Everything Old is New Again: Binary Security of WebAssembly

Daniel Lehmann* Johannes Kinder‡ Michael Pradel*

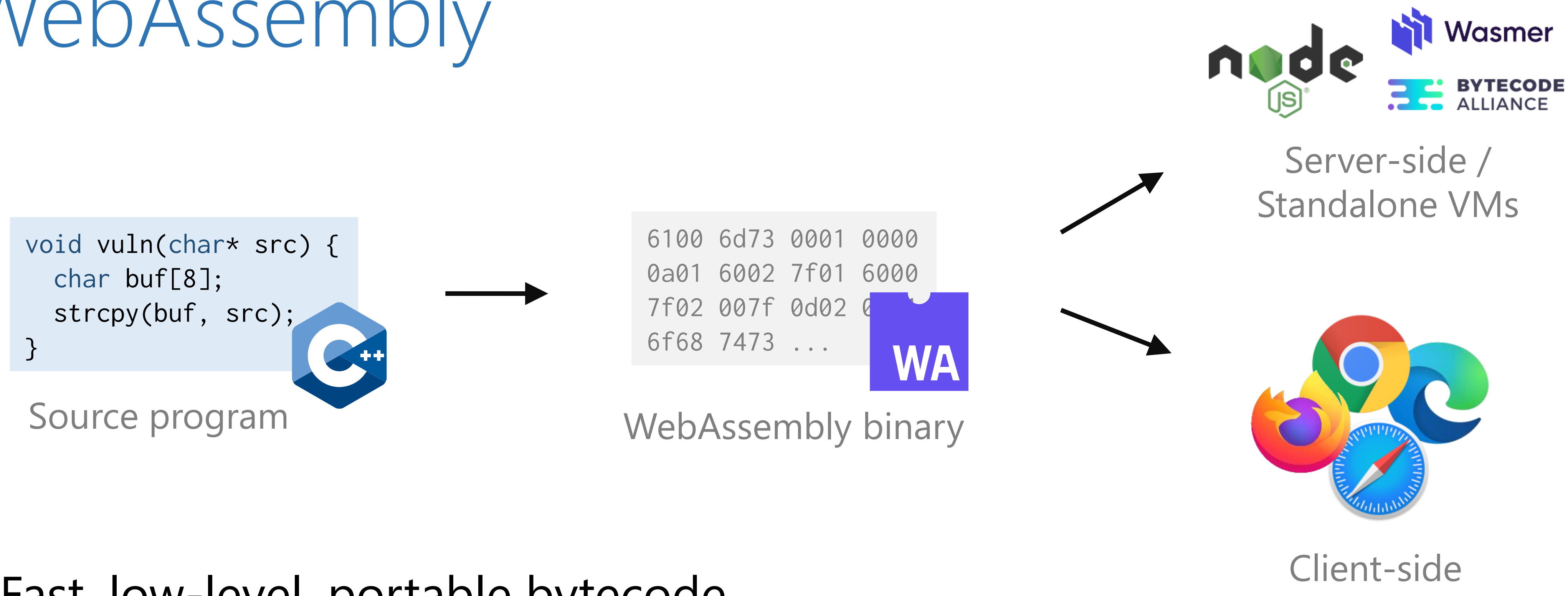
* University of Stuttgart
Germany

‡ Bundeswehr University Munich
Germany

University of Stuttgart
Germany

der Bundeswehr
Universität München

WebAssembly



- Fast, low-level, portable bytecode
- Support in browsers, Node.js, standalone VMs
- Compiled from C, C++, Rust, Go, …

Security?

AAAAAA
AAA...

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

Source program

```
6100 6d73 0001 0000
0a01 6002 7f01 6000
7f02 007f 0d02 0
6f68 7473 ...
```

