CSCI 599: Software Engineering for Security

Project Update 2

April 19

Group II
Sara Baradaran, Jungkyu Kim, Ritu Pravakar, Yutian Yan

USC Viterbi School of Engineering

University of Southern California

Vulnerability Study in WebAssembly Runtimes/Virtual Machines

Presentation Outline

1. Project Overview
2. Data Collection
3. Selected Runtimes
4. Vulnerability Analysis & Reproduction

Project Overview

- The goal of this project is to analyze, categorize, and reproduce vulnerabilities detected in WebAssembly runtimes
- Understanding detected vulnerabilities can help developers know vulnerable code patterns, functions/APIs, and vulnerability root causes and avoid them when developing analogous software
- Since many vulnerabilities may share similar root causes, understanding root causes for known vulnerabilities may reveal undiscovered vulnerabilities
- Also, finding vulnerable products and the root causes of vulnerabilities can help developers to decide which products should be used for a specific purpose. For example, if a runtime does not validate the input WebAssembly files, the products that use it must validate the codes before passing them into the runtime

Data Collection

- We have collected a set of 114 CVEs related to WebAssembly from 2016 to 2024 using keywords, such as “WebAssembly”, “Wasm”, etc.
- Out of 114, 61 are associated with the WebAssembly runtimes and the remaining are for non-runtime tools such as Binaryen and wabt, which are designed for optimizing, debugging, or compiling of WebAssembly.
- In addition to CVEs, we collected a set of 346 GitHub issues from the runtimes repositories using keyworks “Vulnerability”, “Security”, “Overflow”, etc.
- Out of 346, 123 refers to the reporting of an actual vulnerability.

Execution Modes

- Just-in-time (JIT) compilation is an execution technique that dynamically translates the bytecode or intermediate representation of a program into native machine code during runtime, just before it is getting executed

- Ahead-of-time (AOT) compilation involves precompiling WebAssembly code into machine code before execution. The compiled native code is typically stored as a disk file and loaded into memory when the Wasm code needs to be executed

- A WebAssembly interpreter is a software component specifically designed to directly execute WebAssembly bytecode without first translating it into machine code. It operates by reading the binary format of a WebAssembly module and sequentially executing the instructions it contains

Analyzed Runtimes

- **Wasm3**: A lightweight and efficient WebAssembly interpreter that focuses on quick startup times. It is suitable for resource-constrained environments
- **Wasmtime**: A fast and secure runtime for WebAssembly that supports multiple programming languages and provides a standalone command-line interface. It is developed by Bytecode Alliance and supports both JIT (Just-In-Time) and AOT (Ahead-of-Time) compilation
- **WAVM:** WebAssembly runtime for non-web applications that uses LLVM compiler to compile WebAssembly code to machine code with close to native performance
- **WAMR:** WebAssembly Micro Runtime (WAMR) is a lightweight standalone runtime with small footprint, high performance and suitable for applications of embedded devices, IoT, and smart contract. Besides providing an interpreter, it also supports two binary execution modes, AoT and JIT

Wasmtime Vulnerabilities Overview

- There exist 15 CVEs for known vulnerabilities detected in Wasmtime
- Wasmtime vulnerabilities have detailed description and commits that patch the vulnerabilities are clearly identified
- Most vulnerabilities are no longer alive in the latest version of Wasmtime

Wasmtime Vulnerabilities Root Causes

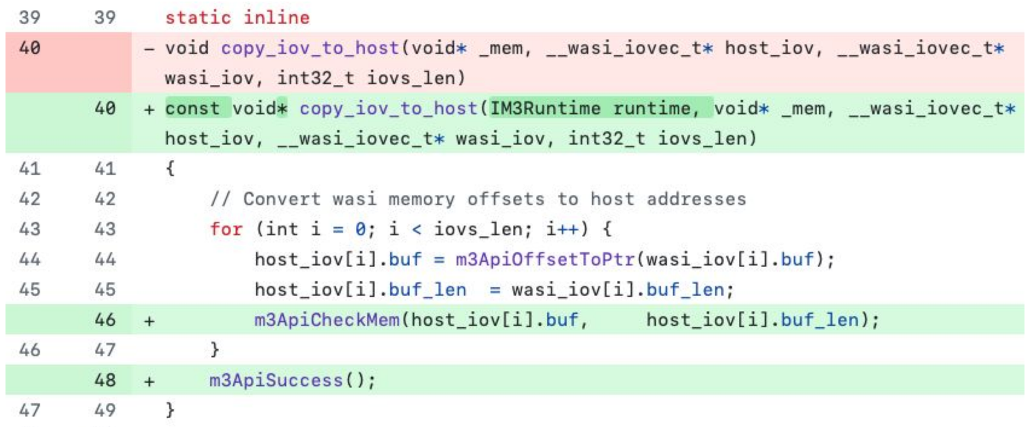
- Handling reference types (externref)
  - WebAssembly supports a value type called externref that aims to enable more efficient communication between the host (JS) and the wasm module
- Implementing SIMD proposal
  - SIMD stands for single instruction, multiple data. SIMD instructions are a special class of instructions that exploit data parallelism in applications by performing the same operation on multiple data elements at the same time
- Bugs in Wasmtime’s code generator (Cranelift)
  - Cranelift is a fast, secure, and relatively simple compiler backend. It takes an intermediate representation of a program generated by some frontend and compiles it to executable machine code. It is used by the Wasmtime for JIT and AOT compilation

