curl [35], but by default it saves the data on a file, instead of printing it to the standard output. The protocols supported by Wget [36] are much less than the ones supported by Curl, only dealing with HTTP, HTTPS, FTP, and FTPS. It has the advantage of being able to resume the download of pages that may have failed, and recursively remote downloading directories, enabling us to crawl faster, without needing to make the requests for the files individually. Like Curl, Wget also treats the content as static, so it is unable to process requests made by JavaScript, or even all the contents an HTML page has, like its styles files or images.

Selenium: With Selenium [37], we are able to automate web crawling using a browser since it supports using all the major web browsers. Selenium is a browser automation framework that exposes an interface that can handle all browsers in a generic manner. Specific features and configurations of each browser can also be used, however, using them makes a solution less portable when changing testing with other web browsers. Since it uses a browser, it is able to handle all the requests made by JavaScript and load all the contents of the web page. This tool has an API available for a big number of languages, allowing it to be embraced by many other projects without much effort. It provides an interface to execute JavaScript commands and also to interact with the DOM and alert boxes. One limitation of Selenium is not allowing to intercept requests and responses, so if we want to analyze the requests on the fly and modify them, it cannot be done without modifications on the source code (this is addressed by Selenium Wire[4], but is limited to Python). Selenium also does not detect some of the requests made by JS code.

[4] https://github.com/wkeeling/selenium-wire

Puppeteer: In [38], Puppeteer is presented as a Node [39] library that is capable of controlling Chrome and Chromium browsers. Being a Node library it is developed in JavaScript so its official API is limited to that language. It is similar to Selenium because it automates browser interaction however, Puppeteer specializes in the features of Chrome and Chromium. This tool can generate screenshots and PDFs of pages, enabling us to easily check the current status of a webpage. It is also suitable for UI testing, automatically and it allows us to intercept requests and responses, contrary to Selenium.

OpenWPM: In [40], the authors propose a crawling framework, OpenWPM, which was developed to study 1 million websites in order to identify online tracking (either stateful or stateless) sites, among other privacy issues on the Internet. It supports the running of multiple browser instances providing higher speed than single browser solutions, however, has a more complex architecture. This tool is composed of three pieces: a task manager which distributes commands to the browser managers; browser managers which automate web browsers through Selenium [37]; and finally the data aggregator which instruments the requests of the browser. This is a very promising approach since it uses a leader process to distribute the work to the working process. However, OpenWPM is not able to intercept requests and responses due to the lack of this functionality in Selenium, not allowing it to selectively collect the necessary data or perform analysis in a distributed manner.

3.5 Web Datasets and Related Studies

Through the usage of web crawling techniques and tools like the ones we presented in Section 3.4, many datasets have emerged with multiple goals in mind. Also, many studies have been performed on the web and on the datasets generated by crawlers. Although these datasets may exist in big numbers, some are incomplete, do not represent the web nowadays, or are not suitable for all types of studies.

We now present some web page datasets and studies performed by the scientific community, as well as some of their limitations:

Minified and Obfuscated Code: In [41], the authors analyze minified and obfuscated JavaScript code. This code, due to the processes that have passed, is very hard to read by humans, so it is easy to hide malicious programs this way. It contained scripts from the top 1 hundred thousand websites of the Majestic Million service[5]. Since this study was conducted on obfuscated code, its dataset is only composed of JavaScript, not being a good sample to evaluate WebAssembly prevalence nowadays, even though it was performed in 2019, when it was already popular. Also, by only performing an analysis on the most accessed websites, it cannot obtain results that can be transposed to the whole web, which contains many diverse pages with diverse content.

OpenWPM: The authors of OpenWPM [40] developed their tool to measure online tracking on the top 1 million websites of Alexa[6]. Since this study was performed in January 2016, WebAssembly was not

[5] https://de.majestic.com/reports/majestic-million
[6] https://alexa.com

even a minimum viable product, so it was not present on the Web or was in a very small amount, due to it only starting to gain traction in the last years. However, with the rise of Wasm, we can see more diverse ways of fingerprinting and tracking. The data obtained from the authors only refers to the top 1 million websites, which are not representative of the web because those are the most accessed ones. Scam websites, lesser-known blogs or news pages may a different number of trackers and certainly different features from the ones provided here, so we can have only a glimpse of what the bigger ones have.

WasmBench: In [22], the authors present us with a dataset more focused on WebAssembly, in order to evaluate the usage of these binaries, their source languages and to test them for vulnerabilities. They have gathered them from multiple sources: crawling, source code repositories, package managers, and live websites (manually). Despite the use of crawling taking as a starting point the Tranco top websites' list [42], it contains pages from the Web Archive [43] and from GitHub source code repositories, making it not suitable to evaluate the state-of-the-art Internet regarding WebAssembly. This dataset is composed of only Wasm binaries so analyses on HTML or JavaScript are not possible, only allowing analysis on this type of file.

Summary

In this chapter, we took a look at vulnerability scanners that have been presented for the WebAssembly language. Then, we saw some tools that are capable of automatically generating exploits for detected vulnerabilities, but that are only present for other languages. Next, we saw some of the existing web crawlers and their respective limitations. Finally, we looked at some of the datasets that have been presented recently and why they were not a good fit for our study.

4

Framework Design

Contents

This chapter presents our approach to generating a dataset of web pages that will later be analyzed. To accomplish this, we developed our analysis framework named WebAF. We start by giving an overview of the framework, its division, and how its parts interact. Lastly, we present each of its parts and the challenges of their implementation.

4.1 Framework Overview

WebAF is the framework that we developed in order to generate and analyze a part of the web, seeking for websites featuring WebAssembly code and analyzing them. Our framework is composed of two main parts: a **crawler** and a **scraper**, each with a different role. The crawler will simply access the websites and store information about them, e.g., if they contain WebAssembly or make use of the WebAssembly JavaScript keyword. On the other hand, the task of retrieving the actual contents is left to the scraper, as well as the task of analyzing the web page contents.

In terms of their respective architectures, the crawler and the scraper are very independent of each other. The only point they have in common is the PostgreSQL database which is where the scraper obtains its data to process – the websites signaled by the crawler. The crawler functions in a microservices architecture, allowing it to scan a wider range of websites faster. We decided that the scraper would not need such a complicated architecture since it would scan only a smaller subset. The crawler and the scraper can be run in parallel, but we need to keep feeding the new data to the scraper because the crawler stores data in PostgreSQL, and the scraper queries MongoDB. Also, if following this approach, we need to be careful not to export the same data twice from PostgreSQL, because this will affect the results we will obtain in the end. Next, we present these two components in detail.

4.2 Crawler

We start by presenting the architecture of our crawler, taking a deep look at how each part functions inside it. We then proceed with the challenges we faced when implementing it and executing it in our own infrastructure.

4.2.1 Overview

Our web crawler uses Puppeteer [38] at its core and navigates through the web obtaining and making a quick analysis of the received data. As we can see from Figure 4.1, our tool is loaded with the data provided by Amazon Alexa (which has recently been shut down by Amazon [44], however, its API is still available [45]) – the websites ranking based on traffic. It contained the top 1 million most accessed websites on the Internet, so it would be a good source for analyzing patterns in them. Those websites may be following the most recent trends to keep their users engaged and achieve better performance.

This list is then passed to the **Scheduler** component, which selects a batch to load to a cache database (**Redis** [46]). By default, this value is set to 1000 websites. The scheduler then retrieves an

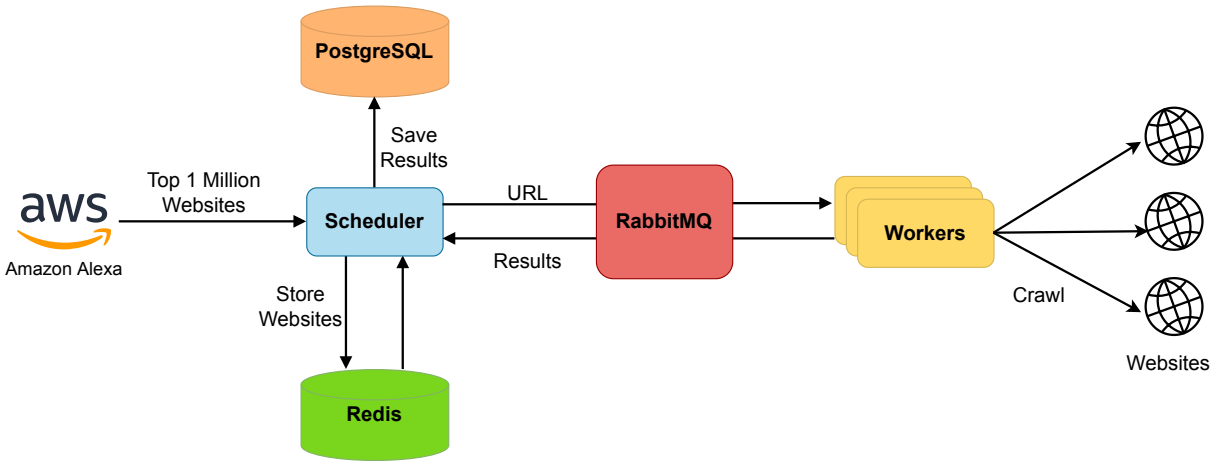


Figure 4.1: Crawler high-level architecture

URL from the cache and puts it into a RabbitMQ [47] queue to be then consumed and accessed by a **Worker** node. If the number of inserted URLs in the queue is different from the number of worker nodes, it will retrieve more until this condition is met. Then, it will wait until a response is emitted from one worker. When this happens, it parses the response from the worker and stores it into **Postgres** [48], our persistent storage database. Lastly, it proceeds to get another URL from Redis and the cycle repeats itself until the batch reaches its end, in which case it will load the next batch, or there are no more links and the scheduler waits for the remaining responses and exits.

Figure 4.2 takes a deeper look at the details of our crawling utility. We can see that there are only two queues present in RabbitMQ: one to fill with URLs and the other to fill with the crawling results. These queues are shared among all workers. Each worker node is composed of two main parts:

- **Puppeteer [38].** The Google Chrome [49] web browser controller. It will simply access the websites the worker node commands it to.
- **Mitmproxy [50].** The proxy that is going to be used by Puppeteer to perform all requests. Mitmproxy is the ideal tool for this job since it allows us to register all the requests made by the Chrome browser. Since it supports plugins, it also enables us to create our own and perform all the operations we want in both the requests and the responses, after and before they leave the proxy.

By default, the worker node uses two plugins: one to signal WebAssembly and another to signal SharedArrayBuffer websites. Both plugins have implementations for the proxy (in Python) and for Puppeteer (in JavaScript). The latter needs plugins to perform two different tasks: to override implementations and to implement the `onCallback`, `onSuccess` and `onError` callback operations.

When a worker node is started, it first bootstraps Mitmproxy with the available plugins and then starts the Puppeteer component, also loading the plugins accessible. Then, it waits for an URL to be available so it can start working. As soon as one becomes available, the crawler accesses it, and when a response is emitted by the server, the proxy calls the plugins in order to perform the needed actions on the page. Then, the Puppeteer modules come into place to override anything that may be needed – in our case, the WebAssembly plugin overrides attempts to create or run a WebAssembly module. Only after these

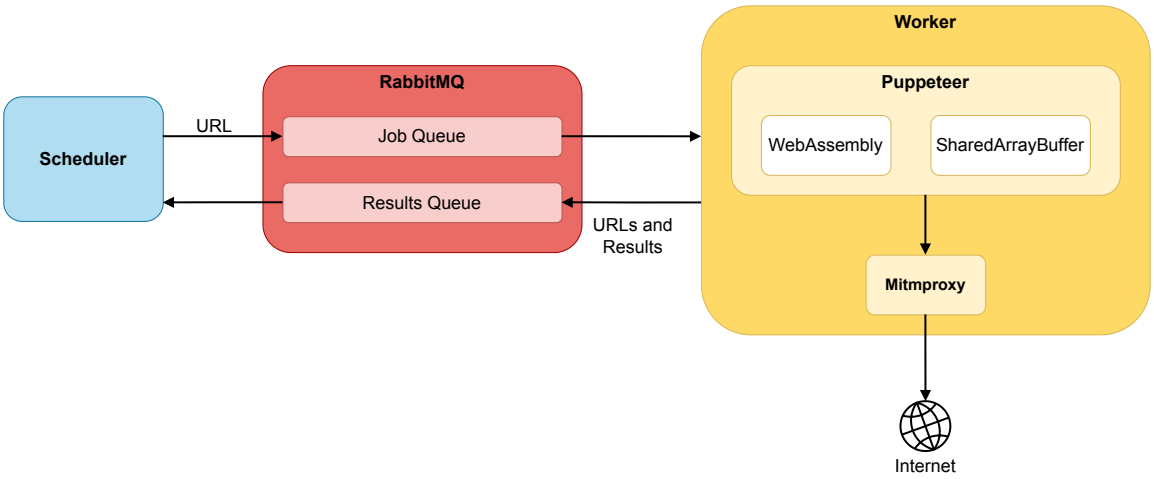


Figure 4.2: Low-level architecture of the worker

steps the response reaches the browser and is then executed.

