Bringing Binary Exploitation at Port 80: Understanding C Vulnerabilities in WebAssembly

Emmanuele Massidda[a], Lorenzo Pisu[b], Davide Maiorca[c] and Giorgio Giacinto[d]
Department of Electronic and Computer Engineering, University of Cagliari, Italy

Keywords: Web Assembly, Wasm, Software Security, Web Security.

Abstract: WebAssembly (Wasm) has emerged as a novel approach for integrating binaries into web applications starting from various programming languages such as C, Rust and Python. Despite the numerous claims about its memory safety, issues such as buffer overflow, format strings, use after free, and integer overflow have resurfaced within Wasm. These vulnerabilities can be used to impact web application security, potentially leading to critical issues like Cross-Site Scripting (XSS) and Remote Code Execution (RCE). Our work aims to demonstrate how memory-related vulnerabilities in C codes, when compiled into Wasm, can be exploited for XSS and RCE. Our methodology proposes proof of concepts related to exploiting important stack- and heap-based vulnerabilities. In particular, we demonstrate for the first time that specific vulnerabilities (such as format string) can be effectively employed to achieve arbitrary read and write in Wasm contexts. Our results pose serious concerns about the reliability of Wasm in terms of memory safety, which we believe should be addressed in the next releases.

1 INTRODUCTION

Presenting itself as a fast, efficient, and safe new binary instruction format, WebAssembly (Wasm) found its way into the realm of web applications, enabling developers to build performance-oriented websites that run binary code directly in the browser. WebAssembly allows using languages such as C/C++, C#, and Rust to be compiled into Wasm to run on the web (Wang, 2021; Jazayeri, 2007; Ray, 2023). This allows developers to recompile existing code for use in their web applications, enhancing re-usability and efficiency. This technology has already been implemented in many commercial products, including AutoCAD and Figma (Lehmann et al., 2020; Hilbig et al., 2021), highlighting its growing popularity due to its potential to enhance runtime performance (De Macedo et al., 2022; Yan et al., 2021).

Despite its numerous advantages, adopting WebAssembly introduces many security challenges, particularly when porting code written in languages known for their vulnerability to certain classes of bugs, such as C (Stiévenart et al., 2022). The process of compiling code to WebAssembly, e.g. using tools like Emscripten (a compiler that uses Clang and LLVM (Low-Level Virtual Machine) to compile to WebAssembly and JavaScript (Zakai, 2011)), does not neutralize the vulnerabilities within the new runtime environment. On the contrary, their exploitation may create a dangerous link between traditional binary exploitation and contemporary web exploitation tactics. The risk is amplified by WebAssembly's and Emscripten's lack of built-in security features like Address Space Layout Randomization (ASLR) or stack canaries, which are commonly employed in compilers for traditional binaries. Although Emscripten generates standard compile-time error warnings similar to those of traditional compilers, it does not detect many vulnerabilities that only emerge during runtime, exposing WebAssembly to a wider range of exploits.

Our work investigates how vulnerabilities typical of memory-unsafe languages (like C) manifest within WebAssembly applications when compiled using Emscripten. Specifically, we focus on stack and heap-based vulnerabilities whose exploitation may lead to obtaining complete control of the target server. We propose proof of concepts that exploit the characteristics of Wasm to manipulate the program flow even in unexpected ways. We explain how these vul-

[a] https://orcid.org/0009-0003-9190-5648
[b] https://orcid.org/0009-0001-0129-1976
[c] https://orcid.org/0000-0003-2640-4663
[d] https://orcid.org/0000-0002-5759-3017

Massidda, E., Pisu, L., Maiorca, D. and Giacinto, G.
Bringing Binary Exploitation at Port 80: Understanding C Vulnerabilities in WebAssembly.
DOI: 10.5220/0012852400003767

In *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*, pages 552-559
ISBN: 978-989-758-709-2; ISSN: 2184-7711

nerabilities can simplify traditional Web exploitation techniques, such as Cross-Site Scripting (XSS) and Remote Code Execution (RCE).