Wasm3 Vulnerabilities Overview

- There exist 9 CVEs for known vulnerabilities detected in Wasm3. We also found 6 more vulnerabilities which are not assigned CVE IDs (unofficial ones)
- Unlike Wasmtime, Wasm3 vulnerabilities do not have detailed description and commits that patch the vulnerabilities are not clearly identified.
- Almost all vulnerabilities share the same root cause.
- Almost all vulnerabilities are still alive in the latest version of Wasm3

Wasm3 Vulnerabilities Root Causes

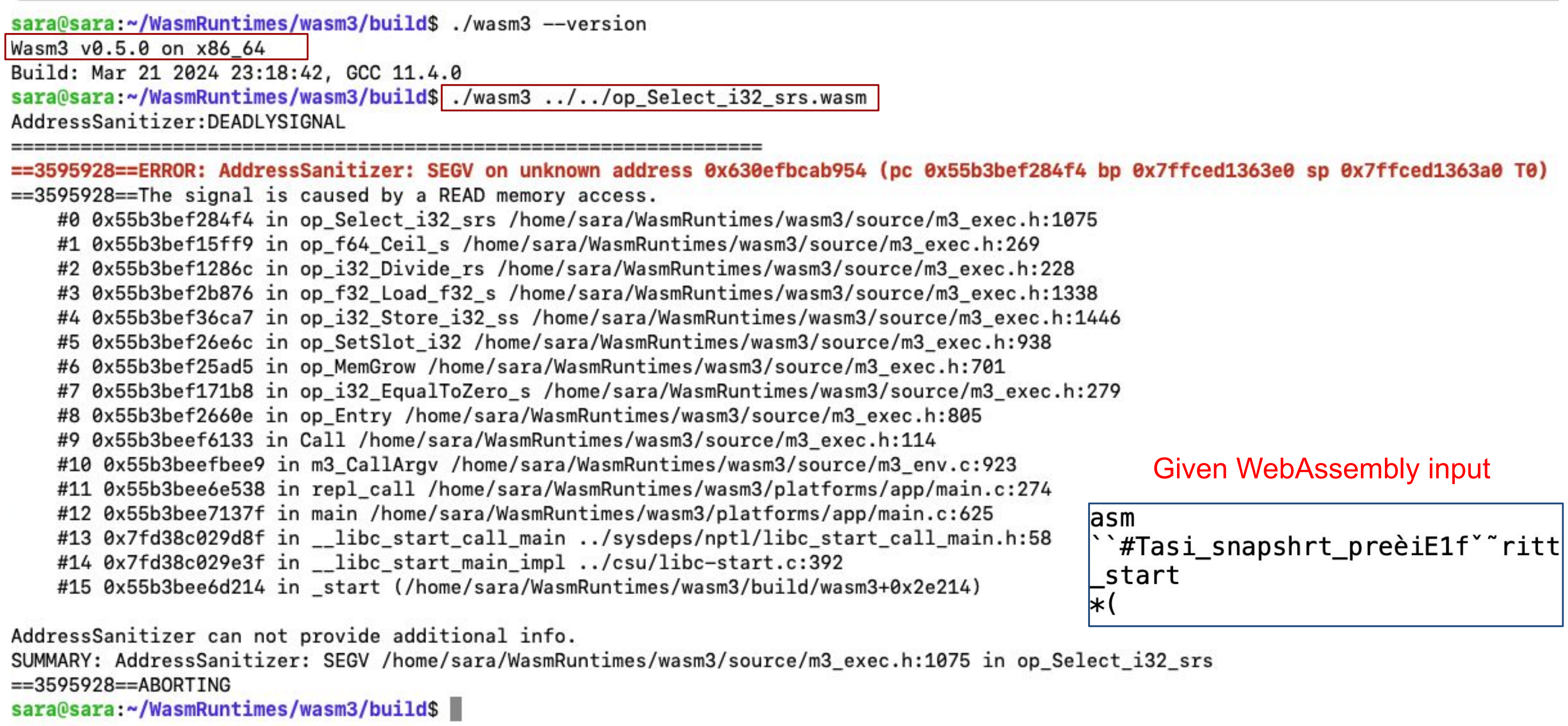
- Incomplete WebAssembly input validation
  - Currently, Wasm3 assumes that input wasm files are validated. It performs many checks, but the Wasm3 validation process is incomplete. Thus, Wasm3 gets many fuzzer errors, the majority of which are produced by invalid WASM inputs
- WASI API, which uses I/O vectors without checking the iovs buffer length and the buffer address



```
   static inline
 - void copy_iov_to_host(void* _mem, __wasi_iovec_t* host_iov, __wasi_iovec_t*
   wasi_iov, int32_t iovs_len)
 + const void* copy_iov_to_host(IM3Runtime runtime, void* _mem, __wasi_iovec_t*
   host_iov, __wasi_iovec_t* wasi_iov, int32_t iovs_len)
   {
       // Convert wasi memory offsets to host addresses
       for (int i = 0; i < iovs_len; i++) {
           host_iov[i].buf = m3ApiOffsetToPtr(wasi_iov[i].buf);
           host_iov[i].buf_len  = wasi_iov[i].buf_len;
 +         m3ApiCheckMem(host_iov[i].buf,      host_iov[i].buf_len);
       }
 +     m3ApiSuccess();
   }
```