WA

WebAssembly binary

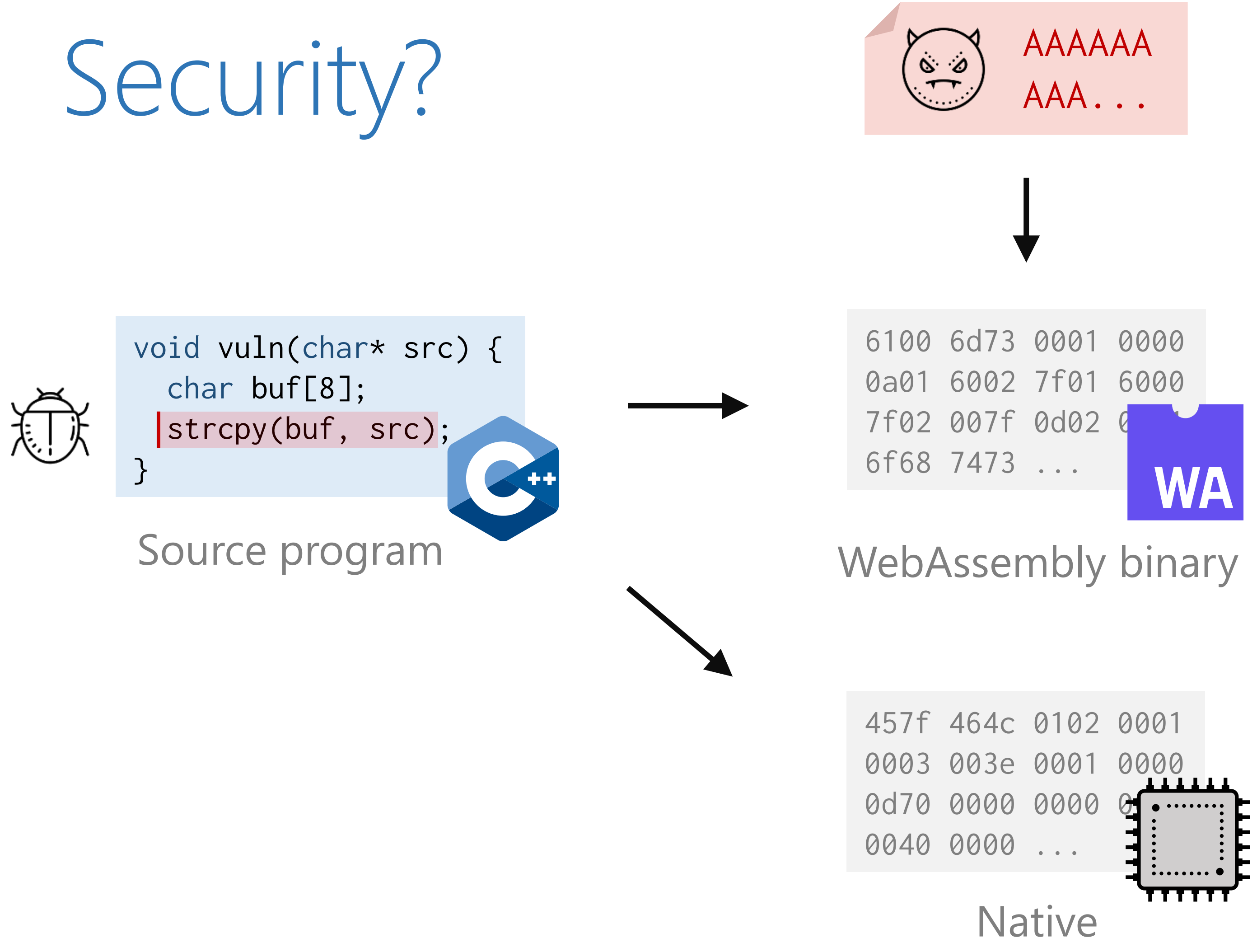
```
457f 464c 0102 0001
0003 003e 0001 0000
0d70 0000 0000 0
0040 0000 ...
```

Native

< ?

- Virtual memory
- Stack canaries
- Control-Flow Integrity (CFI)
- ...

Security?



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

github.com/WebAssembly/design

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

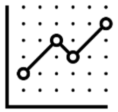
Haas et al., PLDI 2017

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

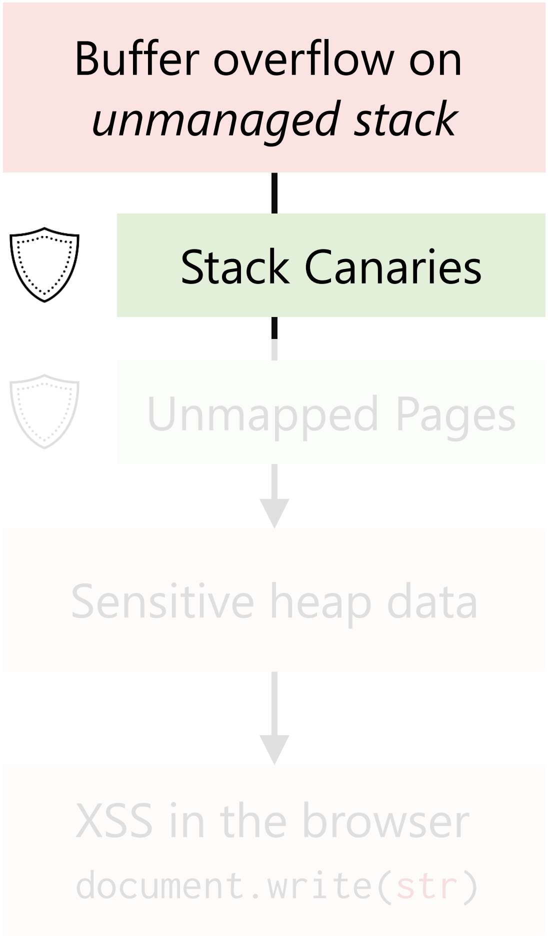
Attack Outline

1. Write Primitive

Mitigations?