The main contributions of our paper can be summarized as follows:

- Providing a structured presentation of Wasm vulnerabilities by systematically categorizing and detailing them, also illustrating their potential to introduce risks on both the client and server sides.
- Providing a clear and reproducible methodology for better understanding the security aspects of Wasm and offering a more organized and cohesive analysis that leads to exploiting previously unexplored vulnerabilities (e.g., Format Strings).
- Analyzing six Wasm vulnerabilities, namely Buffer Overflow (BOF), Format string, Use After Free (UAF), Integer Overflow, Improper Validation of Array Index and Redirecting Indirect Calls, and develop the related Proof of Concepts (PoC) that allow to achieve arbitrary read and write. In particular, we also show how exploiting these vulnerabilities can lead to web-related attacks such as XSS and RCE.

To facilitate the understanding and replication of our findings and furnish a practical resource for further research, we have made the source code for all PoCs available on GitHub (Massidda, 2024). We believe that our work poses major concerns to Wasm security, and we hope it can inspire possible improvements in its next releases.

2 BACKGROUND

WebAssembly is a binary instruction format for a stack-based virtual machine designed for efficient execution on web browsers (Haas et al., 2017). It provides a portable compilation target for high-level languages like C, C++, and Rust, allowing developers to run their code in a web environment with near-native performance. Wasm is designed to run alongside JavaScript, allowing developers to leverage both languages in a single web application and combining the performance benefits of WebAssembly with the flexibility of JavaScript. A WebAssembly implementation is usually embedded into a host environment (embedder), which specifies how modules are loaded, how imports are supplied (including definitions on the host side), and how exports can be utilized.

WebAssembly has two concrete representations: the binary format and the more human-readable text format, called ”wat”. Both representations map to a common structure, described in the form of an abstract syntax.

The text representation of Wasm binaries (wat) can be visualized directly from the browser, which facilitates and speeds up the reverse engineering and debugging processes.

```
(module
  (func $main
    i32.const 7 ;; Push the number 7 on the stack
    i32.const 3 ;; Push the number 3 on the stack
    i32.add ;; Adds the two numbers, leaving 10 on
        the stack
    drop ;; Remove the value at the top of the stack
  )
  (start $main)
)
```

Listing 1: WebAssembly function - wat format.

Listing 1, showcases the code snippet of a simple Wasm function that pushes two values on the stack and sums them up, subsequently removing the result from the stack.

3 RELATED WORKS

Two main works explore C attacks in Wasm presenting PoCs. One (Lehmann et al., 2020), in which the authors identify three situations from which an attack within a Wasm module can be constructed: obtaining a write primitive, overwriting data, and triggering unexpected behaviour. Another (McFadden et al., 2018) offers a broader view of the potential risks of compiling code from memory-unsafe languages to WebAssembly.

An attack that has not yet been explored thoroughly in current works is format strings. Lehmann et al. briefly mention this issue, whereas McFadden et al. offer a PoC demonstrating successful exploitation to leak data from the stack. Despite this success, their attempts to achieve a write primitive were unsuccessful due to JavaScript raising an exception, highlighting the need to explore its exploitation process. In Subsection 4.2, we provide a PoC showing how to successfully exploit this vulnerability to obtain arbitrary writing in linear memory.

Current works also highlight two critical security problems of WebAssembly modules, which can simplify the exploitation process compared to traditional architectures. Firstly, the entirely deterministic approach to memory allocation, specifically the absence of Address Space Layout Randomization (ASLR), significantly simplifies the process of executing targeted modifications to data in the heap. Therefore, even a linear stack-based overflow, if long enough, can lead to data corruption in the heap. Secondly,

WebAssembly has no mechanism that makes data immutable in linear memory, making ”constant” data possibly modifiable, including non-scalar constants and string literals.