Incomplete input validation in Wasm3

An example for invalid WebAssembly input that results in a crash in Wasm3.



```
sara@sara:~/WasmRuntimes/wasm3/build$ ./wasm3 --version
Wasm3 v0.5.0 on x86_64
Build: Mar 21 2024 23:18:42, GCC 11.4.0
sara@sara:~/WasmRuntimes/wasm3/build$ ./wasm3 ../../op_Select_i32_srs.wasm
AddressSanitizer:DEADLYSIGNAL
=================================================================
==3595928==ERROR: AddressSanitizer: SEGV on unknown address 0x630efbcab954 (pc 0x55b3bef284f4 bp 0x7ffced1363e0 sp 0x7ffced1363a0 T0)
==3595928==The signal is caused by a READ memory access.
    #0 0x55b3bef284f4 in op_Select_i32_srs /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:1075
    #1 0x55b3bef15ff9 in op_f64_Ceil_s /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:269
    #2 0x55b3bef1286c in op_i32_Divide_rs /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:228
    #3 0x55b3bef2b876 in op_f32_Load_f32_s /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:1338
    #4 0x55b3bef36ca7 in op_i32_Store_i32_ss /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:1446
    #5 0x55b3bef26e6c in op_SetSlot_i32 /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:938
    #6 0x55b3bef25ad5 in op_MemGrow /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:701
    #7 0x55b3bef171b8 in op_i32_EqualToZero_s /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:279
    #8 0x55b3bef2660e in op_Entry /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:805
    #9 0x55b3beef6133 in Call /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:114
    #10 0x55b3beefbee9 in m3_CallArgv /home/sara/WasmRuntimes/wasm3/source/m3_env.c:923
    #11 0x55b3bee6e538 in repl_call /home/sara/WasmRuntimes/wasm3/platforms/app/main.c:274
    #12 0x55b3bee7137f in main /home/sara/WasmRuntimes/wasm3/platforms/app/main.c:625
    #13 0x7fd38c029d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #14 0x7fd38c029e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #15 0x55b3bee6d214 in _start (/home/sara/WasmRuntimes/wasm3/build/wasm3+0x2e214)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV /home/sara/WasmRuntimes/wasm3/source/m3_exec.h:1075 in op_Select_i32_srs
==3595928==ABORTING
sara@sara:~/WasmRuntimes/wasm3/build$
```

Given WebAssembly input

```
asm
``#Tasi_snapshrt_preèiE1fˇ˜ritt
_start
*(
```

USC Viterbi School of Engineering

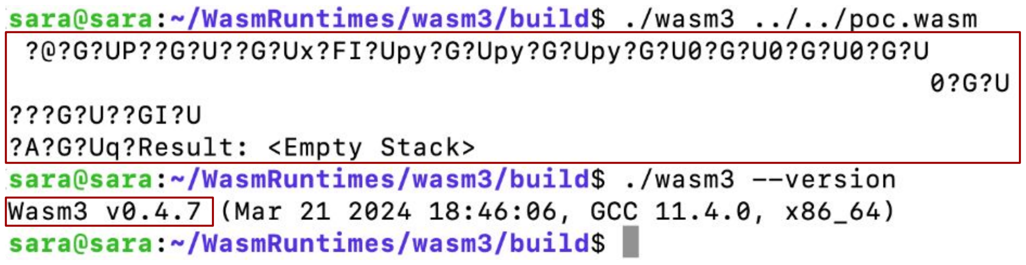
University of Southern California

CVE Reproduction

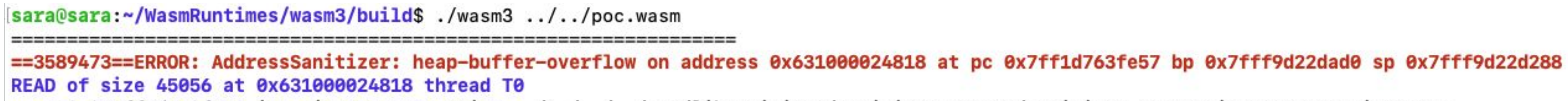
Since CVE-2022-28990 is fixed in the latest version of Wasm3, when we tried to reproduce it using v0.5.0, instead of crash the error is trapped.

```
sara@sara:~/WasmRuntimes/wasm3/build$ ./wasm3 --version
Wasm3 v0.5.0 on x86_64
Build: Mar 21 2024 18:19:53, GCC 11.4.0
sara@sara:~/WasmRuntimes/wasm3/build$ ./wasm3 ../../poc.wasm
Error: [trap] out of bounds memory access
```

