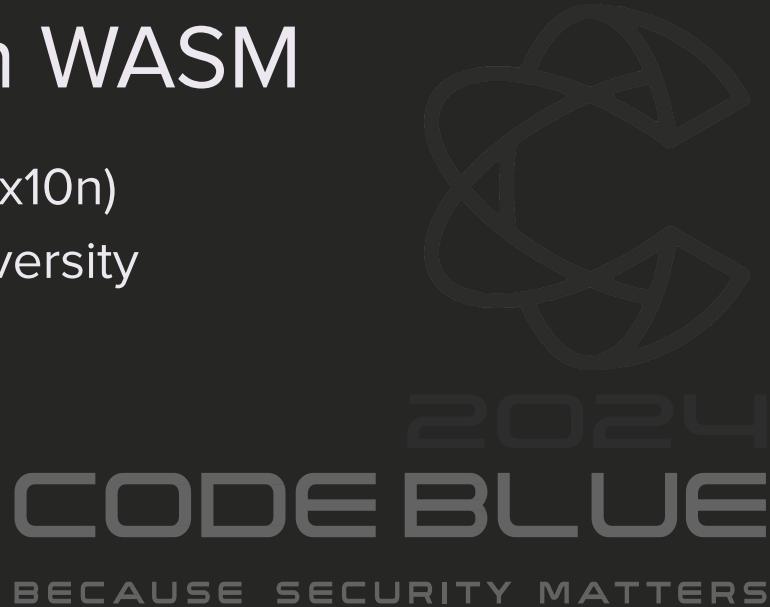


WebAssembly Is All You Need: Exploiting Chrome and the V8 Sandbox 10+ times with WASM

Seunghyun Lee (@0x10n)
Carnegie Mellon University



\$ whoami



- First-year PhD student @ **CMU CSD / CyLab**
- (Former) Research intern @ **KAIST Hacking Lab**
- Occasional CTF player as **PPP**, KAIST GoN

Seunghyun Lee,
a.k.a Xion (@0x10n)
Carnegie Mellon University

\$ whoami



Seunghyun Lee,
a.k.a Xion (@0x10n)
Carnegie Mellon University

- First-year PhD student @ **CMU CSD / CyLab**
- (Former) Research intern @ **KAIST Hacking Lab**
- Occasional CTF player as **PPP**, KAIST GoN
- Independent vulnerability researcher as a hobby
 - Winner of **Pwn2Own Vancouver 2024**:
 - Chrome renderer + Chrome/Edge renderer double-tap
 - Winner of **TyphoonPWN 2024**:
 - Chrome renderer
 - Google kernelCTF & **v8CTF** enjoyer:
 - Q: How many 0-days in a single Chrome milestone?



tsuro Today at 4:32 AM

We got 4 exploits for M129, let's see if we can break the record for M130 😊



CODE BLUE 2024
BECAUSE SECURITY MATTERS

Why this talk?

- Finding and exploiting browser bugs are “hard”?
 - What is it that makes it “hard”?
 - How can we make it easier as an attacker?
 - How can we make it harder as a defender?

Why this talk?

- Finding and exploiting browser bugs are “hard”?
 - What is it that makes it “hard”?
 - How can we make it easier as an attacker?
 - How can we make it harder as a defender?
- Lack of publicly available information on vulnerability research
 - Not a lot of discussions on bleeding-edge vulnerabilities (and understandably so)
 - kernelCTF requires exploit to be published in detail, v8CTF does not? 🤯
 - Publicize knowledge & insights to collectively advance vulnerability research

Agenda

- The Prequel: CVE-2024-2887
 - WasmGC type system
- The Lore: Speedrunning TyphoonPWN with variant analysis
 - Isorecursive type system in WasmGC
- “Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024
 - The `wasm::ValueType` Trinity
- The Sequel: CVE-2024-9859
- Typos Gone Wild: CVE-2024-6779
- “All-You-Can-Eat” Wasm-based V8 Sandbox bypasses
- Going Forward: Other browsers & future targets
- Conclusions & Takeaways

Agenda

- The Prequel: CVE-2024-2887
 - WasmGC type system
- The Lore: Speedrunning TyphoonPWN with variant analysis
 - Isorecursive type system in WasmGC
- “Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024
 - The `wasm::ValueType` Trinity
- The Sequel: CVE-2024-9859
- Typos Gone Wild: CVE-2024-6779
- “All-You-Can-Eat” Wasm-based V8 Sandbox bypasses
- Going Forward: Other browsers & future targets
- Conclusions & Takeaways