2. Overwrite Data

3. Malicious Action



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)`

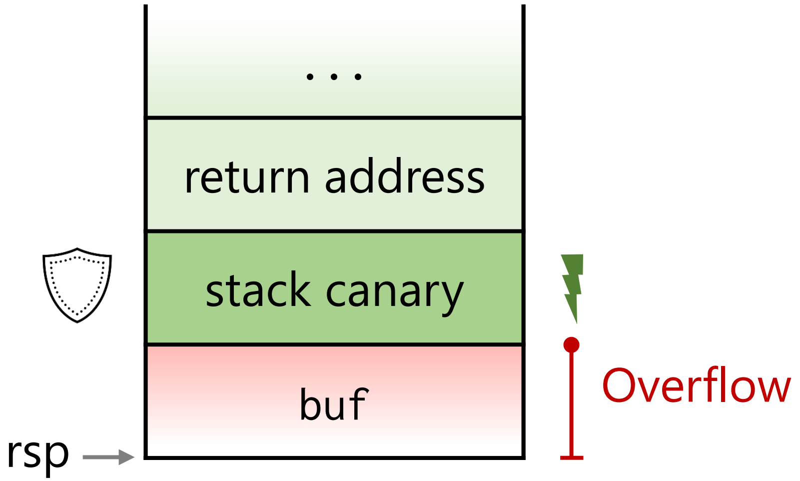
Address taken, e.g., out parameters

Unmanaged stack, used by 33% of all functions

Buffer Overflow – Native

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

Legacy code base



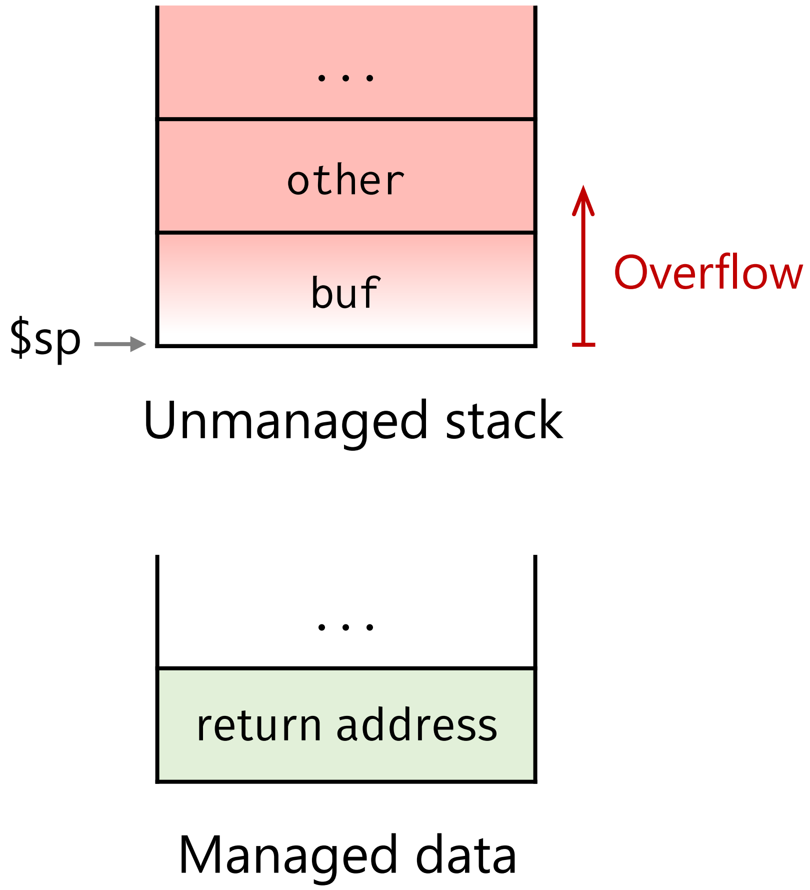
Native stack,
e.g., x86-64

Buffer Overflow – WebAssembly

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

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

Legacy code base



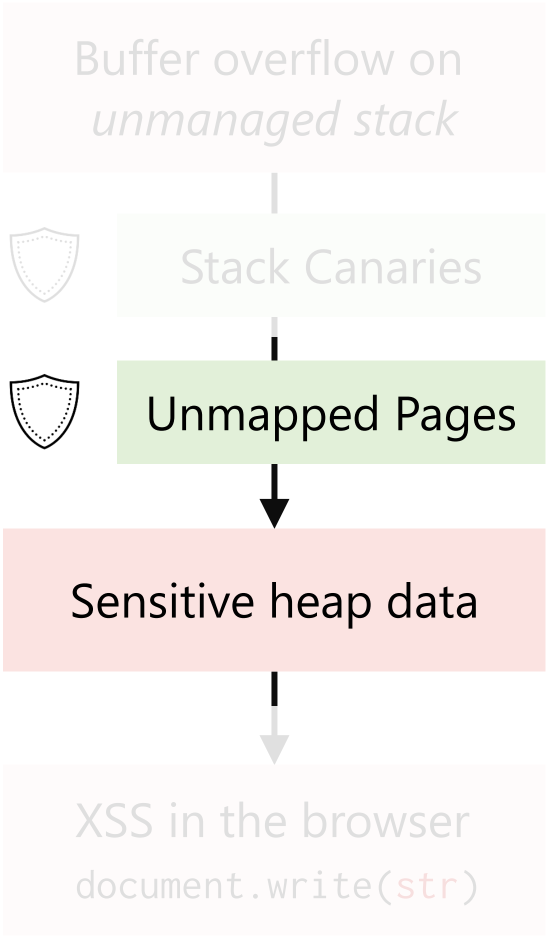
Attack Outline

1. Write Primitive

Mitigations?