We could reproduce the vulnerability by using versions before v0.4.9, where buffer overflow results in memory leakage.



```
sara@sara:~/WasmRuntimes/wasm3/build$ ./wasm3 ../../poc.wasm
 ?@?G?UP??G?U??G?Ux?FI?Upy?G?Upy?G?Upy?G?U0?G?U0?G?U0?G?U
                                                        0?G?U

???G?U??GI?U
?A?G?Uq?Result: <Empty Stack>
sara@sara:~/WasmRuntimes/wasm3/build$ ./wasm3 --version
Wasm3 v0.4.7 (Mar 21 2024 18:46:06, GCC 11.4.0, x86_64)
sara@sara:~/WasmRuntimes/wasm3/build$ 
```



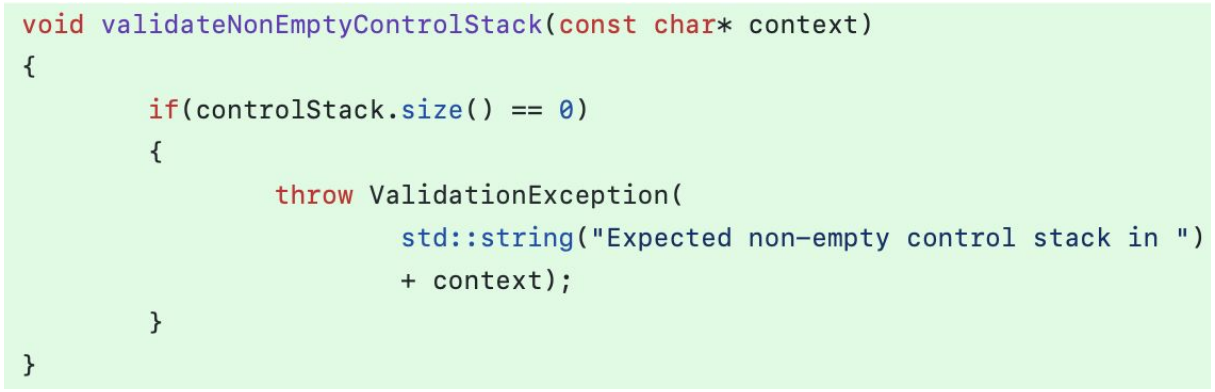
```
sara@sara:~/WasmRuntimes/wasm3/build$ ./wasm3 ../../poc.wasm
=================================================================
==3589473==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x631000024818 at pc 0x7ff1d763fe57 bp 0x7fff9d22dad0 sp 0x7fff9d22d288
READ of size 45056 at 0x631000024818 thread T0
```

WAVM Vulnerabilities Overview

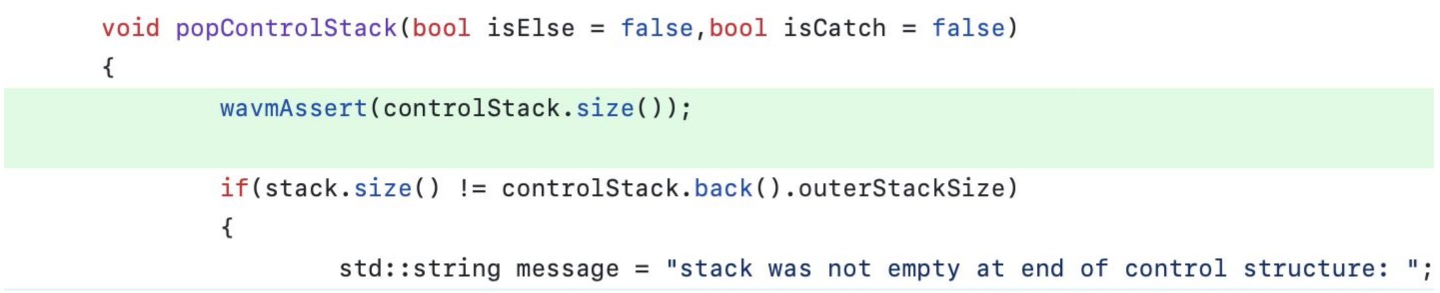
- There exist 9 CVEs for known vulnerabilities detected in WAVM. We also found 2 more vulnerabilities which are not assigned CVE IDs (unofficial ones)
- WAVM vulnerabilities do not have detailed description but the commits that patch the vulnerabilities are clearly identified
- Almost all vulnerabilities share the same root cause, again incomplete input validation
- Currently, WAVM validates input modules by doing many checks and it continuously check the control stack during execution to make sure the control stack is non-empty before reaching the end of execution
- Almost all vulnerabilities have been patched in the latest version of WAVM