After getting to the browser, we collect all the anchors that are present in the website and randomly select a subset to be analyzed next (the number of elements of this subset is four by default but can be altered by changing an environment variable). The worker node then proceeds to queue the selected subset on Puppeteer, to later be accessed. This process continues until we reach the maximum depth from the original website (by default the maximum depth is one website, but can also be changed) in which case we will not select more URLs and proceed to crawl URLs that are on the same level or on the levels above. In the end, the worker generates a response with all the URLs that served as an entry point for all the others, with the accessed URLs, and with the results signaled by each of the plugins in case they have made any work. This response is put into the result queue and is then handled by the Scheduler. Finally, we remove the URL from the job queue to prevent other nodes from performing repeated work. If a node does not perform this operation, after a while the URL will reappear in the queue to be processed, indicating that a worker node has failed and its work was lost.

4.2.2 Implementation

Given the microservices architecture, our web crawler is an excellent candidate to be scaled in multiple machines, like in a Kubernetes (K8s) [51] cluster. Using this technology, we can launch containers at will without concern about which machine should they be deployed to and how they will communicate with other containers that may be placed in different physical devices after the cluster has already been set up. Since we had our own infrastructure with multiple powerful machines, we wanted to host a Kubernetes cluster there, so we could be able to run our tool.

In order to run our tool in our private infrastructure, since only SSH access is provided and there is only a small subset of commands available, we first need to deploy virtual machines (VMs) on the machines and install the tools needed there. The hypervisor available is VirtualBox [52], and Vagrant [53] and Ansible [54] are in each of the machines. We decided to set up the Kubernetes cluster using these two tools as they work perfectly together. Ansible allows us to easily install packages with always the

same version by writing the configuration once and running it multiple times, creating a reproducible environment, meaning all VMs can have the same packages and the same configurations. Vagrant empowers us to deploy multiple virtual machines with the desired specifications and networking configurations. Vagrant also has a plugin for integration with Ansible which revealed itself quite handy.

The first step was to deploy virtual instances on the physical machines, so we started to build our Vagrant config with one virtual machine for each host, with all the CPUs of the host except four and with all of its RAM except 4GB. Our image of choice was Ubuntu 21.10 [55] due to being a recent version, having good container support, and being a Long Term Support (LTS), meaning it will receive support and security fixes for a longer period of time than non-LTS versions. Since our infrastructure only allows VirtualBox as the hypervisor, we quickly reached one of its limitations: "Oracle VM VirtualBox (...) can present up to 32 virtual CPU cores to each virtual machine" [56]. To overcome this issue, we split the previous amount into two VMs and it worked as intended.

The next step was to then deploy a Kubernetes cluster using these machines. A Kubernetes cluster has two types of machines:

- **Control plane.** Instance which controls the whole cluster, namely, scheduling of the containers, detecting and responding to events such as new deployments being made or resources being deallocated, among others [57].
- **Node**[1]**.** Instance where the containers are run and report events to the control plane. This type of instance runs on every machine of a cluster meaning a device in the cluster can both be a control plane and a worker node.

Since these two types of nodes have different jobs and can be run separately, we decided to give more resources to the worker node as it will use more computing power than the control node and not use the latter also as a worker node. All we needed was to install the appropriate software on the machines and start the cluster. To make it easier, we instructed Vagrant to create the Ansible inventory[2] for us with 2 groups: master and worker.

In addition, because we are using our own infrastructure – which is not a private cloud – and the need to deploy a production-grade cluster, we had to set up our cluster using `kubeadm` [58], a tool that facilitates the bootstrapping of a Kubernetes cluster. Kubeadm requires the machine to have:

- 2 CPUs or more
- 2 GB of RAM or more
- swap disabled
- a container runtime
- a networking solution

[1] To avoid ambiguity, this type of node will be called worker node from now on
[2] The IPs, SSH keys, and other attributes of the hosts we are provisioning

For the container runtime, we decided to go with Docker [59] because it is the most widely used platform for running containers both for users and companies [60] and is easy to set up and install. After installing Docker, all that was needed to do was to install `kubeadm` [58], `kubelet` [61] and `kubectl` [62].

Kubelet is the Kubernetes agent that runs in each node, registers the machine to the control plane, and launches and ensures the healthiness of the pods[3] within the machine. Kubectl is a command line tool that allows us to communicate with the cluster to create new resources or to update or delete existing ones. Lastly, we were missing the networking solution which is needed for the nodes and pods to communicate with each other (and consequently pods and containers) and to be accessible from the outside. We went ahead with Calico [63] because it has high performance, is open-source, and is already very mature - suitable for production.

At this point, all the requirements are met to proceed with the installation. After initializing the controller node, we must issue a command on the other nodes we want to join to the cluster. In order for this to work, the machines need to communicate with each other, so in the Vagrantfile we need to create a new interface for each one of them because by default they are behind a NAT and do not communicate.

We had our small cluster deployed in one virtual machine, so we needed to make a quick test on the crawler. It was deployed using our own Docker images and some Helm [64] charts. Helm charts are basically packages that we can easily install in our cluster, like containers that are very easy to configure. For the external technologies used by the crawler, we relied on Bitnami's [65] Helm charts as they are very well documented besides having some modifications over the original images. For our docker images, we needed to have a private docker registry and to accomplish this, we decided to simply host a docker registry as a container on the master node. After setting it up and pushing the images there, we were able to deploy the crawler on the Kubernetes cluster and make a quick run to check everything worked properly.

Since one of our physical machines does not have enough power to create a big cluster, despite having very high-end hardware resources, we needed to expand the computing network to multiple physical devices. Still, a problem arises: **how can we connect multiple virtual machines that are in different physical machines?**

Our first approach to tackle this issue was to use a lightweight Virtual Private Network (VPN) such as Wireguard [66]. We chose this VPN since it has been incorporated in the Linux kernel [67], is very performant as it uses UDP as the transport protocol, and is very simple to configure as we only need to provide peers' IP addresses, public and private keys and which IP ranges are routed through that network. In our solution, we would create a port-forwarding for the master node, so the other nodes (in the other physical servers) could reach it through the physical IP. This approach worked well and allowed us to scale the crawler. However, there were some issues regarding reliability and network bandwidth:

- Some virtual machines stayed up but their pods stopped responding to their liveness probes[4], so the pods would not be rescheduled to another node since the node appears as active. The **virtual machines got corrupt** and we were not able to access them through SSH. **If this would happen**

[3] Smallest deployment unit of Kubernetes. Can be composed of one or more containers
[4] Request made by the control plane to check if the pod can receive traffic, if it's in a healthy state

to our only master node, we could lose our cluster, since the control plane would be down and there would be no way to recover it, because all the cluster state is stored in the local etcd [68] instance, which is only replicated to other control pane nodes (which we do not have). Etcd is a distributed key-value store that can be deployed externally to the cluster, but for simplicity, we will use etcd co-located with control plane nodes.

- The network bandwidth of the master node was reaching its limit as many nodes were trying to download the crawler images from the registry and at the same time, the control plane was attempting to scale and obtain the state of the pods being deployed. **This approach created a huge bottleneck in the network** resulting in many pod restarts, mainly on the virtual machine running on the same hardware as the master node, due to not being able to access the internet.

To address this problem and to improve the reliability of our cluster, we decided to deploy three Kubernetes master nodes, each one positioned on a different physical machine. This way, our tool would be able to work properly even in the event of failures. However, another problem emerged, because we could not deploy the cluster with multiple master nodes using Wireguard. The VPN would generate multiple network routes for the same IP address, leaving the machine without any contact with the outside world. We were looking for a solution where there is no relay on the network and Wireguard does not support this using the current setup. In order for multiple machines to communicate directly, they would need to know their IP beforehand. So, we tried to deploy Nebula [69], a peer-to-peer network initially developed internally for Slack[5]. This software-defined network uses a Public Key Infrastructure (PKI), with a custom Certificate Authority, to issue certificates for the hosts in it. Each private key certificate contains a name for the host, its IP in the network, its public key, and the CA's signature.

Nebula distinguishes between two types of nodes in its network:

- **Lighthouses.** A lighthouse is one of the heads of the network. It has a listener in a predefined port and its IP (on the nebula network) is known by the other peers. It can be seen as the router as it knows how to communicate with the peers in the network, however, packets do not flow through it to interconnect with multiple peers.

- **Hosts.** A host node is a regular node and it has static routes to the lighthouses. If host A wants to communicate with host B, it will first ask one of the available lighthouses for the route and then establish a connection to the obtained IP.

In our case, the lighthouses would correspond to the Kubernetes master nodes, which we considered a good idea since the lighthouses do not require many computing resources. Besides the simplicity of configuration, we were not able to connect two peers using two lighthouses. We believe this is due to some issues in the tool, and the traffic is not able to cross the NATs that our infrastructure has. A packet can leave the machine, get into the other machine, then a response leaves the latter but is never able to reach the former.

[5] `https://slack.com`

Finally, we decided to abandon this tool which showed itself inviable for our use case. The solution we found was to use the bridged mode [70] for the virtual machines, which assigns one IP to each virtual machine as if it was a physical node in the network. With this approach we would not have problems connecting all of our virtual machines, however, there is the risk of running out of IP addresses since our physical machines have public IP addresses, so our virtual ones would also have them. This was our final network setup: one IP for VM, in the same network as all the physical machines[6].

After solving all network-related issues, we still needed to manage how a cluster with multiple masters was set up. After reading the Kubernetes documentation [71], which mentioned the use of a load balancer, we chose to deploy a virtual machine that would only serve this purpose. We decided to go with HA Proxy [72] since it is very mature, is open-source, and has an excellent performance. Also, with a load balancer, we could be able to easily check if a master node is down and take action to bring it back up and running.

To sum up, after several attempts, our final deployment was composed of three Kubernetes masters, twenty Kubernetes worker nodes, and one load balancer, corresponding to a virtual machine each with a public IP. The VM deployment was configured with Vagrant and their provisioning with Ansible, using roles. The number of virtual machines and other parameters can be configured to adjust to the available devices. After the machines were ready, we just deployed our application in the cluster with a hundred crawler worker pods and started crawling the web.

4.3 Scraper

Now we will present the scraper we used to collect the data that was marked by our crawler, and discuss some of the challenges we faced during its implementation. We decided to not fetch the entire content of a website directly in the crawler because this way we would not have to transfer a lot of data to the Scheduler, enabling us to move faster and to store less information on an initial phase. So, we need to access the same website twice, but we can analyse more websites, more quickly.

4.3.1 Overview

Our scraper is a very simple tool that will work with the results of the crawler. It is developed in Python 3 [73] and uses MongoDB [74] as its storage. In Figure 4.3, we can see the complete architecture of our tool, illustrating the following components:

- **PostgreSQL.** Part of the crawler that is simply used to import some of its contents into MongoDB.
- **MongoDB.** Stores both the URLs that are going to be analyzed and the results of our analysis.
- **Scraper.** Core of the tool as it contains all the logic.

The data gets exported from Postgres using `psql` [75] with the `ROW_TO_JSON` function, present in this Database Management System, and is imported into MongoDB through `mongoimport` [76].