Agenda

- The Prequel: CVE-2024-2887
 - WasmGC type system
- The Lore: Speedrunning TyphoonPWN with variant analysis
 - Isorecursive type system in WasmGC
- “Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024
 - The `wasm::ValueType` Trinity
- The Sequel: CVE-2024-9859
- Typos Gone Wild: CVE-2024-6779
- “All-You-Can-Eat” Wasm-based V8 Sandbox bypasses
- Going Forward: Other browsers & future targets
- Conclusions & Takeaways

The Prequel: CVE-2024-2887

The Prequel: CVE-2024-2887



ZERO DAY
INITIATIVE

PRIVACY

WHO WE ARE

HOW IT WORKS

BLOG

ADVISORIES

[LOG IN](#) | [SIGN UP](#)

SUBSCRIBE

CVE-2024-2887: A PWN2OWN WINNING BUG IN GOOGLE CHROME

May 02, 2024 | Guest Blogger

[**< BACK TO THE BLOG**](#)



In this guest blog from Master of Pwn winner Manfred Paul, he details CVE-2024-2887 – a type confusion bug that occurs in both Google Chrome and Microsoft Edge (Chromium). He used this bug as a part of his winning exploit that led to code execution in the renderer of both browsers. This bug was quickly patched by both Google and Microsoft. Manfred has graciously provided this detailed write-up of the vulnerability and how he exploited it at the contest.



CODE BLUE 2024
BECAUSE SECURITY MATTERS

10

The Prequel: CVE-2024-2887

- Presented by Manfred Paul (@_manfp) at Pwn2Own Vancouver 2024
- TL;DR: Universal Wasm type confusion due to missing type count check
 - So what is a “Wasm type”?

WasmGC type system

- WASM modules may contain type section, a list of module-defined heap types
 - Base Spec: func
 - WasmGC Extension: struct, array, ...
- Each module-defined heap types has its own type index
 - The order of their appearance in the type section is its type index
- WASM modules can have at most kV8MaxWasmTypes defined heap types

```
27 // The following limits are imposed by V8 on WebAssembly modules.  
28 // The limits are agreed upon with other engines for consistency.  
29 constexpr size_t kV8MaxWasmTypes = 1'000'000;
```

WasmGC type system

- WASM also supports recursive types within a “recursion group” `rectype`

Recursive Types

Recursive types denote a group of mutually recursive `composite types`, each of which can optionally declare a list of `type indices` of supertypes that it `matches`. Each type can also be declared `final`, preventing further subtyping.

```
rectype ::= rec subtype*
subtype ::= sub final? typeidx* comptype
```

In a `module`, each member of a recursive type is assigned a separate `type index`.

- `rectype` can contain multiple `subtype` members
 - **Each members are assigned a separate type index**, but not to `rectype` itself

WasmGC type system

- Type index example:

```
(module
  (rec
    (type $t1 (struct (field i32 (ref $t2))))
    (type $t2 (struct (field i64 (ref $t1))))
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))
    (type $u2 (struct (field i64 (ref $u1))))
  )
  (type $v (struct (field (ref $t1))))
)
```

0

1

2

3

4

The Prequel: CVE-2024-2887

```
void DecodeTypeSection() {
    TypeCanonicalizer* type_canon = GetTypeCanonicalizer();
    uint32_t types_count = consume_count("types count", kV8MaxWasmTypes); // (1)

    for (uint32_t i = 0; ok() && i < types_count; ++i) {
        ...
        uint8_t kind = read_u8<Decoder::FullValidationTag>(pc(), "type kind");
        size_t initial_size = module_->types.size();
```

```
if (kind == kWasmRecursiveTypeGroupCode) {
    ...
    uint32_t group_size =
        consume_count("recursive group size", kV8MaxWasmTypes);
    ...
    if (initial_size + group_size > kV8MaxWasmTypes) { // (2)
        errorf(pc(), "Type definition count exceeds maximum %zu",
               kV8MaxWasmTypes);
        return;
    }
    ...
    for (uint32_t j = 0; j < group_size; j++) {
```

```
} else {
    ...
    // Similarly to above, we need to resize types for a group of size 1.
    module_->types.resize(initial_size + 1); // (3)
    module_->isrecursive_canonical_type_ids.resize(initial_size + 1);
    TypeDefinition type = consume_subtype_definition();
    if (ok()) {
        module_->types[initial_size] = type;
        type_canon->AddRecursiveSingletonGroup(module_->get());
    }
}
```