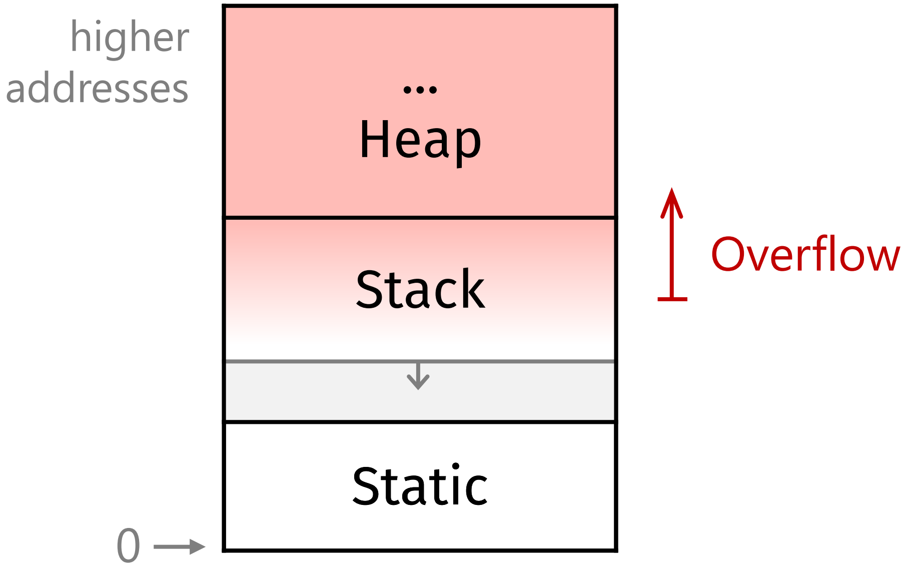
2. Overwrite Data

3. Malicious Action



Linear Memory

- Single 32-bit memory space
  - Contains all unmanaged data
  - No "holes", ptr ∈ [0, 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? — Stack Canaries; Unmapped Pages

