Everything Old is New Again: Binary Security of WebAssembly

Daniel Lehmann*
Johannes Kinder‡
Michael Pradel*

* University of Stuttgart, Germany
‡ Bundeswehr University Munich, Germany
void vuln(char* src) {
    char buf[8];
    strcpy(buf, src);
}

- Fast, low-level, portable bytecode
- Support in browsers, Node.js, standalone VMs
- Compiled from C, C++, Rust, Go, ...
void vuln(char* src) {
    char buf[8];
    strcpy(buf, src);
}

Security?

Source program

WebAssembly binary

Native

• Virtual memory
• Stack canaries
• Control-Flow Integrity (CFI)
• ...
void vuln(char* src) {
    char buf[8];
    strcpy(buf, src);
}

"Data execution prevention and stack smashing protection are not needed by WebAssembly programs."

github.com/WebAssembly/design

"At worst, a buggy or exploited Web-Assembly program can make a mess of the data in its own memory."

Haas et al., PLDI 2017
Contributions

I. In-depth security analysis of WebAssembly
   • Linear memory
   • Mitigations

II. Library of attack primitives

III. Proof-of-concept exploits on three platforms

IV. Measurements on real-world binaries
Contributions

I. **As we go** security analysis of WebAssembly
   - Linear memory
   - Mitigations

II. **Example** attack primitives

III. Proof-of-concept exploits on one platform

IV. Measurements on real-world binaries
Attack Outline

1. Write Primitive

2. Overwrite Data

3. Malicious Action

Mitigations?

Buffer overflow on unmanaged stack

Stack Canaries

Unmapped Pages

Sensitive heap data

XSS in the browser
document.write(str)
Managed vs. Unmanaged Data

- Managed by VM: scalar variables, return addresses
  - `(local $l i32)`
  - `call $func`

- Unmanaged data in memory:
  - `malloc(...)`
  - Heap allocations
  - `const char* string = "..."`
    - Global data, e.g., string literals
  - `char array[10]`
  - `struct Type complex`
    - Arrays, structs
  - `void function(int* out)`
    - Address taken, e.g., out parameters

Unmanaged stack, used by 33% of all functions
Buffer Overflow – Native

Legacy code base

void vuln(char* src) {
    char buf[8];
    strcpy(buf, src);
}

Native stack, e.g., x86-64

Stack canary

return address

Overflow

debugger

9
Buffer Overflow – WebAssembly

Legacy code base

```c
void caller() {
    char other[8];
    vuln(src);
}

void vuln(char* src) {
    char buf[8];
    strcpy(buf, src);
}
```

- Unmanaged stack
- Managed data
- Overflow
- Return address
- $sp
Attack Outline

1. Write Primitive
2. Overwrite Data
3. Malicious Action

Mitigations?

Buffer overflow on unmanaged stack

Stack Canaries

Unmapped Pages

Sensitive heap data

XSS in the browser document.write(str)
Linear Memory

• Single 32-bit memory space
  • Contains all unmanaged data
  • No "holes", $\text{ptr} \in [0, \text{max\_mem}]$

• No page protections
  • No unmapped pages
  • Always writable

• No ASLR, fully deterministic
Attack Outline

1. Write Primitive
   - Buffer overflow on unmanaged stack

Mitigations?

2. Overwrite Data
   - Stack Canaries
   - Unmapped Pages
   - Sensitive heap data

3. Malicious Action
   - XSS in the browser
document.write(str)
std::string html = "<img...";
html += output + "">
document.write(html);

void pnm2png(char* input) {
    // CVE-2018-14550
}

C++ web application

Hea
alert(...)
Stack
AAAA...
Static

Stack-to-heap overflow
XSS in Browser: Demo
More Primitives...

1. Write Primitive
   - Stack-based buffer overflow
   - Stack overflow
   - Heap metadata corruption
   - Stack canaries
   - Unmapped pages
   - Safe unlinking

2. Overwrite Data
   - Heap data
   - Other stack frames
   - Constant data

3. Malicious Action
   - Browser: XSS
   - Node.js: exec()
   - WASI: fwrite()
   - Wasm CFI
   - Redirect calls
Stack → Heap Overwrite → XSS

1. Write Primitive
   - Stack-based buffer overflow
   - Stack canaries

2. Overwrite Data
   - Heap data
   - Other stack frames
   - Constant data
   - Unmapped pages

3. Malicious Action
   - Browser: XSS
   - Node.js: exec()
   - WASI: fwrite()
   - Wasm CFI
   - Safe unlinking
   - Redirect calls
   - Heap metadata corruption
Heap Overflow → Function Ptr → RCE

1. Write Primitive
- Stack-based buffer overflow
- Stack overflow
- Stack canaries
- Unmapped pages
- Safe unlinking
- Heap metadata corruption

2. Overwrite Data
- Heap data
- Other stack frames
- Constant data

3. Malicious Action
- Browser: XSS
- Node.js: exec()
- WASI: fwrite()
- Redirect calls
- Wasir CFI
Stack → String Literal → File Write

1. Write Primitive
   - Stack-based buffer overflow
   - Stack canaries
   - Unmapped pages

2. Overwrite Data
   - Heap data
   - Other stack frames
   - Constant data

3. Malicious Action
   - Browser: XSS
   - Node.js: exec()
   - WASI: fwrite()
   - Wasm CFI
   - Redirect calls

19
Everything Old is New Again: Binary Security of WebAssembly

Daniel Lehmann  
*University of Stuttgart*  
Johannes Kinder  
*Bundeswehr University Munich*  
Michael Pradel  
*University of Stuttgart*

**Abstract**

WebAssembly is an increasingly popular compilation target designed to run code in browsers and on other platforms safely and securely, by strictly separating code and data, enforcing types, and limiting indirect control flow. Still, vulnerabilities in memory-unsafe source languages can translate to vulnerabilities in WebAssembly binaries. In this paper, we analyze to what extent vulnerabilities are exploitable in WebAssembly binaries, and how this compares to native code. We find that many classic vulnerabilities which, due to common mitigations, are no longer exploitable in native binaries, are completely exposed in WebAssembly. Moreover, WebAssembly enables unique attacks, such as overwriting supposedly constant data or manipulating the heap using a stack overflow. We present a set of attack primitives that enable an attacker (i) to write arbitrary memory, (ii) to overwrite sensitive data, and (iii) to trigger unexpected behavior by diverting control flow or manipulating the host environment. We provide a set of vulnerable proof-of-concept applications along with complete end-to-end exploits, which cover three WebAssembly plat-
Summary

WebAssembly binary security

Manager vs. unmanaged data

Attack primitives and mitigations

PoCs on three platforms
Questions?

WebAssembly binary security

Managed vs. unmanaged data

Attack primitives and mitigations

PoCs on three platforms

mail@dlehmann.eu
michael@binaervarianz.de
johannes.kinder@unibw.de