Many other attacks are described in the papers above, such as Integer Overflow, Stack Overflow, Heap Metadata Corruption, redirecting indirect calls, injecting code into the host environment, or overwriting application-specific data. We encourage the readers to delve into the aforementioned papers to gain a comprehensive understanding of these vulnerabilities.

4 METHODOLOGY

In this section, we will describe our methodology to create vulnerable code for various C vulnerabilities and their subsequent exploitation.

Table 1: Overview of Proof of Concepts implemented. The headings are Client-Side (CS), Server-Side (SS), Data Leak (DL), Data OVerwrite (DO), Unexpected Behaviour (UB). The first column has Buffer Overflow (BOF), Format String (FS), Use After Free (UAF), Improper Array Index Validation (IAIV), Integer OVerflow (IO), Redirecting Indirect Calls (RIC).

Vulnerability	CS	SS	DL	DO	UB
BOF → XSS	✓	-	-	✓	-
BOF → RCE	-	✓	-	✓	-
FS	✓	✓	✓	✓	-
FS → XSS	✓	-	✓	✓	-
UAF	-	✓	-	✓	-
IAIV	-	✓	-	✓	-
IO	-	✓	✓	-	-
RIC	✓	✓	-	-	✓

4.1 Buffer Overflow

The objective of this PoC was to build an application vulnerable to a stack-based buffer overflow (BOF), enabling an attacker to gain an out-of-bounds write primitive and perform an XSS or RCE attack. The application asks the user for input, handling it in an unsafe way that introduces the buffer overflow vulnerability. Furthermore, the user's input is used inside the function `EM_ASM()`, which allows embedding inline JavaScript Code inside the C language. This function can pave the way to new security issues, especially if combined with the `eval` function. If an attacker manages to gain control over the arguments passed to `EM_ASM()` (and therefore to `eval`) they could potentially inject malicious scripts, leading to XSS on the client and RCE on the server. Notably, the impact of RCE is even more critical, allowing an attacker to inject arbitrary code to execute on the server.

4.1.1 Proof of Concept Overview

The PoC features an input field that requests the user's name. This input is subsequently passed to a C function called `greetings`, detailed below:

Listing 2: greetings Function - PoC for buffer overflow

```
void greetings(char* name){
    char sys_cmd[70] = "document.write('<h1>Welcome to
        WebAssembly!</h1>')";
    char greet[20];

    strncpy(greet, name, strlen(name));

    EM_ASM({
        eval(UTF8ToString($0));
        document.write("<h2>Hello, " + UTF8ToString($1) +
            "</h2>");
    }, sys_cmd, greet);
}
```

This function uses `strncpy` to copy user input to a buffer named `greet` on the stack. Stack-based buffer overflow occurs because the input length is used as `strncpy`'s third parameter, potentially overwriting adjacent memory if the input exceeds the buffer size. The function doesn't limit copied characters to the `greet` buffer size. Another string, `sys_cmd`, stores JavaScript code passed to `EM_ASM()`. Overwriting `sys_cmd` allows an attacker to inject arbitrary JavaScript code, possibly performing a Cross-Site Scripting attack. For the RCE BOF, the application has the same structure. There are only two differences from the client-side PoC: `sys_cmd` calls the `console.log` function instead of `document.write`, and the same happens inside the call to `EM_ASM`.

4.1.2 Exploitation

Our attack strategy focused on finding the length of the buffer and then overwriting the content of the variable `sys_cmd`, the argument of the eval function. Supposing we are an attacker that does not have access to the buffer length, we can send a long sequence of characters (e.g., the 40 characters string `aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaa`) with the aim of triggering an error that will reveal the size of our buffer. The previous sequence will trigger a `iaaajaaa is not defined` error, confirming that the first 32 characters had filled `greet` and the following 8 overwritten `sys_cmd`.