WAVM Vulnerabilities Root Causes

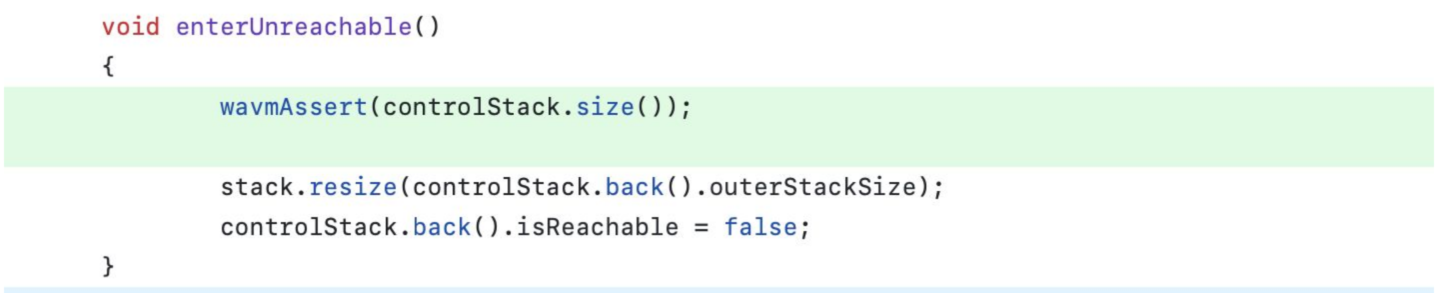
- **Empty control stack:** In WebAssembly, control flow instructions (e.g., if, end, loop, block) are used to structure the control flow of programs. Each time a control frame is encountered, it is pushed onto the control stack, and when the control structure is completed, the corresponding frame is popped. If a given WebAssembly input violates the WebAssembly specification, it may contain incorrect or mismatched control flow instructions. These violations might cause the control stack to become empty at a certain point before reaching the end of execution.



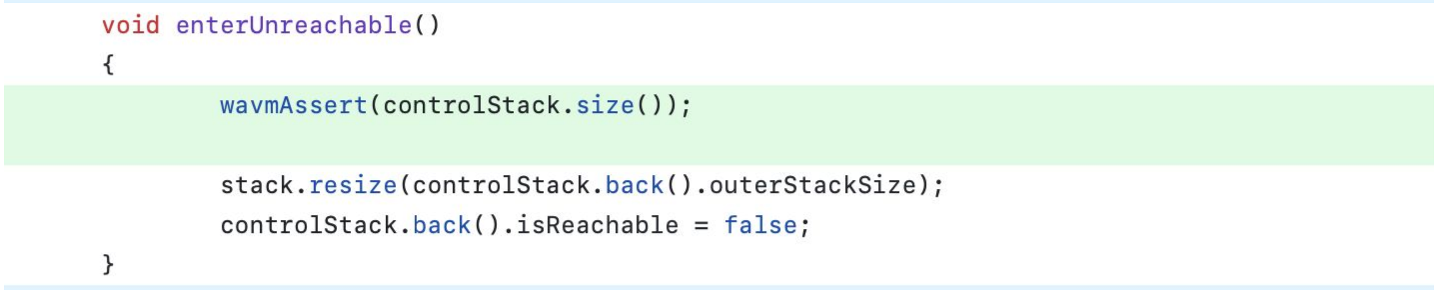
```
void validateNonEmptyControlStack(const char* context)
{
        if(controlStack.size() == 0)
        {
                throw ValidationException(
                        std::string("Expected non-empty control stack in ")
                        + context);
        }
}
```



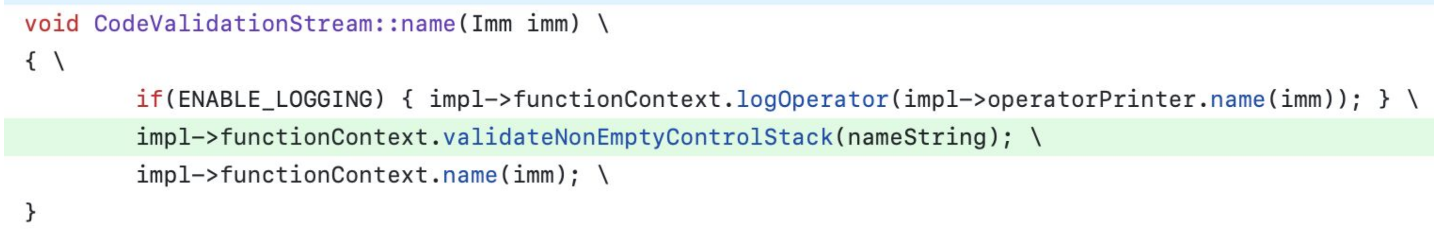
```
void popControlStack(bool isElse = false,bool isCatch = false)
{
        wavmAssert(controlStack.size());

        if(stack.size() != controlStack.back().outerStackSize)
        {
                std::string message = "stack was not empty at end of control structure: ";
```



```
void enterUnreachable()
{
        wavmAssert(controlStack.size());

        stack.resize(controlStack.back().outerStackSize);
        controlStack.back().isReachable = false;
}
```