[6] We do not need to protect these machines with a firewall as they can only be accessed from inside our private VPN

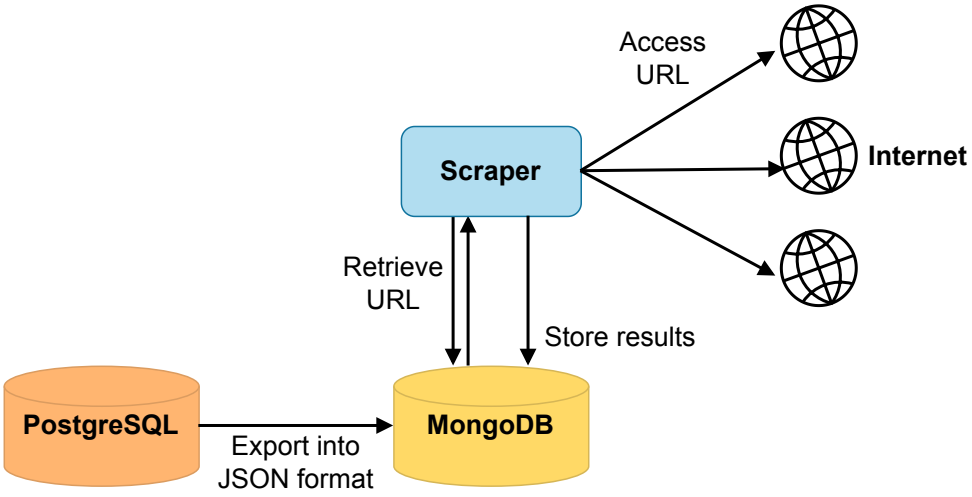


Figure 4.3: Scraper high-level architecture.

The scraper component can also be decomposed in the items we can see in Figure 4.4:

- **Controller.** Controls which websites to visit and passes the results to the Analyzer.
- **Selenium [37].** The browser controller which accesses the websites.
- **Mitmproxy [50].** Serves the same function as it did in the crawler: obtain all the requests and responses, allowing us to analyze them through the use of plugins.
- **Filter.** Is a Mitmproxy plugin that analyzes the responses obtained by the scraper.
- **Analyzer.** Perform an analysis of HTML, JavaScript, and WebAssembly (described in Chapter 5).

The scraper is initialized with multiple threads (the number can be configured on startup) and each one of them will scan a different website. The controller will start a Mitmproxy instance with the Filter plugin and set up its Selenium browser instance to make requests through the proxy. Then, it will contact the database, get one link to visit, and load it into the browser, to access it. When the responses are obtained, they are passed to the Analyzer, which will filter and accept only valid HTML, JavaScript, and WebAssembly. These responses are stored, hashed to only save repeated responses once, and then passed to the Selenium browser so that it can process the page and make any new requests.

When all is done, the Controller continues its job and forwards the findings to the Analyzer, which will perform a series of static analyses on the obtained files. Finally, both the responses and the results of the analyses are sent to the database where they are going to be stored and later processed.

All accesses to the database are mediated by a wrapper that we defined and they are synchronized, in order to avoid repeating saved requests. This wrapper is shared among all threads, and this is how we can guarantee that each thread gets a different URL to analyze.

4.3.2 Implementation

We decided to simplify this part of our framework since it will only visit a subset of the URLs the crawler has accessed. The scraper runs on a single machine and can take advantage of multiple cores.

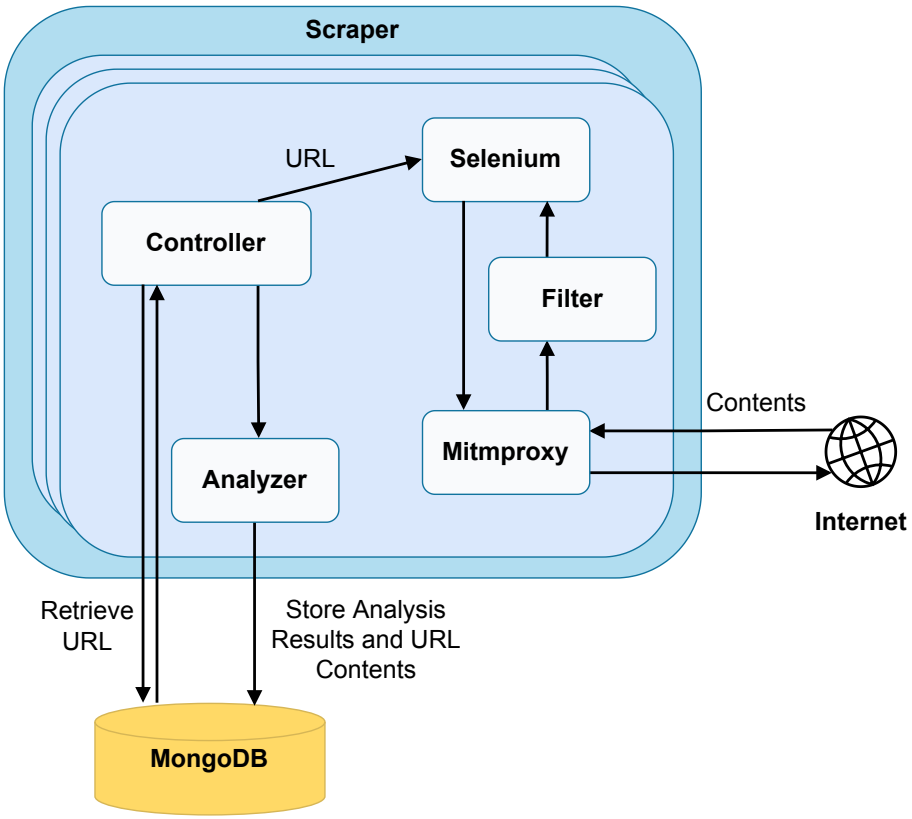


Figure 4.4: Scraper low-level architecture

The biggest challenge in the implementation of this tool was whether to first obtain all the results and then analyze them, or analyze them first and then store them. The first approach allowed us to scan more websites more quickly but when it came to analyzing their contents, it became too slow, possibly due to retrieving big responses from MongoDB, from multiple collections. The whole process of retrieving the contents to analyze and then storing the results was just too slow, so we decided to put the analysis step in between accessing the website and storing its contents - the second approach. This showed itself the best way to tackle analysis since we already had the contents in memory and did not need to fetch them again from the database. It takes a little more to access the same amount of websites as the first approach but we can obtain the results right away and the overall progress is not that slow in comparison.

When crawling the websites, we may have repeated results either because the crawler signaled the same website two times or because there are resources that are referenced by multiple websites. To mitigate this issue and avoid saving the same file multiple times, we use two hashes of the file as its key. The key has the format `MD5(content)_SHA1(content)`. We decided on these two hashes because they are weak since there are already many collisions found [77, 78], however, they are very easy to calculate and by mixing the two hashes, we are more confident that if a file matches the two hashes it has a low-probability of being different because it needs to match in the two weak hashes but that are calculated in different ways. When a response is saved, it checks if there is already an entry for it and if so we only add an entry for the link that referenced the resource.

One of the other challenges we had to overcome when implementing the scrapper was how to handle

the failures of Mitmproxy, because it fails silently, meaning that the Selenium browser would not receive an error, just an empty response for every request. The way we handled this was to kill the current Mitmproxy instance, close the browser instance and start the proxy in a new port. This would only affect one of the threads, leaving the others intact, since each one has a different instance of the proxy.

Finally, to handle failures from a thread (in case an exception occurred and we could not finish the request), we did not want its website to be forgotten and its results missed, so we implemented a cache in the system with the Least-Recently Used policy. The cache has a number of entries equal to the number of threads running. The cache receives a new URL in case it is not already full, otherwise, it means a thread has restarted and requested a new website, so it pops the link that was previously interrupted. When a controller wants to access a website, it queries the cache giving it the previous URL that it scanned. If this URL is present in the cache, it is removed and we also remove it from the collection in MongoDB. Next, we check if the number of items in the cache is equal to its capacity and if not and a new one is fetched from MongoDB. Using this approach, we guarantee that a website is only removed after being processed and its results saved. This cache is invisible to the controller and is handled in a wrapper to the database. It is a synchronization point, meaning only one thread at a time is able to request a website, allowing us not to repeat the same work multiple times.

Summary

This chapter has described the implementation of our framework. It addressed the architecture of our web crawler and web scraper. The web crawler was composed of several microservices, the scraper was a multithreaded program, and the point of communication lay in the PostgreSQL database. The next chapter presents the dataset collected by the framework and the analyses performed.

5

Dataset Generation and Analysis

Contents

This chapter presents the methods that WebAF uses to be able to analyze the data it gathered from the web pages it connected to. We begin by giving an overview of our techniques and explaining in more detail what were the conditions when the dataset was generated (Section 5.1). Next, we explain in detail each one of the performed analysis techniques - a generic analysis (Section 5.1), HTML (Section 5.2), WebAssembly (Section 5.3) and JavaScript (Section 5.4). Lastly, we present our evaluation of the interaction between JavaScript code and WebAssembly at the end of Section 5.4.

5.1 Collected Dataset and Generic Analysis

The scraper described in Section 4.3 performs multiple analyses on the obtained data. The type of analysis depends on the type of content retrieved from the web. Since we are only dealing with code, the generated dataset contains only HTML, WebAssembly, and JavaScript. To the best of our knowledge, this is the first study that investigates the relationship between JavaScript and WebAssembly only focusing on real web pages. This also constitutes the first study to evaluate the prevalence of WebAssembly in the wild. With this evaluation, we want to better characterize the web nowadays, regarding the usage of WebAssembly, its interaction with JavaScript, and how it is loaded, either directly from JavaScript or from HTML.

Since WebAF is separated into two independent parts, we chose to run the crawler and the scraper at two different times, because the available resources on our infrastructure are limited and due to time limitations, since the resources are shared with other researchers and need to be allocated for a well-defined period of time. The crawler started running on the 17th of May and we needed to stop it on the 19th of the same month. During this period of time, it was able to access 73590 websites from the Alexa list, which gave us a good enough dataset. The total number of accessed websites it was able to access in that time interval was 292514. From there, we only wanted to obtain websites that were not repeated, so needed to filter them, and ended up with 289473 websites, representing 98.97% of the initial set. The scraper started its job on the 22nd of September and we stopped it on the 11th of October with the filtered number of websites. During this time it was capable of obtaining the contents of 116101 websites, representing 39.69% of the fed dataset. We believe this is a good dataset besides not being very extensive since it has websites that are one layer deep, coming from an Alexa-rated one, so the dataset may contain websites that are not popular.

Before performing analysis on the specific resources of a web page, we wanted to get some generic insights on the obtained dataset. For that, we chose to study the size of whole websites (excluding images and other resources that were not relevant to our study), as it allows us to perceive how big (or small) the web is on average. It also enables us to evaluate the percentage of each type of resource. In Figure 5.1, we can see that 50% of the webpages we visited are around 2MiB or less in size, which is a big number, since we are only measuring code, so the pages may take a little while to load. It also shows us that there are only a small amount of websites that are very small.

We also evaluated the prevalence of each resource (HTML, JavaScript, and WebAssembly) on each website, and obtained both their percentage relative to the size of each content (Figure 5.2a) and relative

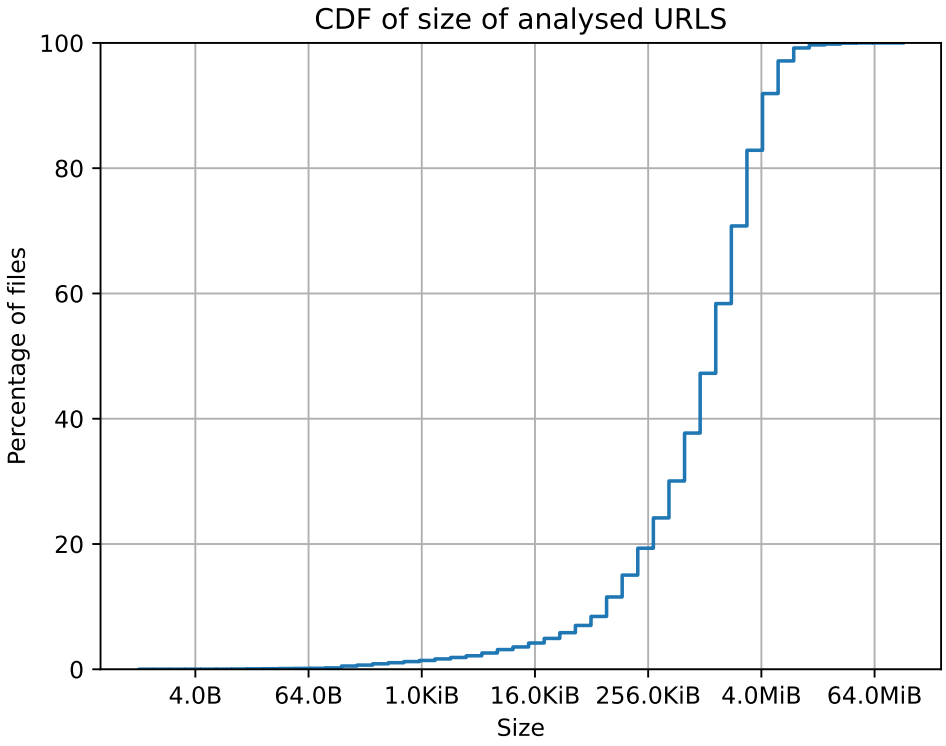


Figure 5.1: Distribution of sizes of the analyzed websites.