With this information, it's possible to craft a payload consisting of a 32-character sequence followed by JavaScript code to inject and execute. The payload below injects the code `alert('XSS!')` inside the WebAssembly module, triggering its execution:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAalert('
XSS!');
```

An essential observation regarding this vulnerability is that stack-based buffer overflows can also compromise data in the heap, requiring just a slightly larger payload depending on the size and the module's structure. This aspect makes attacks more feasible compared to the more complex scenarios in traditional x86-64 architectures.

In a similar manner, we can craft a payload that exploits a buffer overflow on a server-side application with a payload such as `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAconst{execSync}=require('child_process');console.log(execSync('ls').toString());//`. In this case we execute the `ls` command as a simple proof of concept. The choice of `ls` was purely illustrative, meant to highlight the potential for executing arbitrary system commands through the exploited vulnerability.

4.2 Format String Vulnerability

This section showcases two Proofs of Concept highlighting the risks associated with the format string vulnerability in WebAssembly, both in client-side and server-side environments. Additionally, we compare the differences between the two versions of Emscripten employed, finding important distinctions in the memory's structure, which led to the creation of two distinct payloads, one for each version. The format string vulnerability is introduced by bad coding practices, where the user's input is passed directly to the `printf` function, like `printf(user_buffer)` instead of `printf("%s",user_buffer)`. If an attacker injects format specifiers like `%p` or `%x` inside the buffer, the `printf` function will print values found on the stack, starting from the location of the `printf` pointer, therefore leaking data from memory. The number of values leaked corresponds directly to the number of format specifiers provided by the attacker. In addition, there exists a format specifier (`%n`) that, if leveraged, can give attackers arbitrary write to memory. The intended usage of this format specifier is to write the number of characters printed so far on the standard output into a variable. However, if a variable is not provided, the `printf` function will write such value into a memory address placed at a specific stack offset starting from the `printf` pointer. To direct this write operation to a particular offset "x", the `%x$n` syntax comes into play. For instance, a payload like `ABC%3$n` would write the value 0x3 (i.e. the length of "ABC") to the address contained in the third element from the stack starting from the `printf` pointer if such address is valid. Figure 1 showcases this scenario.

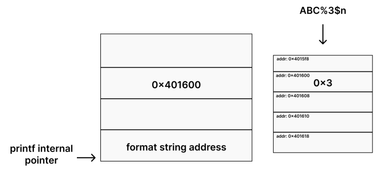

Figure 1: Example of writing data in memory using `%n`.

4.2.1 Proof of Concept Overview

The application (the PoC code can be found on our repository) displays an input field that asks the user for a password, which is then passed as an argument to a function named `check_password`, where it is compared with the correct password stored in the stack. If the user inserts the wrong password, then the program exposes itself to a format string vulnerability by directly passing the user's input (`pass`) to the `printf` function (`printf(pass)`). When a WebAssembly code that is compiled from C calls the `printf` function, it automatically redirects this output to the `console.log` function in JavaScript, allowing us to see the output.

This PoC aims to create an exploit using the format string vulnerability, overwriting the password stored in the stack. Specifically, we aim to overwrite the current password ("supersecretpassword") with the letter "B". Our experiments were conducted using Emscripten version 3.1.6 on the client-side application and Emscripten 3.1.52 on the server-side application (Node.js).

4.2.2 Exploitation

We initiated the client-side exploitation focusing on leaking data from memory. By crafting a payload filled with `%p` format specifiers, each separated by the "|" character for readability. As Figure 2 shows, the browser's console leaked stack values, among which we find we find the correct password, represented in hexadecimal.