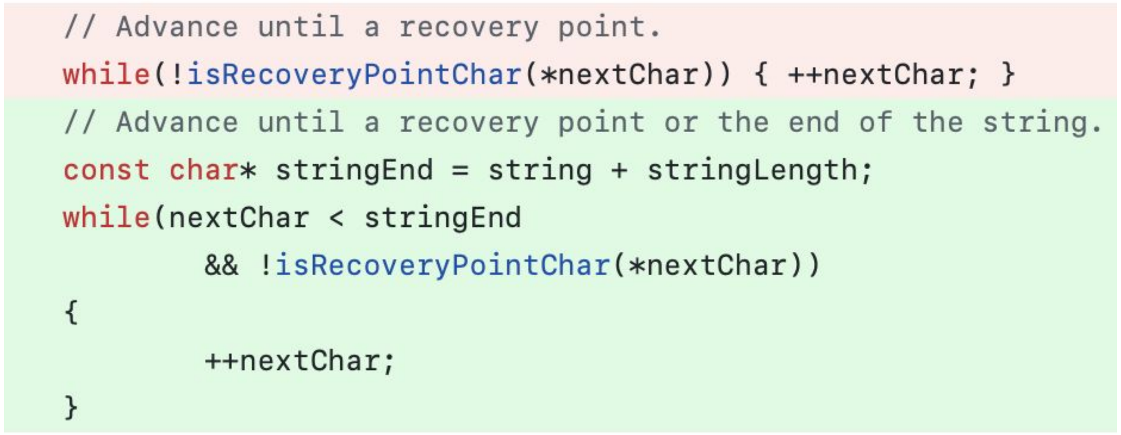
```
void enterUnreachable()
{
        wavmAssert(controlStack.size());

        stack.resize(controlStack.back().outerStackSize);
        controlStack.back().isReachable = false;
}
```



```
void CodeValidationStream::name(Imm imm) \
{ \
        if(ENABLE_LOGGING) { impl->functionContext.logOperator(impl->operatorPrinter.name(imm)); } \
        impl->functionContext.validateNonEmptyControlStack(nameString); \
        impl->functionContext.name(imm); \
}
```

WAVM Vulnerabilities Root Causes

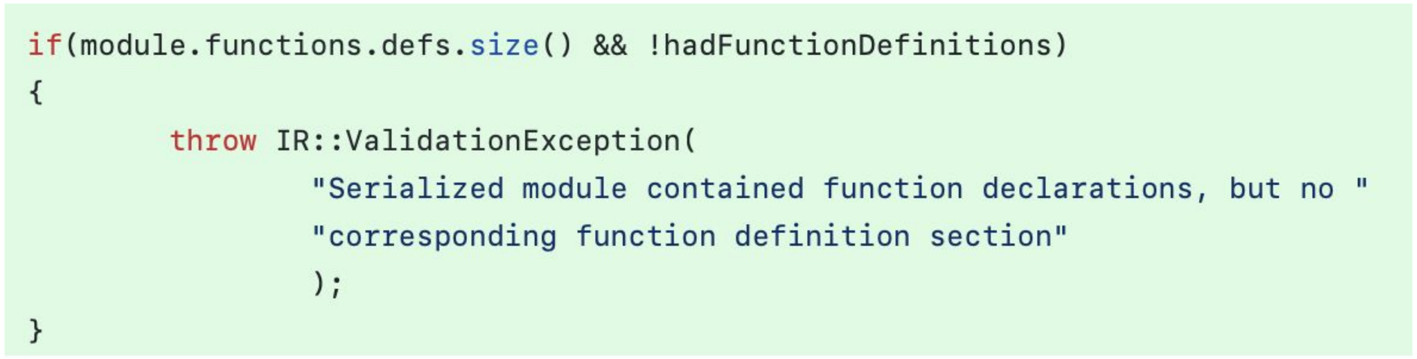
- **Error recovery in lexer:** Error recovery in a lexer refers to the technique employed to handle errors that arise during the scanning or lexing phase of the input source code. WAVM lexer supports error recovery. If it encounters an invalid token, it will advance until it reaches a recovery point. However, there is a bug in implementing the lexer error recovery where the lexer continues to parse without considering the end of the input. Thus, it cause out-of-bounds memory access when loading an invalid input.