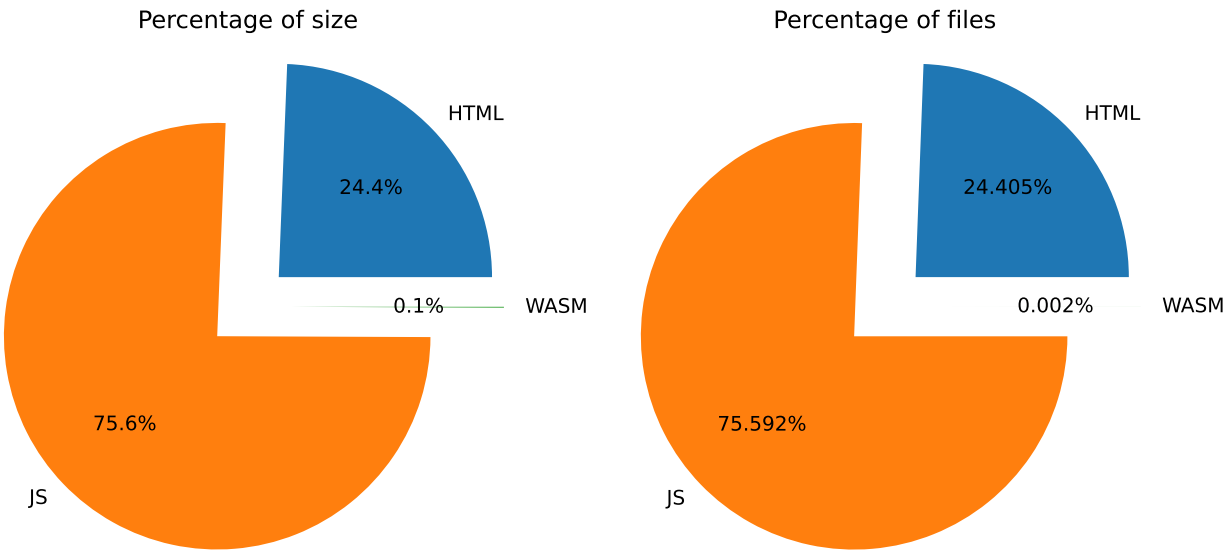
Minimum size	0.0B
Average size	2.3MiB
Maximum size	85.8MiB
Percentile 25	444.4KiB
Percentile 50	1.4MiB
Percentile 75	3.2MiB
Total size	261.8GiB
Total URLs	116101

Table 5.1: General statistics of the dataset.

to the number of files found (Figure 5.2b). We can see that besides being very popular in the last years, Wasm only holds a very small percentage both in size (possibly due to being a binary format, so very compact) and also in the number of files. JavaScript holds the position of the most used resource on the web, allowing us to infer that most of the modern web is driven by this powerful language since there are more JavaScript scripts than HTML content. Most of the content nowadays is loaded through JavaScript, not being present in HTML statically.

We can see that we were able to obtain data from a total of 116101 websites, making a total of 261.8GiB[1]. The minimum size of a website we obtained was 0.0B, meaning we did not obtain any data from them, either due to encoding errors processing the returned data or the website did not respond, although it accepted our connection. Taking a look at Figure 5.1 in conjunction with percentile 25 from Table 5.1, we observe that there are a lot of websites with very low size, meaning those websites were signaled by the crawler so they have been active in the past. However, since the time of crawling was

[1]This is the total amount of data we would store if we did not deduplicate data.



(a) Percentage in size.

(b) Percentage in files.

Figure 5.2: Percentage of each resource in terms of number of files (left) and size (right)

different from the time of scraping, some of them might not be available at the time of scraping due to Internet dynamism. We can see that half of the dataset is 1.4MiB in size, which we consider an amount similar to what we could find in the real world. Due to the time a website may take to load, developers want to ensure it does not take too much time but also that it transmits all the needed information for it to work properly, this is considered a reasonable value by us. Percentiles 50 and 75 are not too far apart from each other, meaning that a big range of websites falls into these two values. Given the maximum value of 85.8MiB, we can see it is too far from 3.2MiB, as we would expect since it would import an enormous amount of code and take many seconds (or even minutes) to finish a request, making a user give up on the website quickly. Next, we present our analysis of the websites' subcomponents.

5.2 HTML Analysis

We decided to analyze HTML code since it is what loads the resources of a web page. It allows for `script` tags to contain either a link to another remote resource or inline JavaScript. Since we are analyzing WebAssembly and because it can be embedded in JavaScript code because `WebAssembly.compile` requires a typed array or an `ArrayBuffer` [79] so we can put encoded raw bytes inside JavaScript, HTML pages may therefore contain WebAssembly code embedded. As a result, for our HTML analysis, we adopted the following metrics:

- Contains embedded JavaScript
- Contains embedded WebAssembly
- Size of the website HTML files

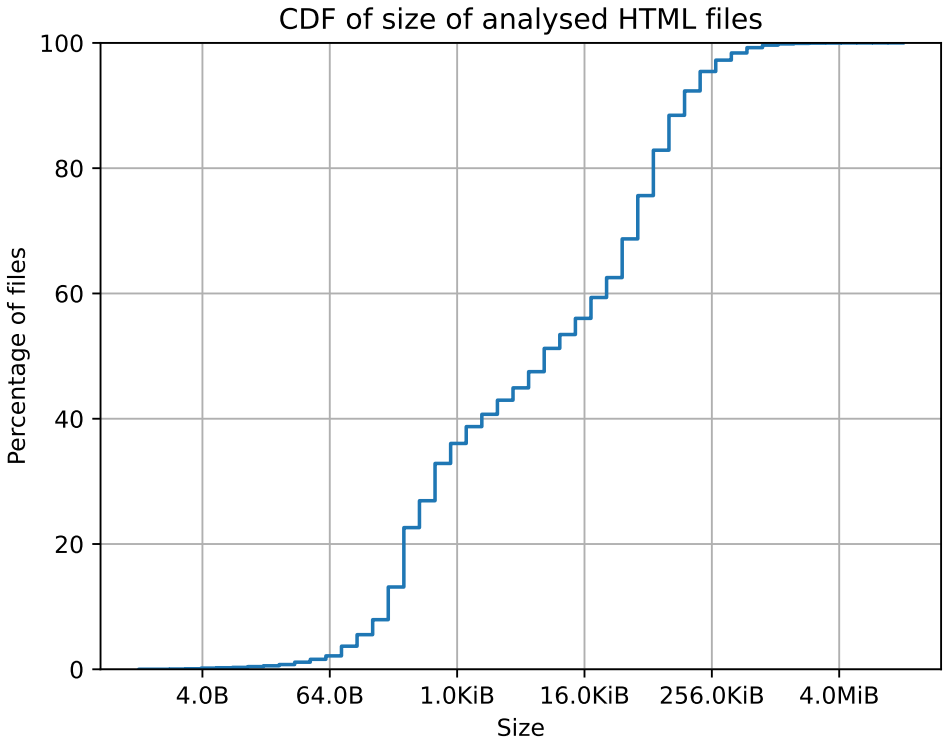


Figure 5.3: Distribution of size of HTML pages.

Minimum size	0.0B
Average size	63.8KiB
Maximum size	13.6MiB
Percentile 25	531.0B
Percentile 50	8.1KiB
Percentile 75	69.4KiB
Total size	25.2GiB
Total unique files	415080
Files with embedded JavaScript	0
Files with embedded WebAssembly	0

Table 5.2: HTML statistics of the dataset.

Beginning with the size of HTML files, we plotted the CDF in Figure 5.3. We see that 60% of the HTML files are around 70KiB or lower, meaning they do not have a lot of content, as we stated before, due to it being dynamically added by JavaScript (or even WebAssembly). We can also see one of the biggest spikes sits between 64B and 1KiB, showing that a big amount of sites fall under this range, possibly serving only as a redirect to others since they are too small to contain any relevant content or to load it through other ways.

As mentioned above, we also wanted to check if there is embedded JavaScript inside HTML files and also if there is WebAssembly inside this JavaScript. By taking a look at Figure 5.3, we can already see that these types of embeddings are scarce, as they would increase the total size of an HTML file. Table 5.2 shows our findings in these areas. We can see that there are no embeds with JavaScript so it is impossible to have WebAssembly embeds. We were expecting to find some websites doing this, but not all, because this is an old way of doing things and does not take advantage of loading multiple files

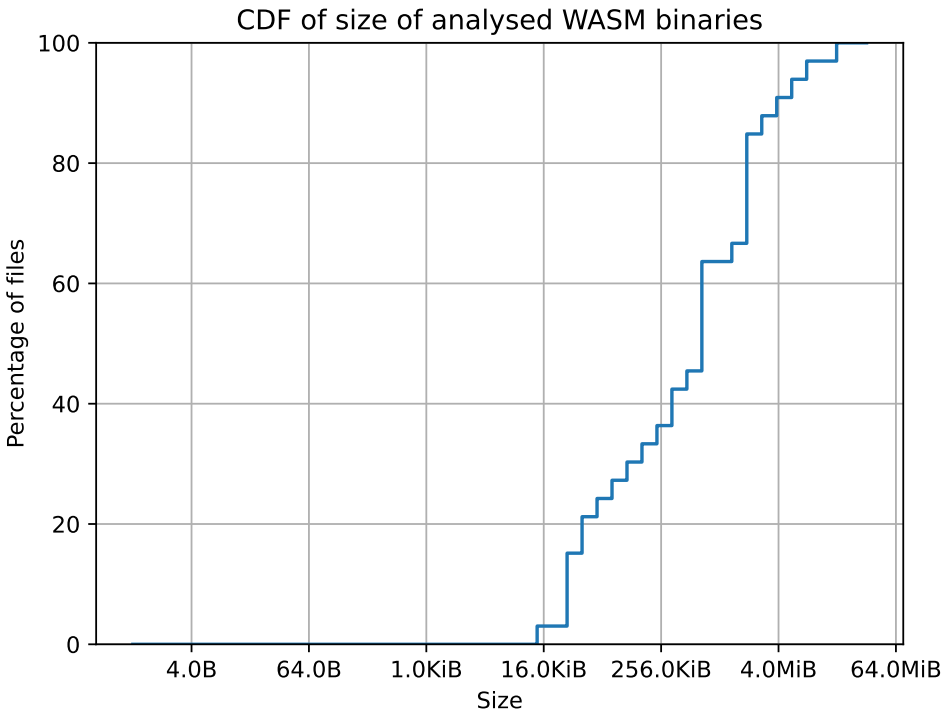


Figure 5.4: Distribution of WebAssembly file sizes.

at the same time, making the page load slower. We thought some websites might have followed this approach however we were wrong and fortunately, this clunky way is not used from what we can infer.

We also observe a minimum size of 0B transferred, we believe for the same reasons mentioned before, since the total URL size is measured by summing the values of HTML, JavaScript, and Wasm. The value of zero bytes can be justified by a web server that identified content as HTML (in this case) in the `Content-Type` HTTP header and then did not respond or took to long to send the contents. Through percentile 25 we can see that there are a lot of web pages that contain very little HTML and through percentile 50 that half of them have at least 8KiB, which we consider a rather small amount of HTML data for a regular webpage. Since the maximum size is way apart from percentile 75 we can infer that between 69.4KiB and 13.6MiB we can find a lot of files with varying sizes, meaning data is very irregular in this zone, making big websites (bigger than 1MiB) a minority of the dataset. This can also be observed by the minimal difference between the average size and percentile 75.