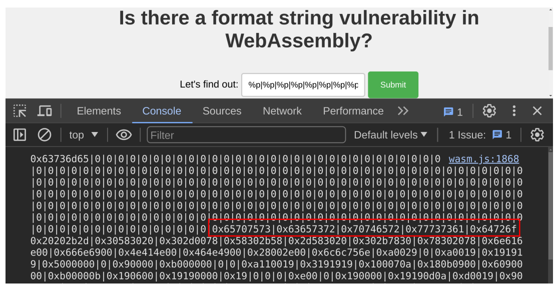

Figure 2: Format String - Arbitrary Read (the highlighted values are the hex representation of the password).

Using the WebAssembly Binary Toolkit (WABT), we converted our WebAssembly module into the

WebAssembly Text (WAT) format through the wasm2wat tool. This conversion allowed us to examine the internal memory layout of the module, discovering that the password was stored at the memory offset 1024. The next phase of our attack strategy focused on obtaining a write primitive, achievable through the format specifier `%n`. To overwrite the password at the address 1024, we first need to store the value 0x400 (1024 in hexadecimal) into a known offset from the `printf` pointer. To achieve that, we craft a payload composed of 1024 characters followed by `%2$n|%p|`: given that the offset 2 initially contains the value 0 (Figure 2), this payload will write the value 0x400 at address 0.
The final step involved writing the ASCII value of "B" (0x42 in hexadecimal or 66 in decimal) to address 1024. This is achieved with a payload formed by a 66-character sequence followed by `%n`, thereby writing 0x42 to the address found at the first offset (which is 1024), effectively overwriting the password with the letter "B".
After executing this attack and deploying the crafted payloads, we can bypass the password check by submitting the character "B".

The exploitation process for the server-side application is similar to the client-side, with two key differences. Firstly, the output of the 'printf' function is shown on the server console, not in the browser. This prevents attackers from directly leaking stack data or accessing the WebAssembly module to locate the password's offset. However, overwriting memory is still possible, and attackers could perform a "blind" attack to find the correct offset via brute force. The second difference is related to the Emscripten version used. To run a Node.js application that loads Wasm modules, we needed to upgrade to Emscripten release 3.1.52. After recompiling the C code and converting the Wasm module to the readable WAT format, we observed a significant shift in the password's offset from 1024 to 65536, requiring a much larger payload for an attack.

This information enabled us to craft a payload capable of writing the value 65536 (0x10000 in hexadecimal) into the first offset, allowing us to overwrite the password at the correct offset. Given the substantial size of this payload, an attacker might encounter buffer size constraints. A workaround for this issue involves using the format specifier `%65536c`, which prints the specified amount of characters without exceeding buffer limits.

Contrary to previous works (McFadden et al., 2018), we successfully exploited the format string vulnerability within the WebAssembly environment, managing to overwrite sensitive data in memory. The predictability of memory address locations in WebAssembly, due to the lack of Address Space Layout Randomization (ASLR), considerably facilitates the exploitation process. In contrast, exploiting environments with ASLR is more challenging due to the randomized memory addresses. This highlights the importance of addressing memory layout predictability in WebAssembly's modules.

Moreover, we observed that including the `%p` specifier is crucial for the payload `%2$n|%p` to successfully write to memory. Although this payload aims to write to memory using the `%n` specifier, the `%p` specifier, typically used for reading data, appears to be necessary for the write operation to proceed. Omitting the `%p` specifier results in the payload failing to write to memory. The exact reasons behind this remain an area for further investigation, highlighting an intriguing aspect of format string vulnerabilities within WebAssembly.

4.3 Use after Free

In this PoC, we present a server-side application exhibiting a Use After Free (UAF) vulnerability, highlighting the dangers associated with heap management errors when developing WebAssembly applications.

4.3.1 Proof of Concept Overview

Before diving into the details of the front-end side of the application, we focus on a snippet of the vulnerable C code, which can be seen in Listing 3.

Listing 3: Code of the PoC for the Use After Free vulnerability