The Prequel: CVE-2024-2887

- (L) For recursive type groups, type count limit is checked
- (R) For “standalone” types, limit is not checked???
 - types_count bounded above by kV8MaxWasmTypes, but this includes rectypes

```
if (kind == kWasmRecursiveTypeGroupCode) {  
    ...  
    uint32_t group_size =  
        consume_count("recursive group size", kV8MaxWasmTypes);  
    ...  
    if (initial_size + group_size > kV8MaxWasmTypes) { // (2)  
        errorf(pc(), "Type definition count exceeds maximum %zu",  
               kV8MaxWasmTypes);  
        return;  
    }  
    ...  
    for (uint32_t j = 0; j < group_size; j++) {  
        ...  
        TypeDefinition type = consume_subtype_definition();  
        module_->types[initial_size + j] = type;  
    }  
    ...
```

```
} else {  
    ...  
    // Similarly to above, we need to resize types for a group of size 1.  
    module_->types.resize(initial_size + 1); // (3)  
    module_->isrecursive_canonical_type_ids.resize(initial_size + 1);  
    TypeDefinition type = consume_subtype_definition();  
    if (ok()) {  
        module_->types[initial_size] = type;  
        type_canon->AddRecursiveSingletonGroup(module_->get());  
    }  
}
```



The Prequel: CVE-2024-2887

- Exceed max type count with a **standalone type** - no check!

```
<script src="wasm-module-builder.js"></script>
<script>
    const builder = new WasmModuleBuilder();

    // 1. recursive group with 1000000 types
    builder.startRecGroup();
    for (let i = 0; i < 1000000; i++) {
        builder.addArray(kWasmI32);
    }
    builder.endRecGroup();

    // 2. standalone type
    //builder.startRecGroup();
    builder.addStruct([makeField(wasmRefType(kWasmI32), true)]);
    //builder.endRecGroup();

    console.log(builder.instantiate());
</script>
```