We can also see that the number of HTML files is bigger than the number of URLs analyzed, meaning that there are some websites fetching some content that is in HTML format, which we also studied.

5.3 WebAssembly Analysis

The main focus of our study is in WebAssembly binaries, so this analysis is the core of our investigation. We studied the amount of imported and exported functions as it constitutes the main point of interaction of WebAssembly code to the outside, mainly JavaScript code. We also wanted to take a look at the size of the binaries we were able to find.

Minimum size	0.0B
Average size	1.9MiB
Maximum size	20.7MiB
Percentile 25	41.2KiB
Percentile 50	627.4KiB
Percentile 75	2.3MiB
Total size	70.0MiB
Total unique files	36
Maximum number of reuses	19

Table 5.3: WebAssembly statistics of the dataset.

Minimum number of imports	0
Minimum number of exports	0
Average number of imports	61.33
Average number of exports	32.36
Maximum number of imports	354
Maximum number of exports	184

Table 5.4: WebAssembly imports and exports statistics.

In Figure 5.4 we can see that 20% of the total collected WebAssembly files is around 50KiB or less, meaning they are very small, as one would expect from a file in a binary format. We can see two spikes between 256KiB and 4MiB, meaning this is where the biggest number of WebAssembly binaries is located, which we consider being a range close to one of the real world, because a binary that is a lot less than 256KiB probably has very little function and may not be worth its existence, and one bigger than 4MiB is more likely to be a very important program, which handles big chunks of data, most likely a fully-featured application or a library, which are not very common yet.

The number of binaries we found was way below what we were expecting, but still, we believe 36 is a reasonable number for our dataset. The minimum size we observed was 0B, meaning we could not receive and analyze the response from the website. We can see that the average is almost three times the value of percentile 50, meaning that data is not well balanced, so we have fewer files that are bigger, as we saw in the CDF plot. So we also expect percentile 75 to be far from the maximum size we collected since data is very sparse in this area.

Since a binary can be included by various sources, we also studied this parameter and found out that "the most popular" one in our dataset was included 19 times and corresponds to Amazon IVS Player[2]. It is a web worker that aims at optimizing video playback and is used in websites like Twitch[3], which is hosted on Amazon AWS; and other video-related websites.

We also wanted to evaluate the functions that constitute a communication point with the outside. For this, we converted Wasm files from binary to text format and counted the number of imported and exported functions. The result is plotted in Figure 5.5. We observe that there are binaries that do not use external APIs or that do not allow their functions to be called from the exterior. However, there are some who import a big number of APIs, meaning they require a lot of access from the outside. We consider 354 to be a rather huge number of imports, but as we have manually checked this is a game, so it needs

[2] `https://docs.aws.amazon.com/ivs/latest/userguide/player.html`
[3] `https://twitch.com`

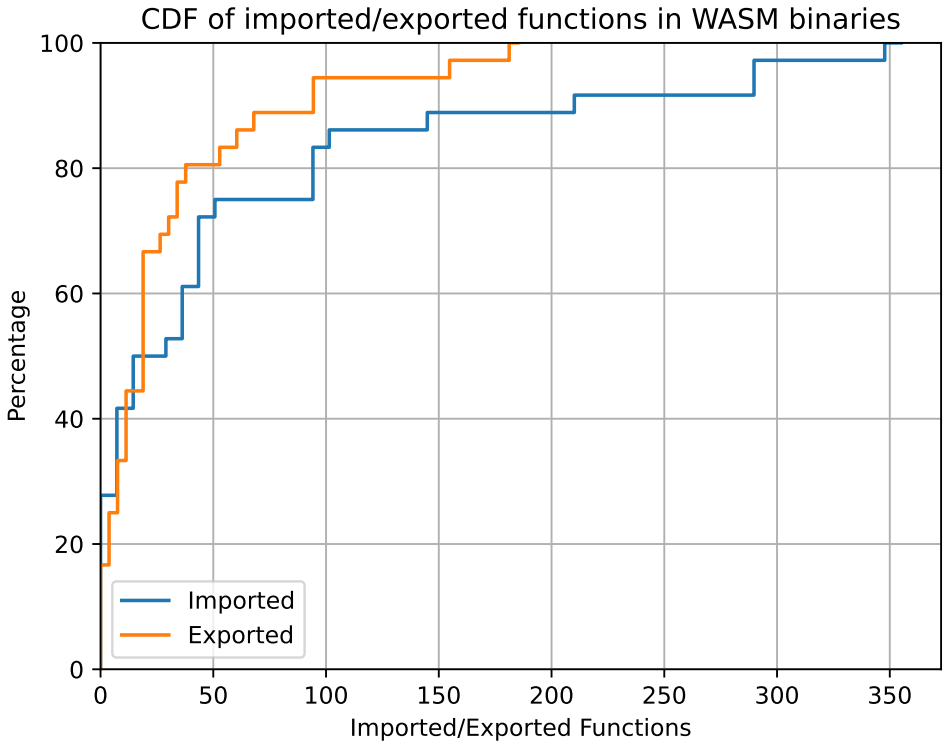


Figure 5.5: Imported and exported functions in WebAssembly.

to call a lot of external functions in order to draw its elements. Unfortunately, we were not able to view which functions it is calling because they are minified. On the other side, we have a binary that exports 184 functions, which in a preliminary analysis looks like a library. After manual inspection, we found out it is a library from Tableau[4] to help in data visualization since this is the area of work of this company.

Taking a look at the averages of both parameters, they are higher than our expectations, which are around half of the presented values. We did not expect that WebAssembly programs would interact this much with the outside, opening space for the existence of vulnerabilities.

Finally, we also tried to identify the compilers that generated the binaries we collected. We performed this analysis in a very simple way, just to have a very high-level overview of the toolsets used currently. The results are presented in Figure 5.6 and we can see that Emscripten is the most popular one, as we would expect since most code bases still rely on unsafe languages like C/C++. Followed by it, there is a piece where we could not identify the compiler due to the very naive approach we took. Then, we have Rustc with a non-negligible percentage, allowing us to infer that there is also a small yet significant presence of Rust in the web, making the arising of vulnerabilities more complicated.

5.4 JavaScript Analysis

Finally, the last analysis we performed was on JavaScript and we defined that we wanted to analyze the following parameters:

[4] https://www.tableau.com

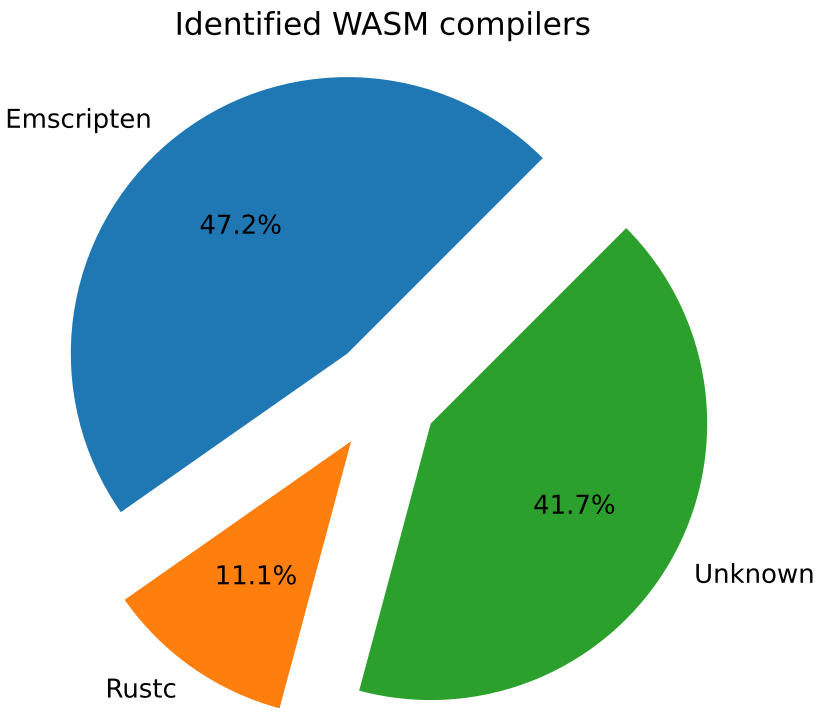


Figure 5.6: Compilers of Wasm binaries in the dataset.

Minimum size	0.0B
Average size	100.3KiB
Maximum size	23.8MiB
Percentile 25	2.4KiB
Percentile 50	15.3KiB
Percentile 75	109.2KiB
Total size	78.3GiB
Total unique files	818576
Maximum number of reuses	75277
Calling WebAssembly API	0
Importing WebAssembly Memory	0

Table 5.5: JavaScript statistics of the dataset.

- Usage of typed arrays, SharedArrayBuffers, and ArrayBuffers
- Size of JavaScript files
- Check or usage of WebAssembly API
- Reuses of JavaScript files

The CDF of sizes of the analyzed JavaScript files is represented in Figure 5.7. From this plot, we can see that around 40% of the files are approximately 10KiB or less, meaning they are not very big. However, since the modern web uses minified and obfuscated code, we can fit a lot of code in less space. If we were to prettify all this code to make it readable, we would expect it to be approximately 50% larger, however, the minification process could have saved up to around 80% of bytes [80].

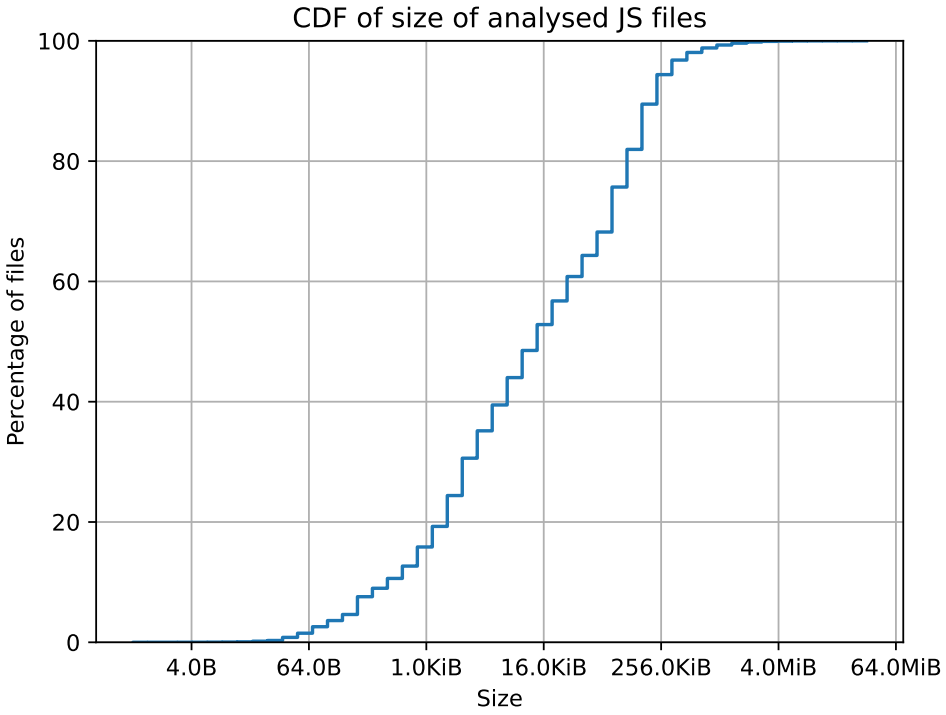


Figure 5.7: Distribution of JavaScript file sizes.

By taking a look at Table 5.5, we can have a deeper understanding of the JavaScript present in the dataset. The minimum file size is also 0B, for all the reasons we mentioned in the previous analyses. The average size of a file of this type is 100.3KiB, which we consider a normal value, since today it is very common to include popular libraries in a website, like JQuery[5] or React[6], which are around this size. The maximum size is way above what we expected since we want pages to be as light as possible, so we manually inspected this file and it belongs to Evolutto[7], containing a JavaScript library that is not minified, hence the enormous size. We can also observe that from the percentiles, the sizes are not too far apart, so it means this may be just an exception. We identified one file as being reused by 75277 links, leaving us curious on what it should be, due to the huge number of appearances. This file is just a single-line file, containing a call to `processGoogleToken` pointing to Google's AdService, a service to configure advertisements to websites, so it makes sense to have such a big number of inclusions.