```
typedef struct {
  char a[30];
  char flag[5];
} object;

object *x;

bool check_password(char* pass){
    return (strcmp(pass, x->flag) == 0);
}

void init() {
    x = malloc(sizeof(object));
    strncpy(x->flag, "wasm", 5);
}
```

This code snippet defines a structure named "object", which has a size of 35 bytes. The `init` function dynamically allocates memory for an "object" instance, storing its address in the variable "x" and initializing the "flag" field with the string "wasm". However, the `check_password` function compares this "flag" field with user input, potentially after its memory has been freed, thereby introducing a Use After Free (UAF) vulnerability into the application.

Additionally, the function `free_memory` is designed to deallocate the memory assigned to the variable x, resulting in x becoming a dangling pointer. Meanwhile, the `alloc_object` function allows allocating heap memory of any specified size using `malloc`.

4.3.2 Exploitation

This attack aims to modify the existing password by overwriting the data in the ”flag” field of the ”x” variable. The initial step involves deallocating the heap chunk allocated for ”x”, by invoking `free_memory`. Subsequently, a call to `malloc` requesting a chunk of a similar size as the one just freed results in allocating that same memory space, giving us access to the released memory. To exploit this, we allocate a new chunk of 35 bytes, identical to the size of ”x”, containing the payload allowing us to overwrite the password. We craft a payload composed of a 30-character sequence that fills the ”a” field buffer, followed by the string ”pass”, which will be the new password inside the ”flag” field.

Summarizing, the exploitation process is composed of 3 steps: (i) Free the chunk of ”x”; (ii) Allocate a 35-byte object containing the payload `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAApass`; (iii) Authenticate using the new password ”pass”.

We demonstrated that, if leveraged, this vulnerability can cause damage within the WebAssembly environment. Depending on the application’s structure, this vulnerability can introduce many other risks, such as possible data leaks or memory corruption. Another common vulnerability in heap memory is the ”Double Free”, which we leave for future research within WebAssembly.

4.4 Improper Array Index Validation

When an application fails to properly check or validate the indices used to access elements within an array, it can open the way to unauthorized access to sensitive data, corruption of data, crashes, and execution of arbitrary code.

4.4.1 Proof of Concept Overview

The application provides users with two functionalities: verifying their entered password and modifying a character in their password by choosing an index and a new character. However, the application uses the provided index to access the user’s password, stored in heap memory along with the admin password and another password called `unreachable_password`, without any checks to ensure the index is within valid boundaries.

Listing 4: Code snippet of the PoC for Improper Array Index Validation

```
char unreachable_password[30] = "unreachablepassword";
char user_password[30] = "thisisyourpassword";
char admin_password[30] = "supersecretpassword";

void change_user_password_character(int index, char
     new_char){
   user_password[index] = new_char;
}
```

Listing 4 displays the code snippet where the three passwords are allocated in the heap memory, alongside the function that allows modifying the user’s password.

4.4.2 Exploitation

Our experiments aimed to access memory indices outside the valid range of the user’s password, which is between 0 and 29, with the intent to overwrite adjacent data in memory.

Our initial approach involved accessing indices greater than 29 to potentially overwrite the admin’s password stored after the user’s password in memory. We successfully overwrote the admin’s password characters starting from index 32. This allowed us to modify the password and log in as admins. A strategy is to change the character at index 32 to ’A’ and set the next character to a null byte, enabling login with just ’A’. The second experiment involved accessing indices below 0, aiming at overwriting the `unreachable_password`. This experiment was unsuccessful because accessing negative indices in WebAssembly will trigger a ”memory access out of bounds” runtime error. Unlike in languages like C, where accessing negative indices will not necessarily lead to a crash in the program, WebAssembly throws this error to prevent unsafe memory operations.

This PoC highlighted the critical impact of unchecked index boundaries, which enabled overwriting sensitive data like the admin’s password. However, WebAssembly’s protection against negative index exploitation, marked by the ”memory access out of bounds” error, is a key security feature. Despite this, our next Proof of Concept outlines a scenario that, in a way, bypasses these strict memory access controls.

4.5 Integer Overflow

In this Proof of Concept, we explore the Integer Overflow vulnerability within a WebAssembly application. Integer Overflow occurs when an integer variable exceeds the maximum size of the integer type, causing unexpected behaviour. In most systems, unsigned integers are represented by 4 bytes (32 bits), meaning that the maximum representable value is

4,294,967,295 ($2^{32} - 1$). Such value, in the C language, is stored within the macro "INT_MAX". Incrementing by 1 this maximum value, the integer overflows and wraps around to 0 due to the limited number of bits. This PoC aims to demonstrate how such an overflow can be exploited to compromise the application's integrity or security.

4.5.1 Proof of Concept Overview

Listing 5: Code snippet of the PoC for Integer Overflow.