2. Overwrite Data — Sensitive heap data

3. Malicious Action — XSS in the browser

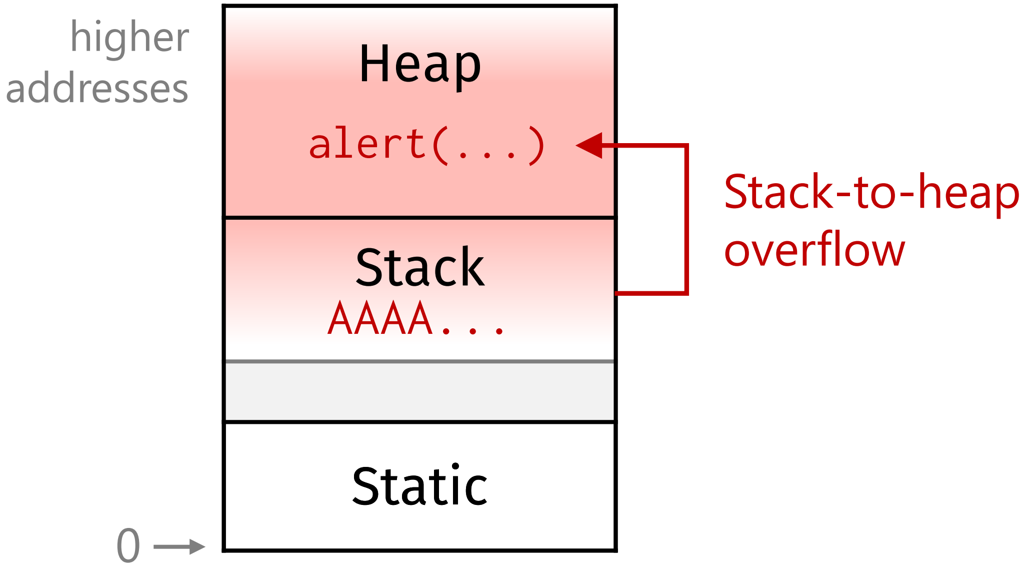
```
document.write(str)
```

XSS in Browser: Demo

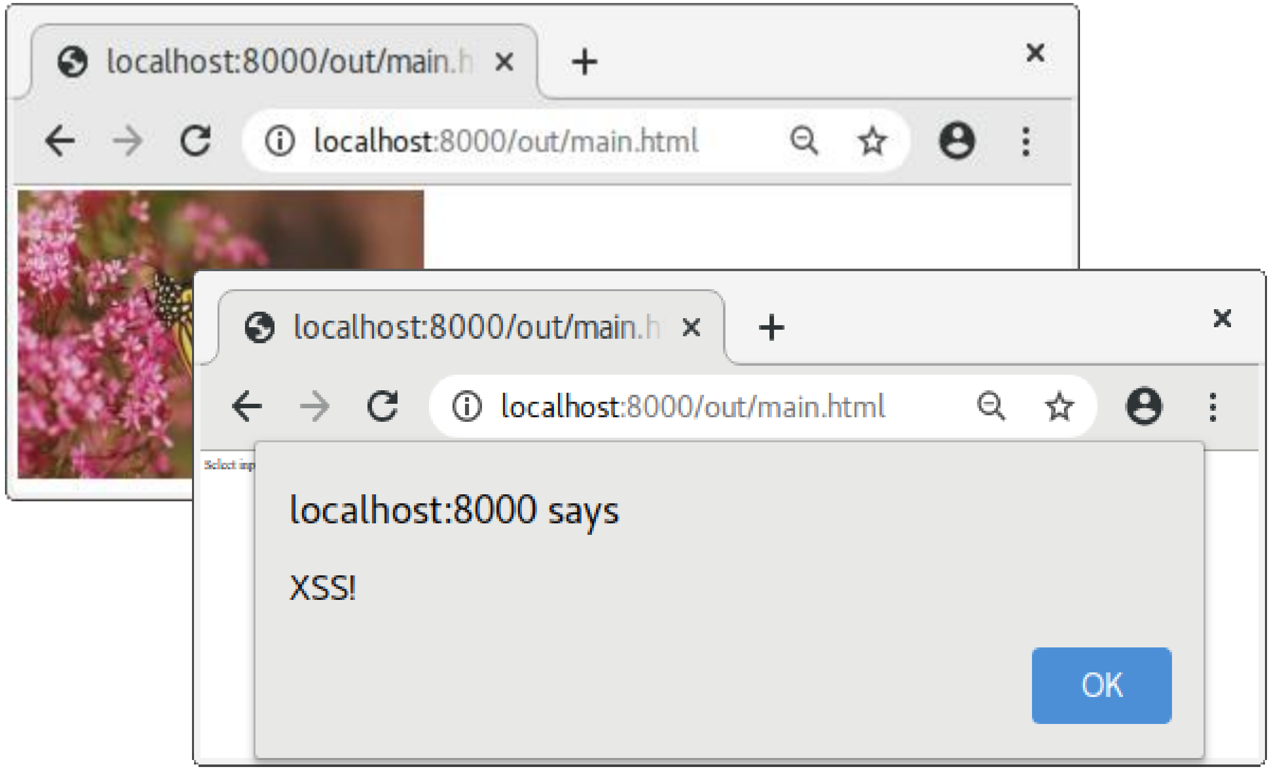
```
std::string html = "<img…";
pnm2png(input, output);
html += output + ">";
document.write(html);
```