0 ~ 999999

1000000

▶ Instance {exports: {...}}

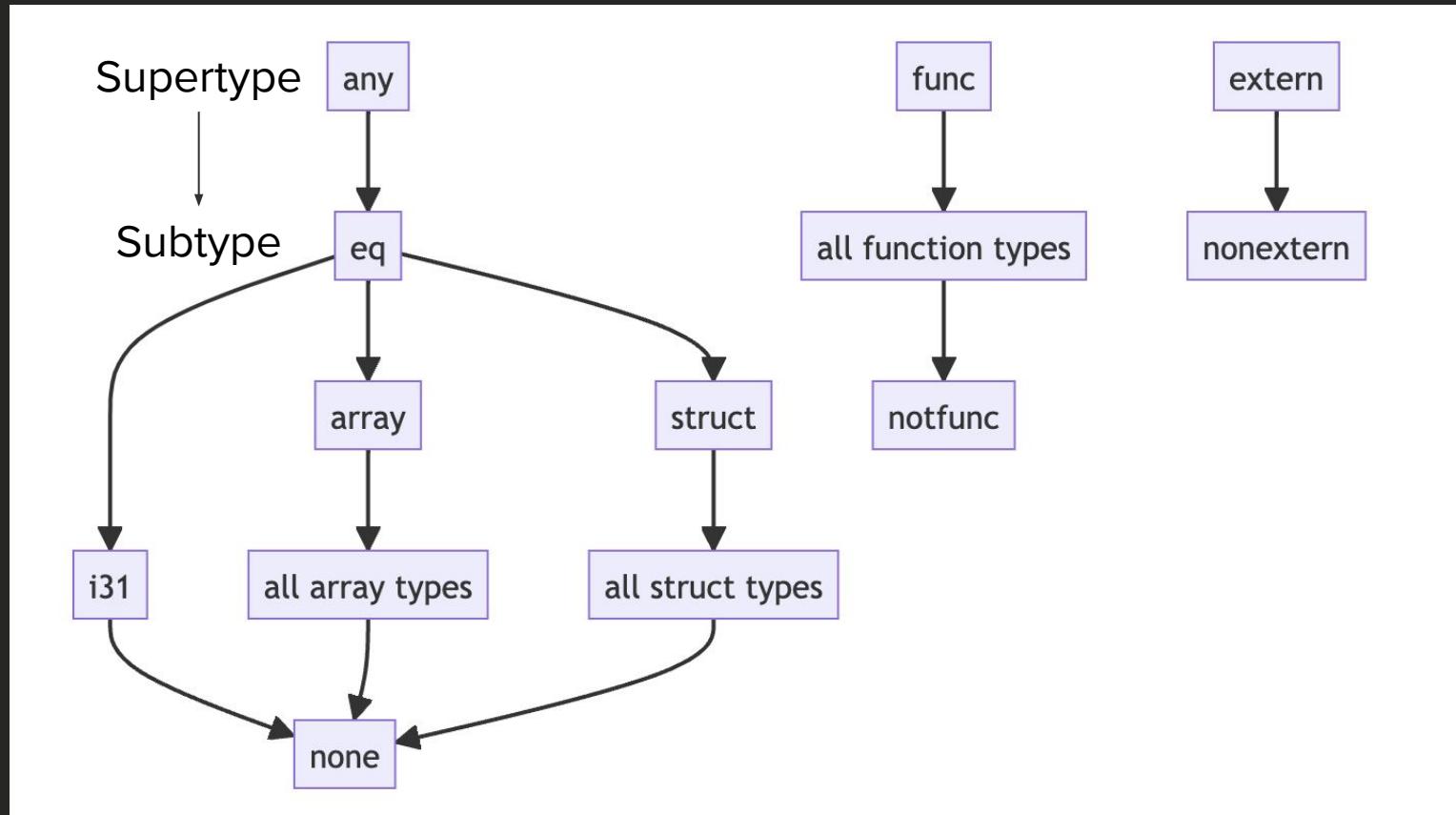
cve-2024-2887.html:17

The Prequel: CVE-2024-2887

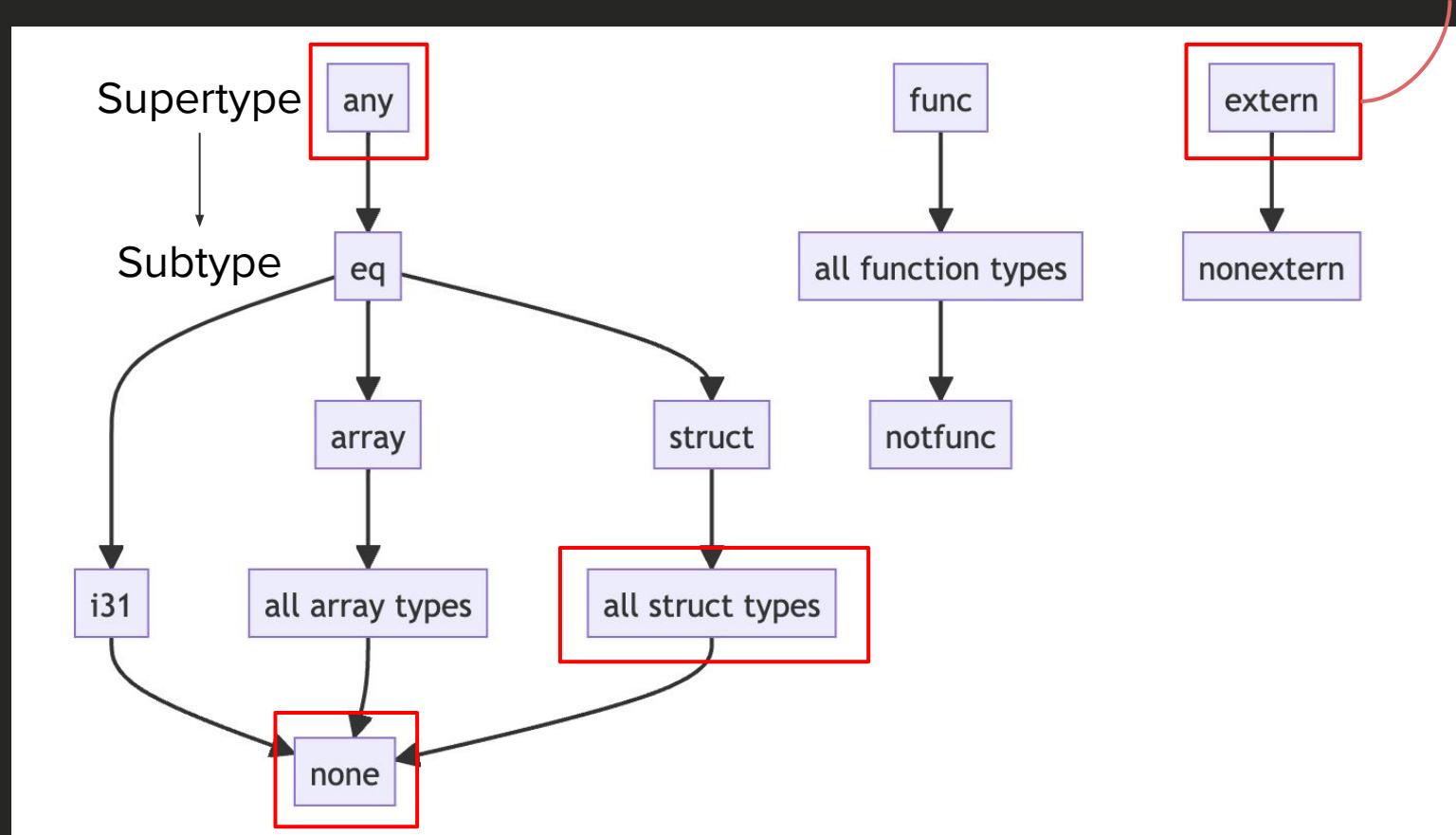
- How is this exploitable? It's just a resource exhaustion “bug”?
 - Generic heap types to the rescue!

```
// Represents a WebAssembly heap type, as per the typed-funcref and gc
// proposals.
// The underlying Representation enumeration encodes heap types as follows:
// a number t < kV8MaxWasmTypes represents the type defined in the module at
// index t. Numbers directly beyond that represent the generic heap types. The
// next number represents the bottom heap type (internal use).
class HeapType {
public:
    enum Representation : uint32_t {
        kFunc = kV8MaxWasmTypes, // shorthand: c
        kEq,                      // shorthand: q
        kI31,                     // shorthand: j
        kStruct,                  // shorthand: o
        kArray,                   // shorthand: g
        kAny,                     //
```