Regarding the WebAssembly API, we did not find any direct calls to the API (`WebAssembly.instantiate` for example) and also did not find any direct memory imports to WebAssembly modules, due to the process we mentioned before - minification – that makes the task of analyzing JavaScript much harder. We believe that at least the major JavaScript libraries have calls to WebAssembly modules in their code, even if they are not used in the analyzed dataset.

Figure 5.8 shows us the distribution of the arrays we wanted to evaluate. We can see that typed arrays are the most used in JavaScript, not surprisingly, since they are constituted by multiple array types. Followed by them, we have ArrayBuffers and with a very small percentage, SharedArrayBuffers.

[5] `https://jquery.com/`
[6] `https://reactjs.org/`
[7] `https://evoluto.com`

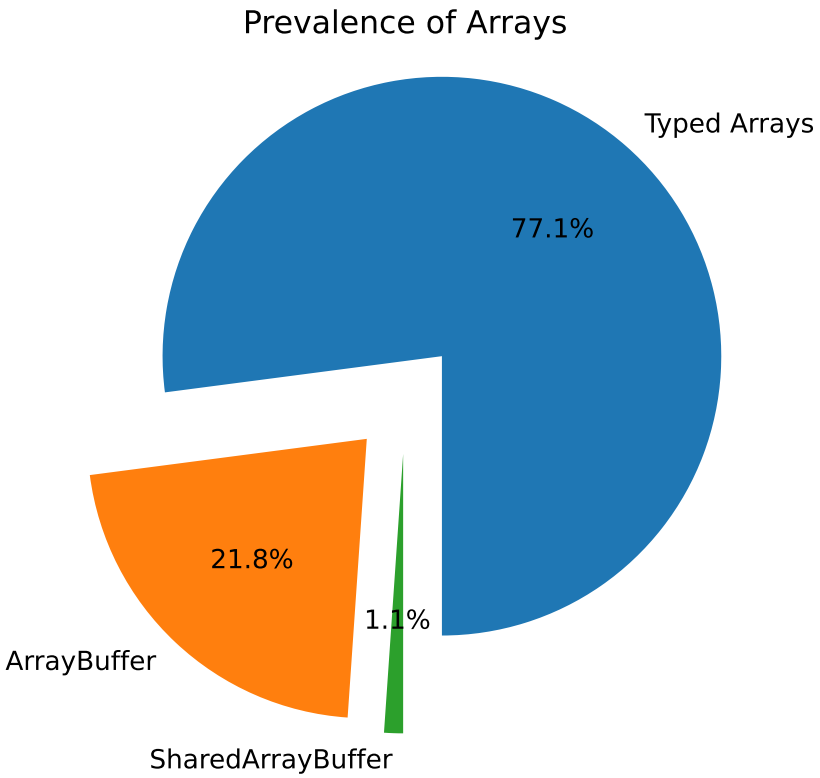


Figure 5.8: Distribution of arrays in JavaScript code

Since WebAssembly supports ArrayBuffers and SharedArrayBuffers to create a memory, we can see that we do not have a huge amount of opportunities to create them directly with these types, but the percentage is still pretty high.

Since the typed arrays are the most prevalent ones, we decided to evaluate them more deeply and the outcome resulted in Figure 5.9. The most common type of array is Uint8Array, meaning it can be wrapped around an ArrayBuffer and then be used as a byte-addressable memory, very similar to computer RAM. Uint32Array is the second biggest and with that, we would have more performant memory accesses. All the other types are very close in percentage but their use for WebAssembly could be more application specific.

Summary

This chapter described the dataset collection process along with the analyses performed. It addressed the problems in collecting the dataset. It also described each one of the performed analyses, focusing more on WebAssembly and its interaction with the outside (JavaScript). As we have seen, there are multiple ways in which we can interact with WebAssembly. The next chapter presents the proof-of-concept exploit generation for WebAssembly binaries.

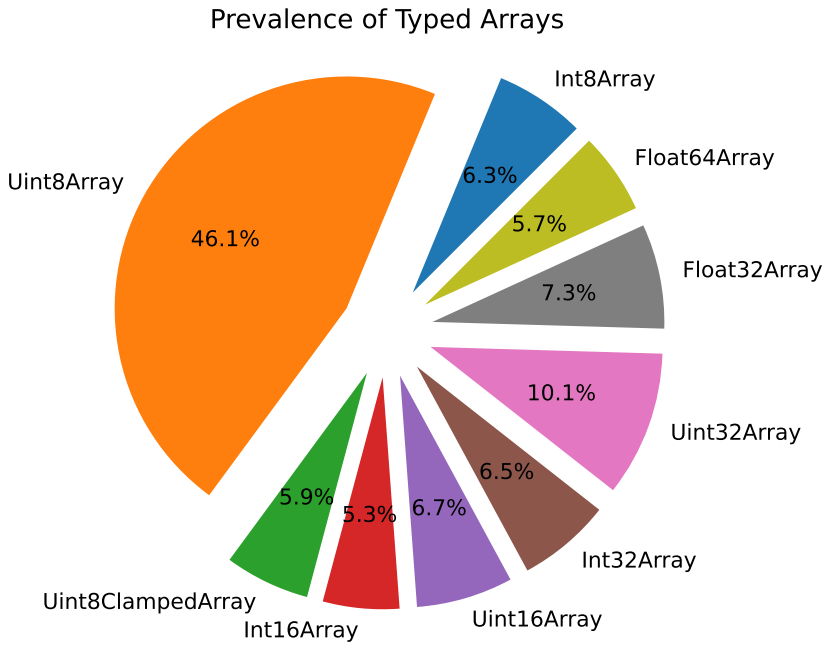


Figure 5.9: Detail on TypedArrays

6

Proof-of-Concept Exploit Generation for WebAssembly

Contents

This chapter presents WebAssault, the tool that we envision to automatically create exploits in JavaScript that trigger vulnerabilities in WebAssembly code. It is an initial step towards generating complex exploits for vulnerabilities present in WebAssembly binaries. We begin with an overview of the design of our tool (Section 6.1). In Section 6.2, we present some of the challenges we faced. Lastly, we discuss implementation details (Section 6.3) and evaluate the tool with a handmade exploit (Section 6.4).

6.1 Overview

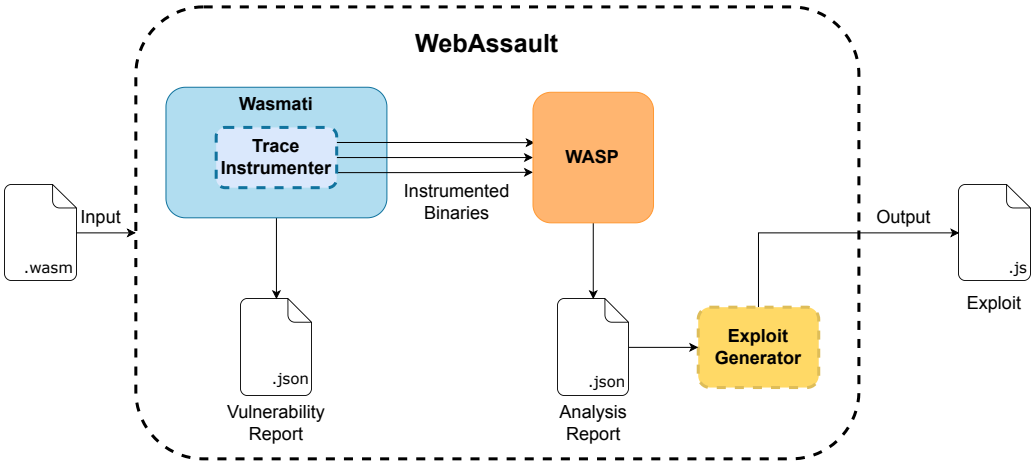


Figure 6.1: System architecture.

Figure 6.1 sketches the architecture of our tool, WebAssault, which aims to generate exploits for WebAssembly vulnerabilities. Dashed lines represent components developed by us whereas the solid ones represent already-existent components. It is composed of multiple stages with different purposes:

1. *Wasmati:* A WebAssembly binary is fed into Wasmati for it to create a CPG which will be queried for the common vulnerabilities. A report (*vulnerability report*) with the findings is then generated in JSON format showing the function(s) where the bug(s) is(are) present, the type of the vulnerability(ies), and a small description of it(them).

2. *Trace Instrumenter:* This component receives the WebAssembly binary and for each vulnerability reported, we trim the possible paths that WASP would try in order to prevent it from exploring paths that will not lead to the vulnerability we are testing. We need access to the binary in order to patch it with instructions that guide WASP to the possibly vulnerable path. Multiple instances of instrumented binaries may be generated, depending on the number of vulnerabilities detected by Wasmati.

3. *WASP:* Executes the instrumented binary provided by the trace instrumented. WASP's concolic execution is guided to execute the program along the path that was selected in the previous step. If the path leads to a real vulnerability, we will get as output a JSON report (*analysis report*) containing the input value(s) of the program that can trigger the vulnerability, and consequently the information needed to generate a simple exploit.

4. *Exploit Generator:* The last component of our system uses as input the analysis report produced by WASP in order to generate the exploit that triggers this vulnerability in the code. If the vulnerability can be triggered, it outputs an exploit written in JavaScript that when executed will also take advantage of it. Otherwise, it will simply disregard this vulnerability, as WASP could not find an execution path so it will not receive any meaningful input.

6.2 Implementation Challenges

For our tool to fulfill the task of automatically generating exploits, using Wasmati and WASP, meaning, static and dynamic[1] analysis, respectively, we need to implement the modules we pictured as dashed in Figure 6.1. We aim at generating multiple binaries based on the vulnerabilities Wasmati identifies. The main challenge here is to trim all of the possible paths on a binary, for it to follow only the one intended, as we will explain in Section 6.2.1. Because we are dealing only with the WebAssembly file it may also be hard to find the exact type of arguments that are being passed on to a Wasm library. To handle this issue we need to analyze the code flow in JavaScript, which we discuss in Section 6.2.2.

6.2.1 Trimming Paths in the Concolic Execution

Concolic execution works by exploring all the possible paths a program can go through, using both symbolic and concrete executions. It starts by generating a random input for the symbolic variables and explores one path until the end collecting all the conditions that were satisfied along the way (path-condition π_1). When it terminates, the global path condition $\Pi = \top$ is updated with this path condition becoming $\Pi = \top \vee \pi_1$, and a new input satisfying the negation of the already explored path conditions ($\neg\Pi$) is generated so that a different path is now explored. The process continues updating the global path condition at the end of each execution i with the path condition π_i, $\Pi = \pi_1 \vee \cdots \vee \pi_{i-1} \vee \pi_i$, and generating an input satisfying $\neg\Pi$, continuing as such until all possible execution paths have been traversed and no other input can be generated in this way, *i.e.*, $\neg\Pi$ becomes UNSAT. The main challenge of this approach is the well-known path explosion problem. However, in our work, since we have the trace from Wasmati, we want to test only a single path at a time, so we need to cut all the other ones in order to obtain the inputs we need in a timely manner. Also, because WASP extends the WebAssembly syntax, we have support for more instructions that can help us guide concolic execution in the way we desire.