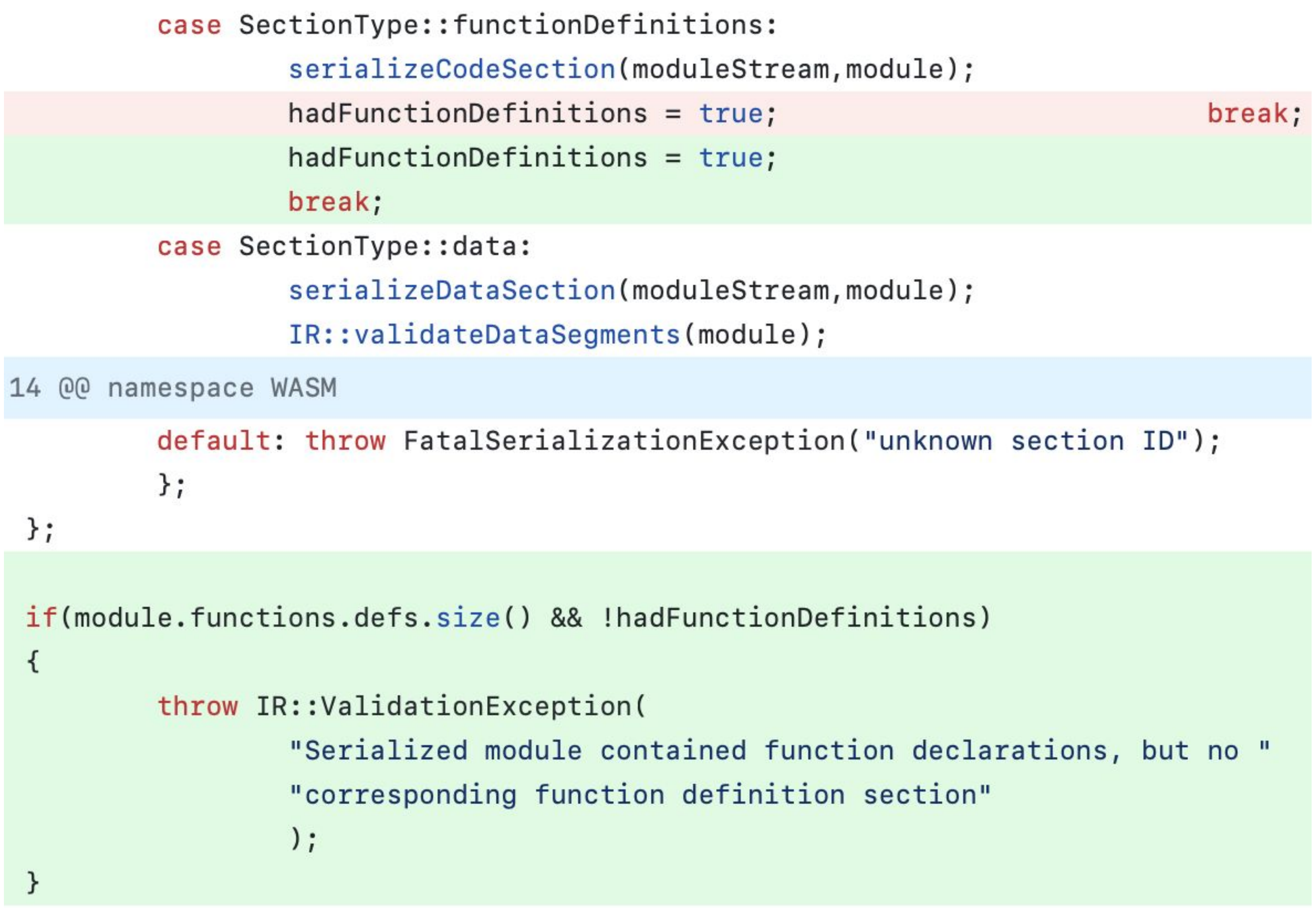
```
// Advance until a recovery point.
while(!isRecoveryPointChar(*nextChar)) { ++nextChar; }
// Advance until a recovery point or the end of the string.
const char* stringEnd = string + stringLength;
while(nextChar < stringEnd
        && !isRecoveryPointChar(*nextChar))
{
        ++nextChar;
}
```

WAVM Vulnerabilities Root Causes

- **Missing of function definition:** Function declarations provide a way to declare the existence and signature of a function without providing its actual implementation. But, function definitions include the implementation code for the declared functions. If a module includes function declarations without a corresponding function definition section, it should be considered an invalid input. In WAVM, the loader did not reject an input containing such a function, which subsequently led to undefined behavior and security issues when calling the function.



```
if(module.functions.defs.size() && !hadFunctionDefinitions)
{
        throw IR::ValidationException(
                "Serialized module contained function declarations, but no "
                "corresponding function definition section"
                );
}
```



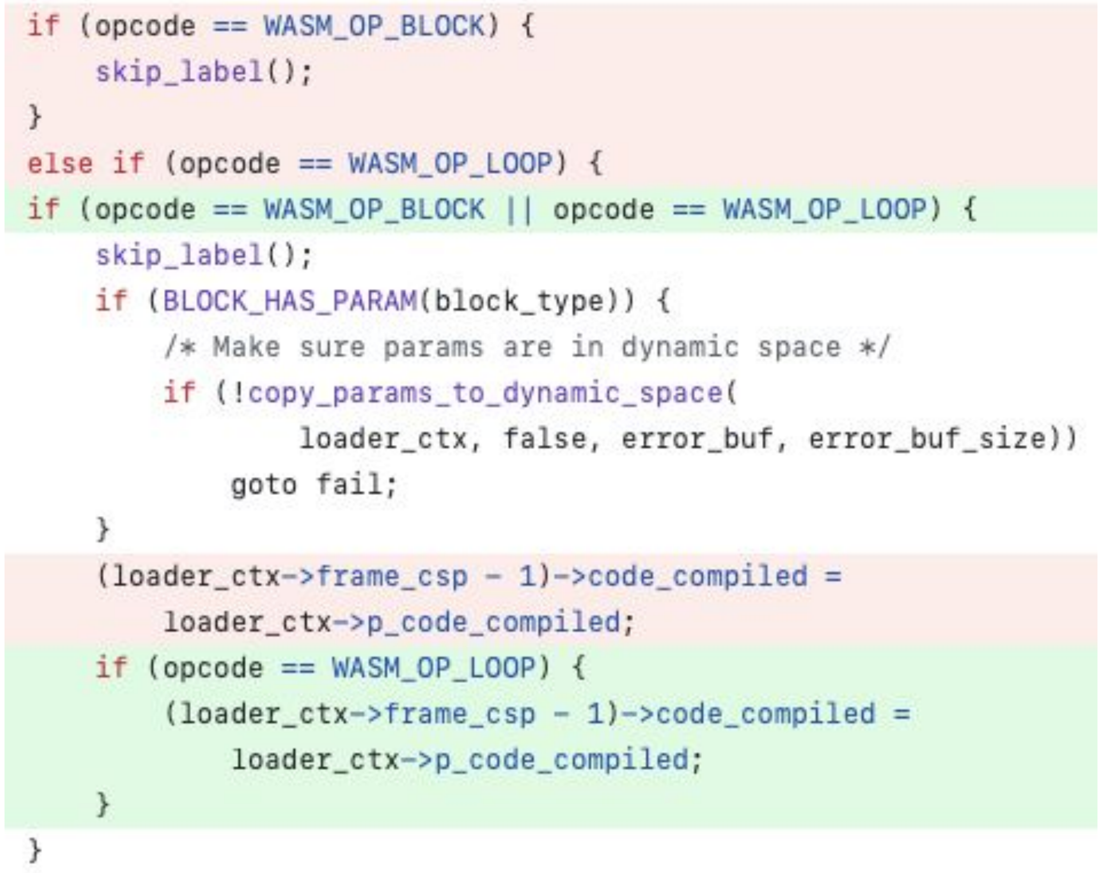
```
        case SectionType::functionDefinitions:
                serializeCodeSection(moduleStream,module);
                hadFunctionDefinitions = true;                         break;
                hadFunctionDefinitions = true;
                break;
        case SectionType::data:
                serializeDataSection(moduleStream,module);
                IR::validateDataSegments(module);
14 @@ namespace WASM
        default: throw FatalSerializationException("unknown section ID");
        };
};

if(module.functions.defs.size() && !hadFunctionDefinitions)
{
        throw IR::ValidationException(
                "Serialized module contained function declarations, but no "
                "corresponding function definition section"
                );
}
```