The Prequel: CVE-2024-2887



The Prequel: CVE-2024-2887



The Prequel: CVE-2024-2887

- Key idea for the exploit:
 - Any concrete struct type is a supertype of **none** by definition
 - **none**: Bottom type (i.e. subtype) of all internal non-function Wasm types
 - An object can be casted to its supertype object
 - Upcast, statically type-checked
 - What happens if, with this bug, a **concrete heap type index aliases with kNone?**
 - Object can be casted to any other type???

The Prequel: CVE-2024-2887

1. Create the following two types:

```
(type $tSrc (struct (field src))) // index = HeapType::kNone  
(type $tDst (struct (field dst)))
```

2. Push value of type src

=> Stack: src

3. Create struct \$tSrc

=> Stack: ref \$tSrc = ref none

4. Type cast to ref \$tDst

=> Stack: ref \$tDst

5. Get field of type dst from ref \$tDst

=> Stack: dst



The Prequel: CVE-2024-2887

1. Create the following two types:

```
(type $tSrc (struct (field src))) // index = HeapType::kNone  
(type $tDst (struct (field dst)))
```

2. Push value of type src

=> Stack: **src**

3. Create struct \$tSrc

=> Stack: ref \$tSrc = ref none

4. Type cast to ref \$tDst

=> Stack: ref \$tDst

5. Get field of type dst from ref \$tDst

=> Stack: **dst**



The Prequel: CVE-2024-2887

- Result: Type confusion from **src** to **dst**
 - “Universal” Wasm type confusion between arbitrary types!
- Immediately acquire all JS exploit primitives:
 - `ref extern` -> `i32`
 - `addrOf()`
 - `i32` -> `ref extern`
 - `fakeObj()`
 - `i32` -> `ref (struct (field i32))`
 - Arbitrary (caged) read/write

The Lore: Speedrunning TyphoonPWN with variant analysis



CODE BLUE 2024
BECAUSE SECURITY MATTERS

The Lore: Speedrunning TyphoonPWN with variant analysis

- **May 27:** Boredom exceeded the procrastination threshold
- **May 30:** TyphoonPWN 2024*
- 3-day Chrome renderer exploit speedrun



(Not a) real footage of me going through source.chromium.org



CODE BLUE 2024
BECAUSE SECURITY MATTERS

26

The Lore: Speedrunning TyphoonPWN with variant analysis

- Opened Chromium Code Search, but where should I look at?
- Recall: I have very limited time
 - I need an approach to find and exploit browser bugs in an “easy” way

May 27, 15:00

 source.chromium.org



May 30, 09:00

TyphoonPWN 2024



The Lore: Speedrunning TyphoonPWN with variant analysis

- How to find and exploit bugs “*easily*”, in the fastest way possible?
 - Target bug classes that grant stable, powerful primitives
 - Target large, complex but legible code
 - Target code that can be easily tested & have my understanding of the code verified
 - Target under-examined code

The Lore: Speedrunning TyphoonPWN with variant analysis

- My answer: WasmGC type system implementation
 - Bugs have shown extremely strong exploitability (CVE-2024-2887)
 - The implementation is huge and complex but manageable
 - `wasm-module-builder.js` to the rescue!
 - Seemingly no public research on Chrome's WasmGC type system implementation

The Lore: Speedrunning TyphoonPWN with variant analysis

- Where are we now?
 - Start recapping CVE-2024-2887

May 27, 15:00
 source.chromium.org



May 27, 17:00
 v8/src/wasm/*



May 30, 09:00
TyphoonPWN 2024



The Lore: Speedrunning TyphoonPWN with variant analysis

- Standing on the shoulders of giants: Recap on CVE-2024-2887

```
    } else {  
        ...  
        // Similarly to above, we need to resize types for a group of size 1.  
        module_->types.resize(initial_size + 1); // (3)  
        module_->isorecursive_canonical_type_ids.resize(initial_size + 1);  
        TypeDefinition type = consume_subtype_definition();  
        if (ok()) {  
            module_->types[initial_size] = type;  
            type_canon->AddRecursiveSingletonGroup(module_.get());  
        }  
    }  
}
```

- What is **isorecursive_canonical_type_ids**?