```
char supersecretpassword[5] = "pass";
char users[5][20] = {
    "guest_user",
    "guest_password",
    "admin",
    "admin_password"
};

char* get_array_element(int index) {
    printf("Accessing index: %d\n", index);
    return users[index];
}
```

The C code in Listing 5 initializes a password in heap memory, followed by an array of five strings. The application prompts the user for an index, restraining the input to values greater than 0 and not equal to 3. This check is performed inside the Node.js code.

4.5.2 Exploitation

We begin our exploration by providing the value INT_MAX + 1 to the application, causing the integer variable to overflow and wrap to 0, enabling us to access and print the string in the index 0. Subsequently, we input the number INT_MAX + 4, obtaining access to the "forbidden" index 3, and printing the content of the admin's password. This PoC demonstrates the potential of integer overflow to compromise application security. This finding underscores the need for rigorous boundary checks in WebAssembly application development. The testing phase of this PoC led to a notable finding: accessing the array of strings using a negative index does not trigger the "memory access out of bounds" error, allowing the leakage of the password instantiated before the array of strings.

4.6 Redirecting Indirect Calls

Indirect calls in WebAssembly are used to implement function pointers and virtual functions. The `call_indirect` instruction pops a value from the stack, using it as an index in the `table` section and invoking the specified function if the signature matches.

The last PoC presented in this paper demonstrates how an attacker could change the normal control flow of a WebAssembly application by redirecting indirect calls.

4.6.1 Proof of Concept Overview

Similar to previous PoCs showcased in this paper, this application prompts the user for a password.

Listing 6: Code of the PoC for Redirecting Indirect Calls - Server-Side.

```
bool check_password(char* input){
    bool (* login_handler)(void) = 0;
    char password[10];

    strcpy(password, input);