WAMR Vulnerabilities Overview

- There exist 2 CVEs for known vulnerabilities detected in WAMR. We also found 40 more vulnerabilities which are not assigned CVE IDs (unofficial ones)
- WAMR vulnerabilities do not have detailed description but the commits that patch the vulnerabilities are partially identified
- Almost all vulnerabilities share the same root cause, again incomplete input validation
- Currently, WAMR validates input modules by doing many checks
- Almost all vulnerabilities have been patched in the latest version of WAMR

WAMR Vulnerabilities Root Causes

- Implementation bug in loader and control stack
  - wasm_loader_push_pop_frame_offset() may pop *n* operands and it uses stack_cell_num field of loader_ctx to check whether the operand can be popped or not. While stack_cell_num is updated in a later function, the check may fail if the stack is in polymorphic state and lead to integer underflow.
  - Missing of buffer length checking
- Incomplete WebAssembly input validation



```
if (opcode == WASM_OP_BLOCK) {
    skip_label();
}
else if (opcode == WASM_OP_LOOP) {
if (opcode == WASM_OP_BLOCK || opcode == WASM_OP_LOOP) {
    skip_label();
    if (BLOCK_HAS_PARAM(block_type)) {
        /* Make sure params are in dynamic space */
        if (!copy_params_to_dynamic_space(
                loader_ctx, false, error_buf, error_buf_size))
            goto fail;
    }
    (loader_ctx->frame_csp - 1)->code_compiled =
        loader_ctx->p_code_compiled;
    if (opcode == WASM_OP_LOOP) {
        (loader_ctx->frame_csp - 1)->code_compiled =
            loader_ctx->p_code_compiled;
    }
}
```

Firefox Vulnerabilities Overview

- 10 CVEs, all fixed
- 1 CVE still not visible due to disclosure policy
- Over 9 CVEs with detailed information
  - 8 have P1 priority to fix, 1 with P2
    - P1: We definitely want this. It's a major feature
    - P2: We want this, but it's not totally clear or extremely important
    - P3: not a short-term goal; P4: not a long-term goal
  - Severity: 2 with P2, 4 with P3, 1 with P4
    - 2 older CVEs used older system and they are all "normal"
- Reproduction: 7 done, 2 not yet
  - ARM64 platform only: I only have phones with ARM64 and debugging is harder

Firefox Vulnerabilities Root Causes

- Multithread: thread isolation issue
  - CVE-2020-15681, CVE-2021-23970
- Garbage Collection: use-after-free
  - CVE-2021-43539, CVE-2018-5094
- ARM64 platform-specific issues: Register allocation, delayed icache flush
  - CVE-2022-31740, CVE-2022-40957
- Other issues
  - Arithmetic underflow: CVE-2018-5093
  - Wrong data size calculation: CVE-2021-29945
  - Reading stale global variables: CVE-2023-4046

Summary

- We analyzed 83 vulnerabilities in 4 standalone WebAssembly runtimes and 10 bugs in Firefox
- We observed that incomplete WebAssembly input validation is the most common vulnerability root cause
- The WebAssembly input validation process involves different parts such as appropriate handling of the control stack, checking function definitions, checking types, robust error recovery and etc.
- We observed differences between the responsibility of developers for fixing vulnerabilities

Thank You!