The Lore: Speedrunning TyphoonPWN with variant analysis

- `isorecursive_canonical_type_ids`:
 - `isorecursive`: **Isorecursive type system**
 - `canonical_type_ids`: **Canonicalized** representation of the types

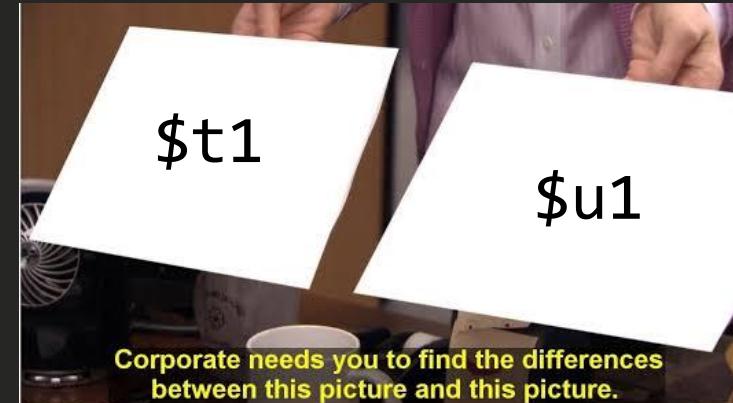
Isorecursive Type Systems

- Disclaimer:
 - I will try my best to be succinct as possible
 - See A. Rossberg, “Mutually Iso-Recursive Subtyping,” in OOPSLA’23 for details

Isorecursive Type Systems: Type Equivalence

- Is type \$t1 equivalent to type \$u1?

```
(module
  (rec
    (type $t1 (struct (field i32 (ref $t2))))
    (type $t2 (struct (field i64 (ref $t1))))
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))
    (type $u2 (struct (field i64 (ref $u1))))
  )
  (type $v (struct (field (ref $t1))))
)
```



Isorecursive Type Systems: Type Equivalence

- Is type \$t1 equivalent to type \$u1?

```
(module
  (rec
    (type $t1 (struct (field i32 (ref $t2))))
    (type $t2 (struct (field i64 (ref $t1))))
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))
    (type $u2 (struct (field i64 (ref $u1))))
  )
  (type $v (struct (field (ref $t1))))
)
```



- Yes, they look the same!
- But exactly how...?

Isorecursive Type Systems: Type Equivalence

Consequently, if there was an equivalent pair of types,

```
(rec
  (type $u1 (struct (field i32 (ref $u2))))
  (type $u2 (struct (field i64 (ref $u1))))
)
```

recorded in the context as

```
$u1 = (rec (struct (field i32 (ref $u2))) (struct (field i64 (ref $u1)))).0
$u2 = (rec (struct (field i32 (ref $u2))) (struct (field i64 (ref $u1)))).1
```

then to check the equivalence `$t1 == $u1`, both types are tied into iso-recursive types first:

```
tie($t1) = (rec (struct (field i32 (ref rec.1))) (struct (field i64 (ref rec.0)))).0
tie($u1) = (rec (struct (field i32 (ref rec.1))) (struct (field i64 (ref rec.0)))).0
```

In this case, it is immediately apparent that these are equivalent types.

Isorecursive Type Systems: Type Equivalence

- In plain language:
 - Represent recursive type group as type tuple `rec = (rec.0, rec.1, ...)`
 - i -th subtype = `rec.<i>`
 - Replace all recursive subtype references into `rec.<i>`
 - Compare this replaced type to check type equivalence

Isorecursive Type Systems: Type Equivalence

- Example: “Infinite list” of i32

```
(rec
  (type $t1 (struct (field i32 (ref $t2))))
  (type $t2 (struct (field i32 (ref $t1)))))
)
(rec
  (type $u1 (struct (field i32 (ref $u1))))
)
```

0
1

2

- None of the `tie()`’d representation equal, thus all different types

```
tie($t1) = (rec (struct (field i32 (ref rec.1))) (struct (field i64 (ref rec.0)))).0
tie($t2) = (rec (struct (field i32 (ref rec.1))) (struct (field i64 (ref rec.0)))).1
tie($u1) = (rec (struct (field i32 (ref rec.0)))).0
```