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

C++ web application



XSS in Browser: Demo



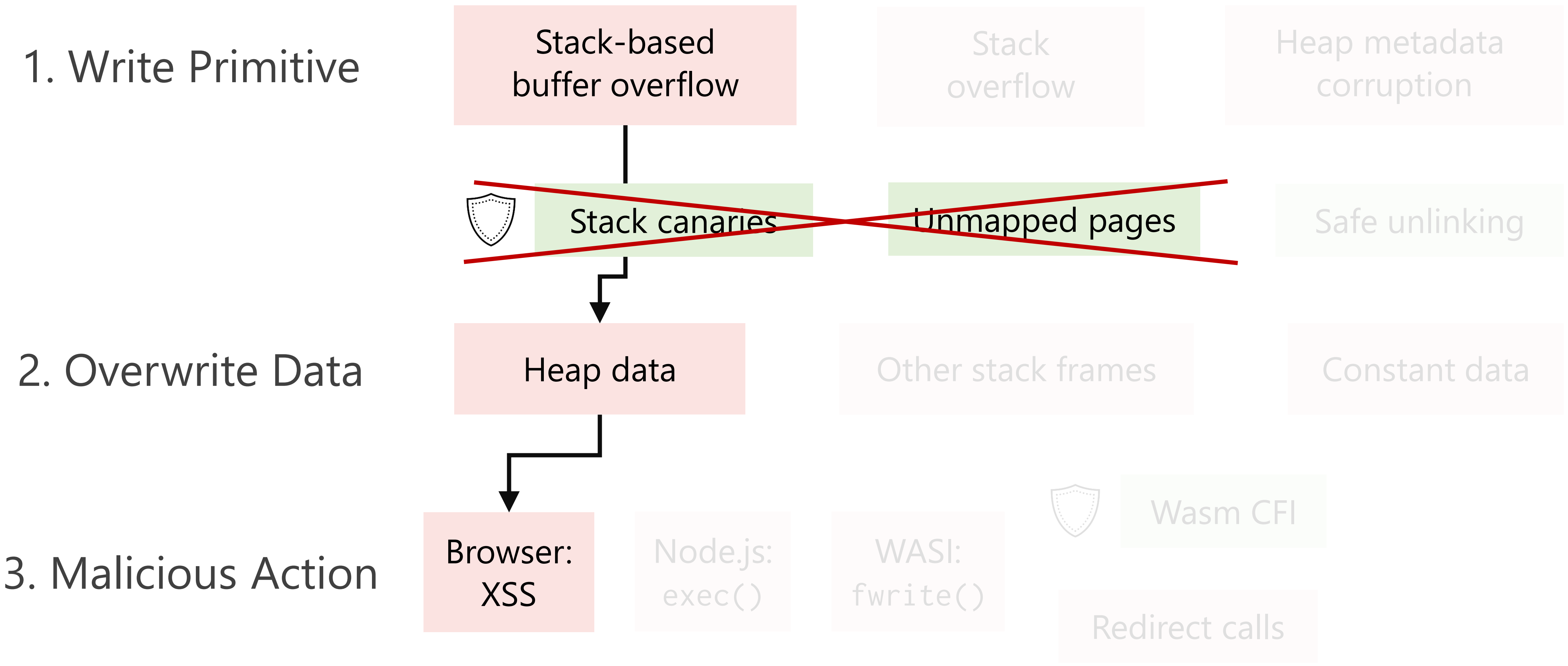
More Primitives…

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

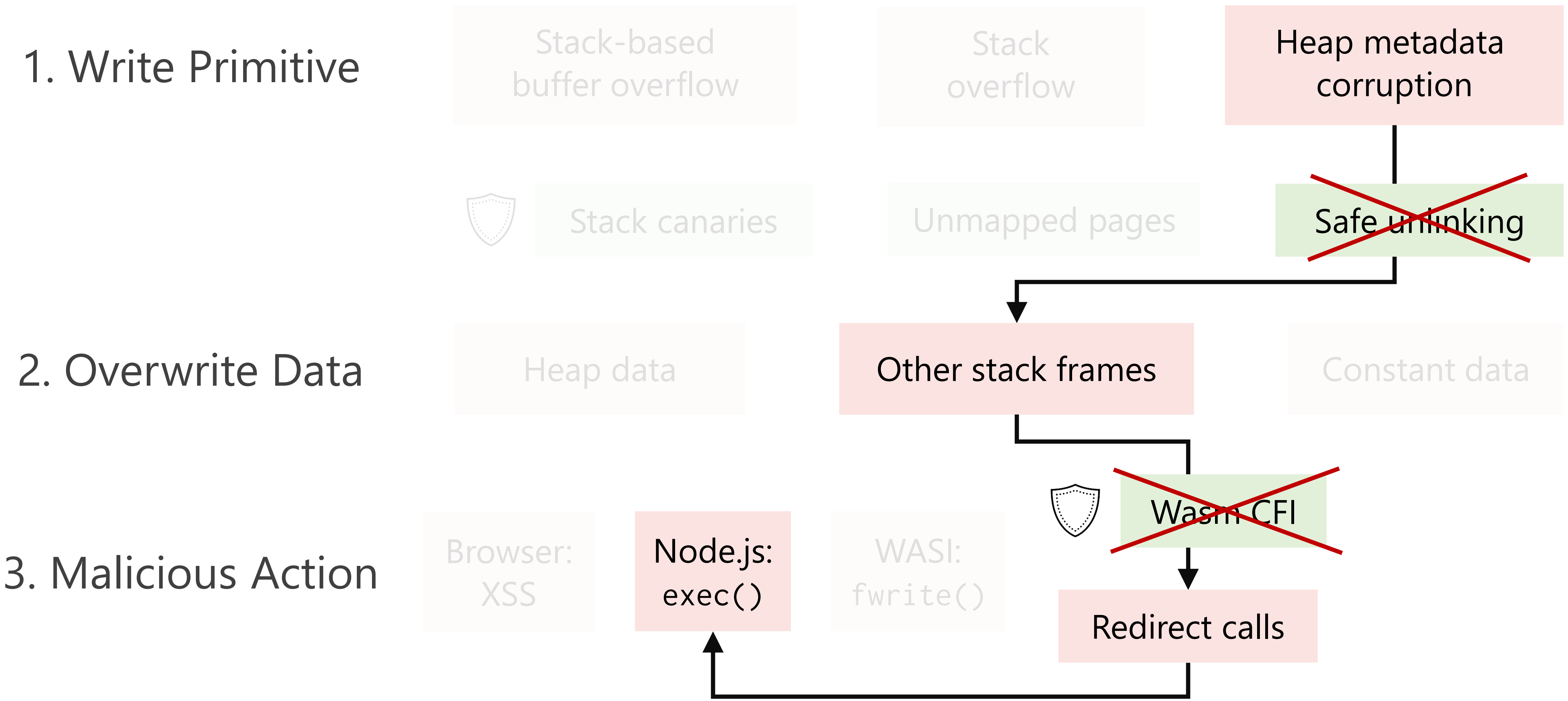
2. Overwrite Data
   - Heap data
   - Other stack frames
   - Constant data