    if (login_handler == 0){
        if (strcmp(password, "secretpass") == 0)
            login_handler = (bool (*)(void))&admin_panel;
        else
            login_handler = (bool (*)(void))&user_panel;
    }
    return login_handler();
}
```

Listing 6 illustrates the `check_password` function in C, highlighting its vulnerability. This function employs a function pointer named `login_handler`, which is conditionally set to the address of either the `admin_panel` or `user_panel` function, based on the user's password input. The code introduces a buffer overflow vulnerability at line 5, caused by using the `strcpy` function, which lacks boundary checks.

The conditional assignment of function pointers, as seen in the code, results in indirect function calls within the WebAssembly environment.

4.6.2 Exploitation

To redirect the program's control flow, we exploited the buffer overflow vulnerability by overwriting the `login_handler` pointer with the address of the `admin_panel` function, effectively bypassing the password verification process.

The payload we designed exploits the buffer overflow by starting with ten "A" characters to fill the user's buffer and reach the function pointer, followed by `String.fromCharCode(1)`. This last byte overwrites the '`login_handler`' with "1", which is the address of `admin_panel`, effectively redirecting the program's execution to bypass the password check.

Low numerical addresses in WebAssembly, such as the `admin_panel` function's address located at address 1, increase security risks. Unlike traditional systems where ASLR complicates the exploiting process by randomizing addresses, WebAssembly's low and deterministic addresses simplify crafting exploits, highlighting the need for stronger security measures in Wasm.

Table 2: Comparison of Vulnerability Coverage between our work and the state of the art.

	Lehmann et al.			McFadden et al.			Our work		
Vulnerability	Mention	Explain	PoC	Mention	Explain	PoC	Mention	Explain	PoC
Buffer Overflow (BOF)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Heap Metadata Corruption	✓	✓	✓	✓	-	-	✓	-	-
BOF → XSS	✓	✓	✓	✓	✓	✓	✓	✓	✓
BOF → RCE	✓	✓	✓	-	-	-	✓	✓	✓
Integer Overflow	✓	-	-	✓	✓	-	✓	✓	✓
Format String	✓	-	-	✓	✓	✓*	✓	✓	✓
Format String → XSS	-	-	-	-	-	-	✓	✓	✓
Format String → RCE	-	-	-	-	-	-	✓	✓	-
Use After Free	✓	-	-	✓	-	-	✓	✓	✓
Double Free	✓	-	-	✓	-	-	✓	-	-
Overwriting Constant Data	✓	✓	-	-	-	-	✓	-	-
Redirecting Indirect Calls	✓	✓	-	✓	✓	-	✓	✓	✓
Redirecting Indirect Call → XSS	-	-	-	✓	✓	✓	-	-	-
Redirecting Indirect Call → RCE	-	-	-	✓	✓	✓	-	-	-
Improper Validation of Array Index	-	-	-	-	-	-	✓	✓	✓

*McFadden et al. provided a PoC for the format string vulnerability without achieving arbitrary writing.

5 CONCLUSIONS

Our findings indicate a crucial need to implement stronger security measures to protect WebAssembly applications against potential exploits. Compiling WebAssembly from memory-unsafe languages like C introduces many security concerns, possibly enabling attackers to use traditional binary exploitation techniques to attack web applications. Such attack techniques could significantly impact real-world scenarios, particularly when developers introduce vulnerabilities by reusing code from other projects without conducting adequate security checks. Future research may focus on the study and development of renewed security measures inside WebAssembly compilers, possibly implementing traditional mitigation techniques such as ASLR or stack canaries. We also plan to expand our research towards more complex vulnerabilities and attack strategies and to examine how WebAssembly security may impact other technologies.

ACKNOWLEDGEMENTS

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program. This work was also supported by project "SUSTAIN - flexible Sensors for secUre and truSTed crowdsensing environmentAl applicatioNs", – CUP F25F21002720001 (D.M. 737/2021 - Interdisciplinary research initiatives on transversal topics for PNR). Both projects are funded by the European Union– NextGenerationEU.

REFERENCES

De Macedo, J., Abreu, R., Pereira, R., and Saraiva, J. (2022). WebAssembly versus JavaScript: Energy and runtime performance. In *2022 (ICT4S)*. IEEE.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN PLDI*. ACM.

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

Jazayeri, M. (2007). Some trends in web application development. In *Future of Software Engineering (FOSE '07)*. IEEE.

Lehmann, D., Kinder, J., and Pradel, M. (2020). Everything old is new again: Binary security of {WebAssembly}.

Massidda, E. (2024). https://github.com/manumassi/vuln_wasm.

McFadden, B., Lukasiewicz, T., Dileo, J., and Engler, J. (2018). Security chasms of wasm. *NCC Group Whitepaper*.

Ray, P. P. (2023). An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, 15(8).

Stiévenart, Q., De Roover, C., and Ghafari, M. (2022). Security risks of porting c programs to webassembly. In *Proceedings of the 37th ACM/SIGAPP SAC*. ACM.

Wang, W. (2021). Empowering web applications with webassembly: Are we there yet? In *2021 36th IEEE/ACM (ASE)*. IEEE.

Yan, Y., Tu, T., Zhao, L., Zhou, Y., and Wang, W. (2021). Understanding the performance of webassembly applications. In *21st ACM IMC*. ACM.

Zakai, A. (2011). Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA '11*.