Isorecursive Type Systems: Canonicalization

- Q: How to represent types $\$u\{1, 2\}$ to be the same as $\$t\{1, 2\}$?

```
(module
  (rec
    (type $t1 (struct (field i32 (ref $t2))))
    (type $t2 (struct (field i64 (ref $t1))))
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))
    (type $u2 (struct (field i64 (ref $u1))))
  )
  (type $v (struct (field (ref $t1))))
)
```

0/0

1/1

2/0

3/1

4/2

- A: Canonicalize the type indices into (opaque) canonical type indices!
 - Type Index / Canonical Index
- `isorecursive_canonical_type_ids[module_type_idx] = canonical_type_idx`

Isorecursive Type Systems: (Declared) Subtyping

- Q: How do we know that the declared subtypes are valid?

```
(type $tSup          (struct (field (ref null any)      )) )  
(type $tSub (sub $tSup (struct (field (ref null none) i32))))
```

- A: Well-known - “Amber rule”^[1,2]

- TL;DR: `mutable ? (sub.i == sup.i) : (sub.i <: sup.i)`

[1] L. Cardelli, "Amber," in LITP'85.

[2] Y. Zhou, J. Zhao, B.C.D.S. Oliveira, "Revisiting Iso-Recursive Subtyping," in TOPLAS'22.

Isorecursive Type Systems: (Declared) Subtyping

- Subtype relationship saved as `canonical_supertypes_[sub] = super`
- So what is all this stuff for?

Isorecursive Type Systems: (Declared) Subtyping

- Cross-module subtype check:
 - Canonicalize, then follow `sub = canonical_supertypes_[sub]` until match / end

```
bool TypeCanonicalizer::IsCanonicalSubtype(uint32_t canonical_sub_index,
                                            uint32_t canonical_super_index) {
    // Multiple threads could try to register and access recursive groups
    // concurrently.
    // TODO(manoskouk): Investigate if we can improve this synchronization.
    base::MutexGuard mutex_guard(&mutex_);
    while (canonical_sub_index != kNoSuperType) {
        if (canonical_sub_index == canonical_super_index) return true;
        canonical_sub_index = canonical_supertypes_[canonical_sub_index];
    }
    return false;
}

bool TypeCanonicalizer::IsCanonicalSubtype(uint32_t sub_index,
                                            uint32_t super_index,
                                            const WasmModule* sub_module,
                                            const WasmModule* super_module) {
    uint32_t canonical_super =
        super_module->isorecursive_canonical_type_ids[super_index];
    uint32_t canonical_sub =
        sub_module->isorecursive_canonical_type_ids[sub_index];
    return IsCanonicalSubtype(canonical_sub, canonical_super);
}
```

“Deja Vu”: CVE-2024-6100

@ TyphoonPWN 2024

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- Enough with the background - let's find the bug



“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- Enough with the background - let's find the bug
- Idea 1: `uint32_t` canonical index overflow
 - Won't work, too much memory needed

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- Enough with the background - let's find the bug
- Idea 1: `uint32_t` canonical index overflow
 - Won't work, too much memory needed
- Idea 2: Confusion between canonical type index vs. module type index?
 1. Two distinct ways to represent types, **where both are just plain integers**
 2. Canonical type index NOT bound by `kV8MaxWasmTypes`

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- Check xrefs on relevant functions & data structures

▼ Definitions (1 result) | Show only

- ▼ v8/src/wasm/wasm-module.h (1 result)
676: std::vector<uint32_t> isorecursive_canonical_type_ids;

▼ Other References (57 results) | Show only

- ▼ v8/src/compiler/wasm-compiler.cc (1 result)
7366: module->isorecursive_canonical_type_ids[type.ref_index()];
- ▼ v8/src/runtime/runtime-wasm.cc (4 results)
171: module->isorecursive_canonical_type_ids[type.ref_index()];
480: module->isorecursive_canonical_type_ids[function.sig_index];
583: module->isorecursive_canonical_type_ids[module->functions[func_index]]
1362: if (module->isorecursive_canonical_type_ids[wti->type_index()] != expected) {
- ▼ v8/src/wasm/baseline/liftoff-compiler.cc (1 result)
8213: decoder->module_->isorecursive_canonical_type_ids[imm.sig_imm.index];