Specifically, our idea to guide WASP's concolic execution is as follows. Since we have access to the CPG of the module we are testing, we are able to get the execution paths by analyzing the control flow graph, allowing us to trim the ones we do not want to test by patching the WebAssembly binary with instructions that will make the path condition automatically *false* in these zones. To illustrate our idea, consider Figure 6.2a. This figure shows a function, `func`, that calls a well-known dangerous function `gets` if certain conditions are met. In this example, we have multiple execution paths, which can be seen in the control flow graph in Figure 6.2b. Two execution paths only change the `ret` variable and return

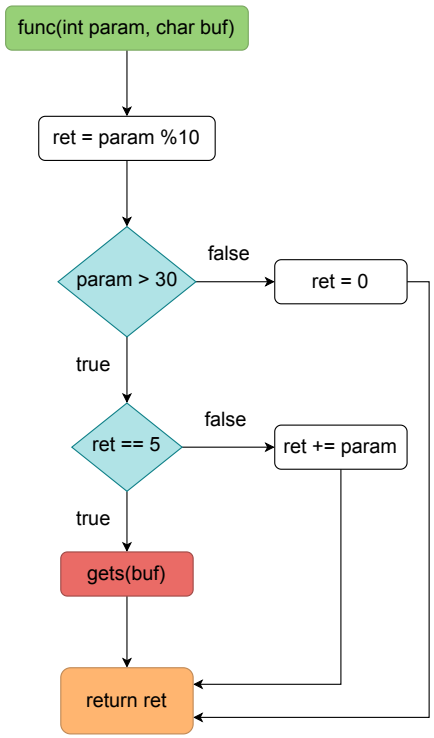
[1] By dynamic analysis in this context we mean concolic execution.

```
int func(int param, char* buf) {
  int ret = param % 10;

  if (param > 30) {
    if (ret == 5) {
      gets(buf);
    } else {
      ret += param;
    }
  } else {
    ret = 0;
  }

  return ret;
}
```

(a) Original code.



(b) Control flow graph.

Figure 6.2: Example of a dangerous function only reachable from one path.

its value, while the remaining one calls `gets` and it is then vulnerable. In order for WASP to efficiently compute the inputs for this path, we need to cut the paths on the right, coming from the blue elements.

To cut a path, we can simply add the instructions `i32.const 0; sym_assume`, because we need the value on top of the stack to be **0** and we also need to tell the concolic execution engine to add this value to the path condition. These instructions are inserted in the branches that we know are not needed for our execution, allowing us to explore the path we need quicker. Figure 6.3a presents the Wasm code of our example and in Figure 6.3b we can see the instrumented code that prunes the execution paths that do not lead to the dangerous function, lines 25 and 26, and 32 and 33. The path-trimming technique just described will be implemented by WebAssault's Trace Instrumenter.

6.2.2 Generating Symbolic Inputs from JavaScript Calls

To determine the correct symbolic type of a variable, we should be able to know how it is declared, or in the case of a function parameter, how it is called. Since a high percentage of WebAssembly code is used on the web, it needs to be called by JavaScript code in order to run. Unfortunately, as of today, WASP cannot determine correctly the proper symbolic type to assign to symbolic variables. In some cases, this limitation is due to the fact that they may be a reference to variables in WebAssembly's linear memory (arrays). Since variables are only referenced using an index, it is quite hard to determine their length. Therefore, our solution is to analyze the data type when the function is called in JavaScript, where we can clearly determine the type of each variable being used.

To showcase our approach consider Figure 6.4, where we have an example of an array being passed

```
(func $func (type $t8) (param $param
↪  i32) (param $buf i32) (result i32)
  (local $ret i32) (local $l3 i32)
  local.get $param
  i32.const 10
  i32.rem_s
  local.set $ret
  i32.const 0
  local.set $l3
  block $B0
    local.get $param
    i32.const 31
    i32.lt_s
    br_if $B0
    block $B1
      local.get $ret
      i32.const 5
      i32.ne
      br_if $B1
      local.get $buf
      call $gets
      drop
      i32.const 5
      return
    end
    local.get $ret
    local.get $param
    i32.add
    local.set $l3
  end
  local.get $l3)
```

(a) WebAssembly generated code.

```
(func $func (type $t8) (param $param
↪  i32) (param $buf i32) (result i32)
  (local $ret i32) (local $l3 i32)
  local.get $param
  i32.const 10
  i32.rem_s
  local.set $ret
  i32.const 0
  local.set $l3
  block $B0
    local.get $param
    i32.const 31
    i32.lt_s
    br_if $B0
    block $B1
      local.get $ret
      i32.const 5
      i32.ne
      br_if $B1
      local.get $buf
      call $gets
      drop
      i32.const 5
      return
    end
    i32.const 0
    sym_assume
    local.get $ret
    local.get $param
    i32.add
    local.set $l3
  end
  i32.const 0
  sym_assume
  local.get $l3)
```

(b) Instrumented WebAssembly code.

Figure 6.3: Differences highlighted between the generated code (left) and the code that WebAssault will generate for WASP to process (right).

```
1  const LEN = 10;
2  const MAX = 1000;
3  var sort = Module.cwrap('sort', '', ['number', 'number']);
4  var create_buff = Module.cwrap('create_buff', 'number', ['number']);
5  var free_buff = Module.cwrap('free_buff', '', ['number']);
6
7  var my_numbers = Uint32Array.from({length: LEN}, () => Math.floor(Math.random() *
   ↪  MAX));
8
9  console.log(my_numbers);
10
11 var buf = create_buff(LEN);
12 Module.HEAPU32.set(my_numbers, Math.floor(buf / 4));
13
14 sort(buf, LEN);
15
16 my_numbers = new Uint32Array(Module.HEAPU32.buffer.slice(buf, buf + LEN * 4));
17
18 free_buff(buf);
19
20 console.log(my_numbers);
```

Figure 6.4: JavaScript code calling WebAssembly functions.

from JavaScript to be sorted by a WebAssembly module (line 14). By analyzing only the WebAssembly code, we would not be able to determine the length of the variable that we here call `my_numbers`. In contrast, since we have access to the JavaScript calling the function, we can clearly see that it is an array of random elements of type `i32` (line 7) and has length *10* (line 1). With this new information, we can create a new function in the WebAssembly module, that will create a symbolic array of `i32` elements and will later be recognized as so by WASP. This new array would then be used as an argument to call the function called by JavaScript, in our case `sort`, enabling us to more accurately determine if in fact, a bug is present in the code or if it is just a false positive reported by the Wasmati tool. If there are multiple calls to the same function and they use arrays of different sizes, we run WASP multiple times for the same path, but with different symbolic types in order to better test all the possible values we can get.

With this approach, we could inject code in the original WebAssembly binary to explicitly assign symbolic types to variables and then call the function we want to analyze with them. By analyzing this new function, WASP will now be able to correctly determine the inputs that are needed in order to properly traverse our path. However, this would still be hard and take some time, since we would have to iterate through all the positions in the array to make them symbolic and would have to patch the exports that the module provides, in order to export our function instead of the older one, that does not have the instructions to handle symbolic arrays.

6.3 Implementation

We decided to start with the instrumentation of the binaries. We started simple and tried to address first the dangerous functions vulnerability, where Wasmati tracks the usage of functions that if used can

certainly issue a vulnerability, like `gets` or `read`. With these functions, it is fairly easy to generate a buffer overflow vulnerability since `gets` completely disregards the size of the buffer it uses and `read` may receive a maximum number of characters to read higher than the size of the destination location.

To implement the instrumenting behavior on Wasmati, we created another visitor which allows us to write Wasm files in text format. We also defined a new data structure that could encode the execution flow paths that can reach the vulnerabilities. This new structure stores a list of CPG nodes in reverse order, representing the path from the vulnerability to all the possible sources that can trigger it. We achieved this by creating a path with just a single node - the one where the vulnerability is identified - and performing a modified Breadth-First Search (BFS) algorithm on it.

This BFS runs from the vulnerable node, in reverse order, until we reach the `Module` node, which is used in the CPG to represent the root of the graph. If a path we were taking did not reach this node, it is discarded. The algorithm normally follows only the Control Flow Graph (CFG) and Call Graph (CG) edges, since these are the ones that show us the information flow through the module. However, to reach the root node we have an exception, where we need to follow an Abstract Syntax Tree (AST) edge to get from the `Instructions` node to it, so we checked if the destination of the edge was the node with the instructions and the type of the edge was AST. To obtain all the possible paths to the vulnerable node this BFS maintains a queue of paths that need to be expanded and it expands each one of them separately. Whenever we reached a branching point or a function that could be called by JavaScript (*i.e.* exported by the module), we would clone the path and add the source nodes of the branching point to the original path and its copy. One of the nodes would continue to be expanded and the other one would be appended to the queue. This means that in a `br_if` instruction, we would generate two different paths from an initial one. Through this modified version of the BFS algorithm, we are able to obtain all the possible execution paths that lead to the vulnerable function.

After this, the found execution paths are returned and converted to JSON so that the relevant labels of the nodes can be printed to the vulnerability report generated by Wasmati and allow the user to make a manual inspection if desired.

Finally, we create a directory named `instrumented` where we will place the instrumented WAT files. In this phase, we receive the paths we obtained from all the vulnerabilities and run the previously mentioned visitor on each one of them. The `wat-instrumenter` is fed with the CPG and a path. It starts traversing the graph by the root module, trying to reconstruct a file very similar to the one that generated it, in text format. Whenever it encounters a function node it tests if it is the function where the vulnerability was found, if it was it will trim the paths inside of it, otherwise it will just print the regular function. We followed this approach because other functions in the module can have multiple execution paths and some of them may be useful to our path, but since we could not decide on which ones were and which did not, we took a conservative perspective and allowed them to be fully present. This allows us to decrease the number of false positives WASP could issue due to calculating function outputs that could not be generated by a given function, but since we would have defined them as symbolic, they would be valid for WASP. As stated in Section 6.2.1, we were able to prune branching conditions using the described technique where the path should not be taken.

The instrumentation functionality can be activated for Wasmati using the flag `--instrument`, which will activate the `--native` flag since we need to have the native queries determine vulnerabilities in order to instrument the file accordingly.

We were able to generate valid WAT files that can be given as input to WASP. However, when testing for the dangerous functions vulnerability, we noticed that WASP could not detect the vulnerability since in our tests we were using a local array, not a heap-allocated one. This is one limitation of WASP: it can only reason about the dynamic memory, more concretely, the WebAssembly unmanaged stack, because uses the official WebAssembly interpreter with a patch. It is able to reason about this type of memory because it overwrites the `alloc` and `free` instructions with a call to one of its own hooks, in order to obtain information about the size of each allocation and when they are freed.

So, we needed to move to a vulnerability that WASP could support but that was not related to static buffers. This was a serious problem since Wasmati is a static analysis tool, it has a limitation on the type of variables it can reason about. Reasoning about dynamic memory is too hard for a tool of this type since it would need to have a map of the memory and in WebAssembly this map can be of around 4 GiB, making it unreasonable. The only possibilities of vulnerabilities we had left to test were double-frees, use-after-frees, and buffer overflow with a static allocation size. Since buffer overflows are the most simple ones, we chose to switch to them.

Applied the same process as described earlier on this new type of vulnerability and we were also able to generate functional WebAssembly binaries. To generate the test case for each of the runs, we used C and then compiled them with WASP-C. However, since WASP-C compiles a file and already fills it with the hooks needed for WASP, we needed to perform a modification if we wanted the file not to receive these modifications because Wasmati would not be able to read the instructions WASP uses. After doing this, we ran Wasmati on the generated file and it gave us the vulnerability we wished for and an instrumented file. After passing this instrumented file through the post-processing WASP needs, we were able to make WASP find an input that would trigger the vulnerability. With the output from WASP, it is relatively easy to generate an exploit to trigger the vulnerability.

6.4 Evaluation

We came up with the very simple buffer overflow present in Figure 6.5a, where there is a function that receives an argument. In this function, we dynamically allocate a buffer with 256 bytes and then we check if the argument we received was 1337. If it did not correspond to this number, everything is fine, however, if the argument was 1337, then we would write 10000 bytes on a buffer that can only fit 256. In Figure 6.5b, we can clearly see that WASP was capable of determining the correct input and so, we could generate an exploit that would be `vuln(1337)`, which would call the Wasm module's function.

Because of all the limitations of WASP and Wasmati, we only managed to generate PoC exploits for WebAssembly, so experiments with real-world binaries still need to be performed to assess the applicability of WebAssault in real scenarios. However, we showed that it is possible to apply this technique to Wasm using static and dynamic analysis together.

```c
void vuln(int i) {
    char *buff = malloc(256);

    if (i == 1337) {
        read(0, buff, 10000);
    } else {
        printf("NOPE");
    }
}
```

(a) Proof-of-Concept C vulnerable code.


```json
{
  "specification": false,
  "reason": {
    "type": "Overflow",
    "line": "235.9-235.28"
  },
  "witness": [
    {
      "name": "arg",
      "value": "1337",
      "type": "i32"
    }
  ],
  "coverage": "80.8118081181",
  "loop_time": "0.078536",
  "solver_time": "0.012413",
  "paths_explored": 2,
  "instruction_counter": 305785,
  "incomplete": true
}
```


(b) WASP output for vulnerability.