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

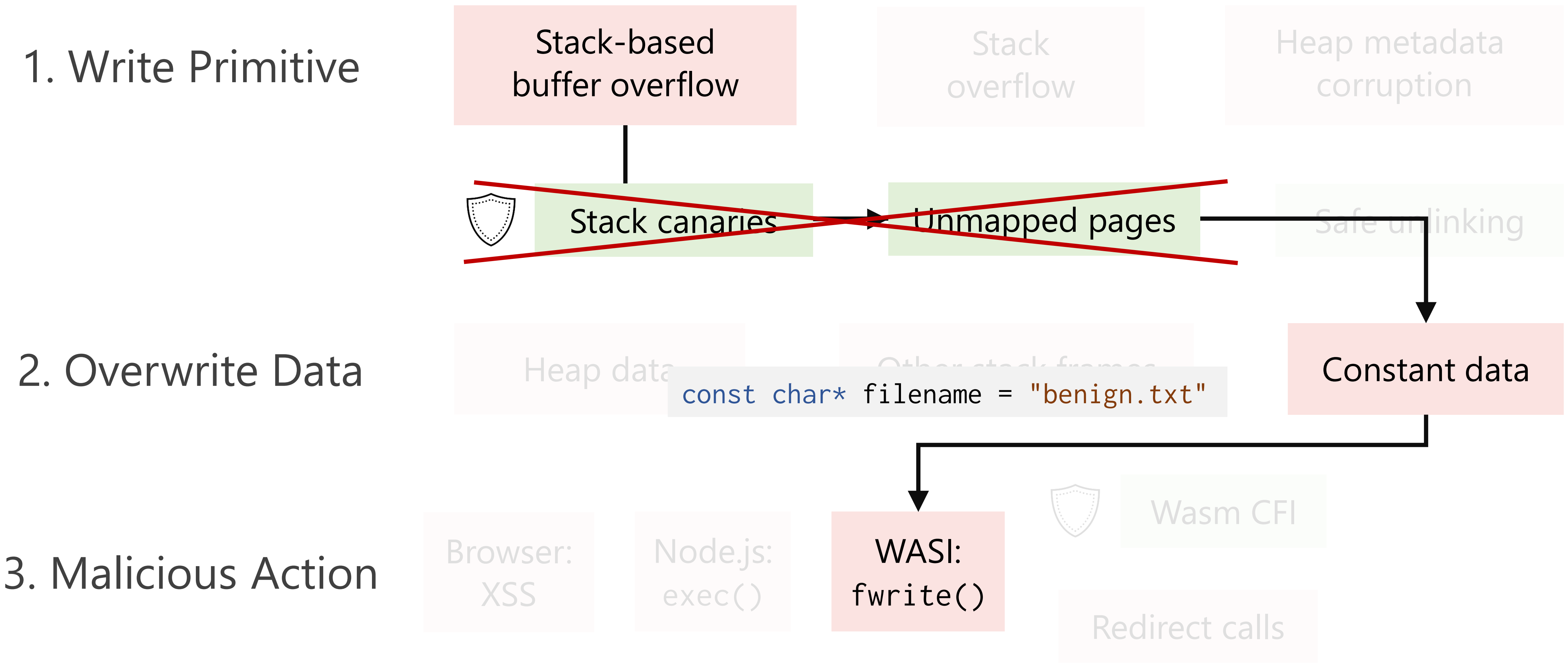
Stack → Heap Overwrite → XSS



Heap Overflow → Function Ptr → RCE



Stack → String Literal → File Write



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

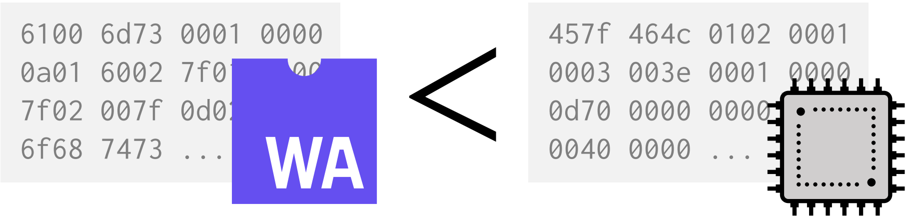
WebAssembly is an increasingly popular compilation target designed to run code in browsers and on other platforms safely and securely, by strictly separating code and data, enforcing types, and limiting indirect control flow. Still, vulnerabilities in memory-unsafe source languages can translate to vulnerabilities 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-

both based on LLVM. Originally devised for client-side computation in browsers, WebAssembly's simplicity and generality has sparked interest to use it as a platform for many other domains, e.g., on the server side in conjunction with Node.js, for "serverless" cloud computing [33–35, 64], Internet of Things and embedded devices [31], smart contracts [44, 53], or even as a standalone runtime [4, 23]. WebAssembly and its ecosystem, although still evolving, have already gathered significant momentum and will be an important computing platform for years to come.

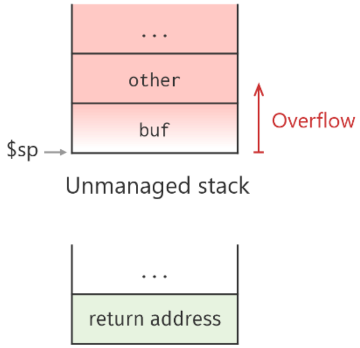
WebAssembly is often touted for its safety and security. For example, both the initial publication [32] and the official website [12] highlight security on the first page. Indeed, in WebAssembly's core application domains, security is paramount: on the client side, users run untrusted code from websites in their browser; on the server side in Node.js, WebAssembly modules operate on untrusted inputs from clients; in cloud computing, providers run untrusted code from users; and in smart contracts, programs may handle large sums of money.

There are two main aspects to the security of the WebAs-

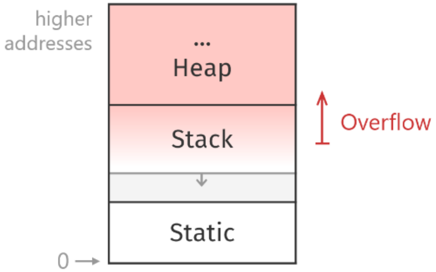
Summary



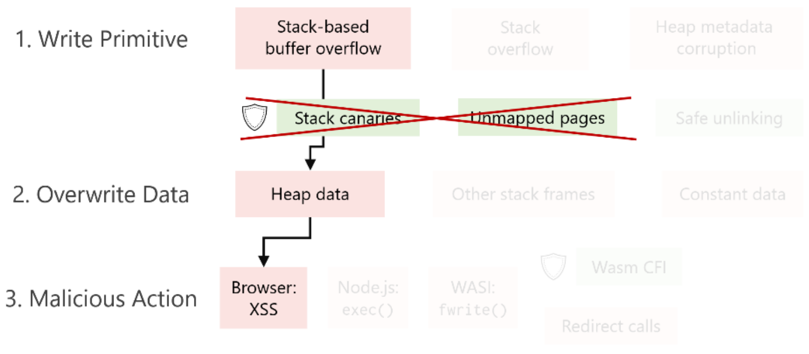
WebAssembly binary security



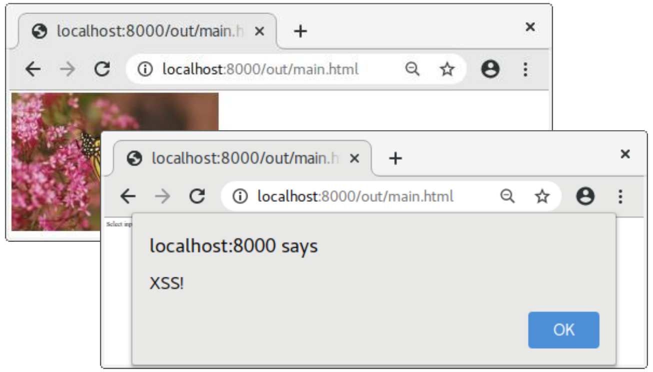
Managed vs. unmanaged data



Linear memory



Attack primitives and mitigations



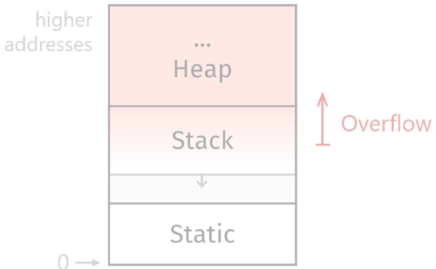
PoCs on three platforms

Questions?

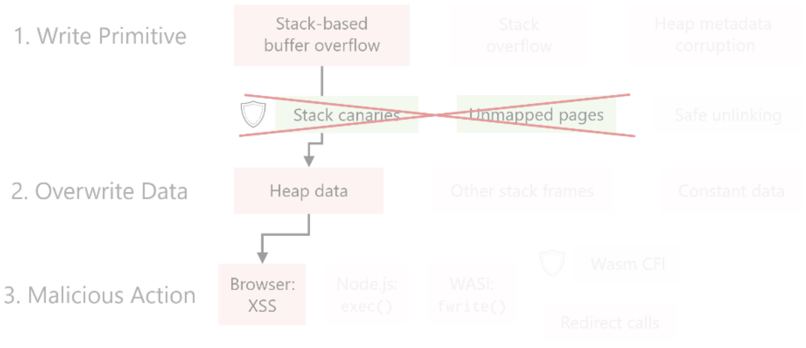
mail@dlehmann.eu
michael@binaervarianz.de
johannes.kinder@unibw.de

WebAssembly binary security

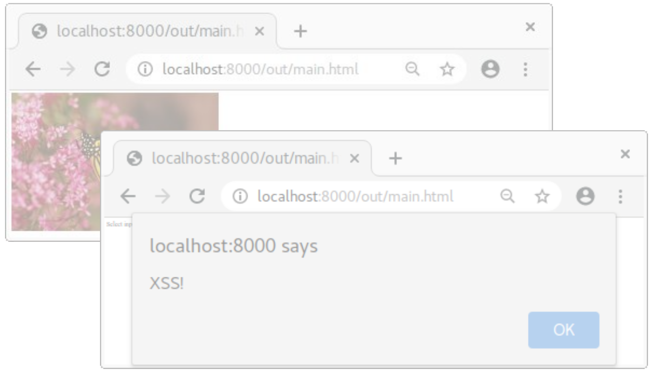
Managed vs. unmanaged data



Linear memory



Attack primitives and mitigations



PoCs on three platforms

small icons: icons8.com