Figure 6.5: Buffer overflow with static malloc size.

Due to time limitations and complexity of analyzing JavaScript, we were not able to address the second proposed challenge (Section 6.2.2). JavaScript is already a very complex language and is constantly evolving making it very hard to keep up with all of the features it has. In order to analyze it we would need more time and we would need to develop a code analysis tool that would analyze the files and feed this data to Wasmati. This task is left for future work.

Summary

This chapter presented WebAssault, our framework for automatically generating exploits for WebAssembly vulnerabilities. We presented the overall architecture of the framework. We also showed some of the challenges we intended to tackle. Finally, we presented the implementation and evaluated our tool with a simple example, and with it, we were able to generate an exploit.

7

Conclusions

Contents

7.1 Achievements

WebAssembly is a new format that is becoming more present in the modern web by running code more efficiently in the browser and also by being very compact. To the best of our knowledge, no study has been performed that evaluated the prevalence of WebAssembly in the wild, so there also does not exist a dataset comprised of only wild Wasm binaries. We are also not aware of any tool capable of creating automatic exploits for vulnerabilities present in WebAssembly binaries.

We evaluated the web by looking for HTML, JavaScript, and WebAssembly files, generating a representative dataset that was later analyzed to the relative presence of different file types and other metrics specific to the file type. We saw that besides being popular, Wasm's footprint is almost negligible on the current web, but we hope to see an increase in the coming years. We proposed WebAF as a web analysis framework and verified that it could scale very well, allowing for multiple parallel scans to occur.

In this thesis we also propose WebAssault as the first framework capable of generating Proof-of-Concept exploits for WebAssembly vulnerabilities combining static and dynamic analysis. We were capable of generating a functional exploit for a handmade vulnerability.

7.2 Future Work

In the future, we would like to improve the crawler present in our framework, allowing it to be more easily deployed, and study how it could scale to more than one scheduler, enabling it to improve its scalability. Due to time constraints, we could not address all of the challenges we intended to in WebAssault, so, we would like to solve the JavaScript's types analysis challenge. One thing that could be improved in this framework is the Exploit Generator because it is not finished and consequently it is not fully automatic. Since we were only able to test the exploit framework with a very simple example, we would like to test it with more complex data, as well as overcome some of the limitations of WASP and Wasmati, so we could use a wider range of tests and could work with more vulnerabilities. Finally, we would like to automate the whole WebAssault framework, as it requires us to run the programs manually.

Bibliography

[1] D. Herman, L. Wagner, and A. Zakai. asm.js Website. http://asmjs.org/spec/latest/, 2014. Accessed: 2021-12-15.

[2] Google Inc. Native Client. https://developer.chrome.com/docs/native-client/, . Accessed: 2021-12-15.

[3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[4] Mozilla. asm.js - Game development. https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js, 2021. Accessed: 2021-12-15.

[5] Google Inc. Goodbye PNaCl, Hello WebAssembly! https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html, 2017. Accessed: 2021-12-15.

[6] P. Lopes. Discovering security vulnerabilities in webassembly with code property graphs. Master's thesis, Instituto Superior Técnico, https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/846778572212698, 2021.

[7] F. Marques. Robust symbolic execution for webassembly. Master's thesis, Instituto Superior Técnico, https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1128253548922792, 2021.

[8] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 2004.

[9] D. Crockford. Jslint: The javascript code quality and coverage tool. https://www.jslint.com. Accessed: 2021-12-21.

[10] J. Dahse and T. Holz. Simulation of built-in php features for precise static code analysis. In *NDSS*, 2014.

[11] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 2019.

[12] D. Wang, B. Jiang, and W. K. Chan. Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection. *arXiv*, 2020.

[13] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, 2019.

[14] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg : Automatic exploit generation. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2011.

[15] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler. Security chasms of wasm. *NCC Group Whitepaper*, 2018.

[16] Mozilla. JavaScript: Number. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number`, . Accessed: 2022-01-04.

[17] CVE. libpng. `https://www.cvedetails.com/cve/CVE-2018-14550/`, 2021. Accessed: 2021-12-16.

[18] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. 2014.

[19] L. D. Moura and N. Bjørner. Z3: An efficient smt solver. 2008.

[20] W3C Community Group. WebAssembly. `https://webassembly.org`, . Accessed: 2022-01-03.

[21] W3C Community Group. Security - WebAssembly. `https://webassembly.org/docs/security/#memory-safety`, . Accessed: 2022-01-03.

[22] A. Hilbig, D. Lehmann, and M. Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*, 2021.

[23] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of webassembly. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[24] D. Lehmann and M. Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2019.

[25] Q. Stievenart and C. D. Roover. Compositional information flow analysis for webassembly programs. 2020.

[26] D. Lehmann, M. T. Torp, and M. Pradel. Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly. *arXiv*, 2021.

[27] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. 2008.

[28] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *International conference on computer aided verification*, 2007.

[29] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings - IEEE Symposium on Security and Privacy*, 2012.

[30] S. K. Huang, M. H. Huang, P. Y. Huang, H. L. Lu, and C. W. Lai. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability*, 2014.

[31] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, 2016.

[32] L. Xu, W. Jia, W. Dong, and Y. Li. Automatic exploit generation for buffer overflow vulnerabilities. In *Proceedings - 2018 IEEE 18th International Conference on Software Quality, Reliability, and Security Companion, QRS-C 2018*, 2018.

[33] Google Inc. Google. `https://www.google.com`, . Accessed: 2022-09-02.

[34] Microsoft. Bing. `https://www.bing.com`. Accessed: 2022-09-02.

[35] Open Source Software. curl. `https://curl.se`. Accessed: 2022-09-02.

[36] Free Software Foundation, Inc. Wget - GNU Project - Free Software Foundation. `https://www.gnu.org/software/wget/`. Accessed: 2022-09-29.

[37] Software Freedom Convervancy. Selenium. `https://www.selenium.dev/`. Accessed: 2022-08-22.

[38] Google Inc. Puppeteer — Puppeteer. `https://pptr.dev/`, . Accessed: 2022-08-11.

[39] OpenJS Foundation. Node.js. `https://nodejs.org`. Accessed: 2022-09-29.

[40] Englehardt, Steven and Narayanan, Arvind. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[41] Skolka, Philippe and Staicu, Cristian-Alexandru and Pradel, Michael. Anything to hide? studying minified and obfuscated code in the web. In *The world wide web conference*, pages 1735–1746, 2019.

[42] Pochat, Victor Le and Van Goethem, Tom and Tajalizadehkhoob, Samaneh and Korczyński, Maciej and Joosen, Wouter. Tranco: A research-oriented top sites ranking hardened against manipulation, 2018.

[43] Internet Archive. Wayback Machine. `https://web.archive.org/`. Accessed: 2022-09-29.

[44] Amazon. We retired Alexa.com on May 1, 2022. `https://support.alexa.com/hc/en-us/articles/4410503838999-We-retired-Alexa-com-on-May-1-2022`, . Accessed: 2022-08-11.

[45] Amazon. We will be retiring the Alexa.com APIs on December 15, 2022. https://support.alexa.com/hc/en-us/articles/4411466276375, . Accessed: 2022-08-11.

[46] Redis Ltd. Redis. https://redis.io. Accessed: 2022-08-11.

[47] VMWare Inc. Messaging that just works — RabbitMQ. https://rabbitmq.com. Accessed: 2022-08-11.

[48] The PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org, . Accessed: 2022-08-11.

[49] Google Chrome. Google Chrome - Download the Fast, Secure Browser from Google. https://www.google.com/chrome/. Accessed: 2022-08-11.

[50] mitmproxy. mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org/. Accessed: 2022-08-11.

[51] The Linux Foundation. Production-Grade Container Orchestration. https://kubernetes.io/, . Accessed: 2022-08-11.

[52] Oracle. Oracle VM VirtualBox. https://www.virtualbox.org/, . Accessed: 2022-08-11.

[53] HashiCorp. Vagrant by HashiCorp. https://www.vagrantup.com. Accessed: 2022-10-22.

[54] Red Hat Inc. Ansible is Simple IT Automation. https://www.ansible.com. Accessed: 2022-10-22.

[55] Canonical Ltd. Enterprise Open Source and Linux — Ubuntu. https://ubuntu.com. Accessed: 2022-08-16.

[56] Oracle. Chapter 3. Configuring Virtual Machines. https://www.virtualbox.org/manual/ch03.html#settings-processor, . Accessed: 2022-08-11.

[57] The Linux Foundation. Kubernetes Components — Kubernetes. https://kubernetes.io/docs/concepts/overview/components/#control-plane-components, . Accessed: 2022-08-16.

[58] The Linux Foundation. Installing kubeadm — Kubernetes. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm, . Accessed: 2022-08-16.

[59] Docker Inc. Home - Docker. https://www.docker.com/. Accessed: 2022-08-16.

[60] Datadog. 10 Trends in Real-World Container Use — Datadog. https://www.datadoghq.com/container-report/#8, 2021. Accessed: 2022-08-16.

[61] The Linux Foundation. kubelet — Kubernetes. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet, . Accessed: 2022-08-16.

[62] The Linux Foundation. Command line tool (kubectl) — Kubernetes. https://kubernetes.io/docs/reference/kubectl, . Accessed: 2022-08-16.

[63] Tigera Inc. About Calico. https://projectcalico.docs.tigera.io/about/about-calico. Accessed: 2022-08-16.

[64] The Linux Foundation. Helm. https://helm.sh, . Accessed: 2022-08-17.

[65] Bitnami. Bitnami: Packaged Applications for Any Platform - Cloud, Container, Virtual Machine. https://bitnami.com. Accessed: 2022-08-17.

[66] Jason A. Donenfeld. WireGuard: fast, modern, secure VPN tunnel. https://www.wireguard.com/. Accessed: 2022-08-17.

[67] Jim Salter. WireGuard VPN makes it to 1.0.0—and into the next Linux kernel. https://arstechnica.com/gadgets/2020/03/wireguard-vpn-makes-it-to-1-0-0-and-into-the-next-linux-kernel, 2020. Accessed: 2022-08-17.

[68] etcd. etcd. https://etcd.io. Accessed: 2022-08-17.

[69] Defined. Defined Networking. https://www.defined.net. Accessed: 2022-08-20.

[70] Oracle. Chapter 6. Virtual Networking. https://www.virtualbox.org/manual/ch06.html#network_bridged, . Accessed: 2022-08-20.

[71] Google Inc. Creating Highly Available Clusters with kubeadm — Kubernetes. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability, . Accessed: 2022-08-20.

[72] HAProxy. HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. https://www.haproxy.org. Accessed: 2022-08-20.

[73] Python Software Foundation. Welcome to Python.org. https://www.python.org. Accessed: 2022-08-22.

[74] MongoDB Inc. MongoDB: The Developer Data Platform — MongoDB — MongoDB. https://www.mongodb.com, . Accessed: 2022-08-22.

[75] The PostgreSQL Global Development Group. PostgreSQL: Documentation: 14: psql. https://www.postgresql.org/docs/current/app-psql.html, . Accessed: 2022-08-22.

[76] MongoDB Inc. mongoimport — MongoDB Database Tools. https://www.mongodb.com/docs/database-tools/mongoimport, . Accessed: 2022-08-22.

[77] Philip Hawkes and Michael Paddon and Gregory G. Rose. Musings on the Wang et al. MD5 Collision. Cryptology ePrint Archive, Paper 2004/264, 2004. URL https://eprint.iacr.org/2004/264.

[78] Merrill, Nick. Better Not to Know? The SHA1 Collision & the Limits of Polemic Computation. In *Proceedings of the 2017 Workshop on Computing Within Limits*, 2017.

[79] Mozilla. WebAssembly.compile() - WebAssembly — MDN. https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/compile, . Accessed: 2022-10-03.

[80] Gribble, Steve. JavaScript minification study. https://www.gribble.org/techreports/minification/. Accessed: 2022-10-13.