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- Object typecheck at JS-to-WASM boundary (for reference types)
- We construct a `ValueType::RefMaybeNull()` out of a `canonical_index`

```
if (type.has_index()) {
    DCHECK_NOT_NULL(module);
    uint32_t canonical_index =
        module->isorecursive_canonical_type_ids[type.ref_index()];
    type = wasm::ValueType::RefMaybeNull(canonical_index,
                                         type.nullability());
}

Node* inputs[] = {
    input, mcgraph()->IntPtrConstant(
        IntToSmi(static_cast<int>(type.raw_bit_field())));
}

return BuildCallToRuntimeWithContext(Runtime::kWasmJSToWasmObject,
                                    js_context, inputs, 2);
```



“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- ValueType passed down to JSToWasmObject():

```
MaybeHandle<Object> JSToWasmObject(Isolate* isolate, Handle<Object> value,
                                     ValueType expected_canonical,
                                     const char** error_message) {
    // ...
    switch (expected_canonical.heap_representation_non_shared()) {
        // ...
        default: { // [!] ref (concrete type)
            auto type_canonicalizer = GetWasmEngine()->type_canonicalizer();

            if (WasmExportedFunction::IsWasmExportedFunction(*value)) {
                //...
            } else if (IsWasmStruct(*value) || IsWasmArray(*value)) {
                auto wasm_obj = Handle<WasmObject>::cast(value);
                Tagged<WasmTypeInfo> type_info = wasm_obj->map()->wasm_type_info();
                uint32_t real_idx = type_info->type_index();
                const WasmModule* real_module =
                    WasmInstanceObject::cast(type_info->instance())->module();
                uint32_t real_canonical_index =
                    real_module->isorecursive_canonical_type_ids[real_idx];
                if (!type_canonicalizer->IsCanonicalSubtype(
                    real_canonical_index, expected_canonical.ref_index()))
                    *error_message = "object is not a subtype of expected type";
                return {};
            }
            return value;
        }
    }
}
```

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- **ValueType** passed down to `JSToWasmObject()`:
 - Fetching the canonical index back from a `HeapTypeField`?

```
constexpr HeapType::Representation heap_representation() const {
    DCHECK(is_object_reference());
    return static_cast<HeapType::Representation>(
        HeapTypeField::decode(bit_field_));
}
constexpr HeapType::Representation heap_representation_non_shared() const {
    DCHECK(is_object_reference());
    return HeapType(heap_representation()).representation_non_shared();
}
constexpr HeapType heap_type() const {
    DCHECK(is_object_reference());
    return HeapType(heap_representation());
}
constexpr uint32_t ref_index() const {
    DCHECK(has_index());
    return HeapTypeField::decode(bit_field_);
}
```

```
// Extracts the bit field from the value.
static constexpr T decode(U value) {
    return static_cast<T>((value & kMask) >> kShift);
}
```



“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- `ValueType` passed down to `JSToWasmObject()`:
 - Canonical index is stored in `HeapType`, a **20-bit wide bitfield!** ($2^{20} = 1,048,576$)

```
static constexpr int kLastUsedBit = 25;
static constexpr int kKindBits = 5;
static constexpr int kHeapTypeBits = 20;

static const intptr_t kBitFieldOffset;

private:
    // {hash_value} directly reads {bit_field_}.
    friend size_t hash_value(ValueType type);

    using KindField = base::BitField<ValueKind, 0, kKindBits>;
    using HeapTypeField = KindField::Next<uint32_t, kHeapTypeBits>;
    // Marks a type as a canonical type which uses an index relative to its
    // recursive group start. Used only during type canonicalization.
    using CanonicalRelativeField = HeapTypeField::Next<bool, 1>;
```



“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- `ValueType` passed down to `JSToWasmObject()`:
 - Canonical index is stored in `HeapType`, a **20-bit wide bitfield!** ($2^{20} = 1,048,576$)
- 20 bits?
 - Enough to store all valid **module-specific HeapTypes**:
 - Type indices: $0 \sim 999,999$ ($= \text{kV8MaxWasmTypes} - 1$)
 - Generic heap types: $1,000,000 \sim 1,000,0xx$
 - Internal types (invalid): $1,000,0xx + 1$ (`kBottom`)
 - NOT enough to store **canonical type indices**!
 - Canonical type indices: `uint32_t`, bounded only by host memory limits

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024



“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- Bug #1: Canonical type index truncated to 20 bits!
- Effect: Broken typecheck on JS-to-Wasm boundary, where:
 - Intended: Typecheck against `ref T`, where $t = (n \ll 20) + k$ ($0 \leq k < 1E6$)
 - Actual: Typecheck against `ref K` for type `K` with canonical type index `k`
- Result: Universal WASM type confusion `K -> T`

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- What if $t = (n \ll 20) + k$ ($1E6 \leq k < 2^{20}$), i.e. a generic type index?

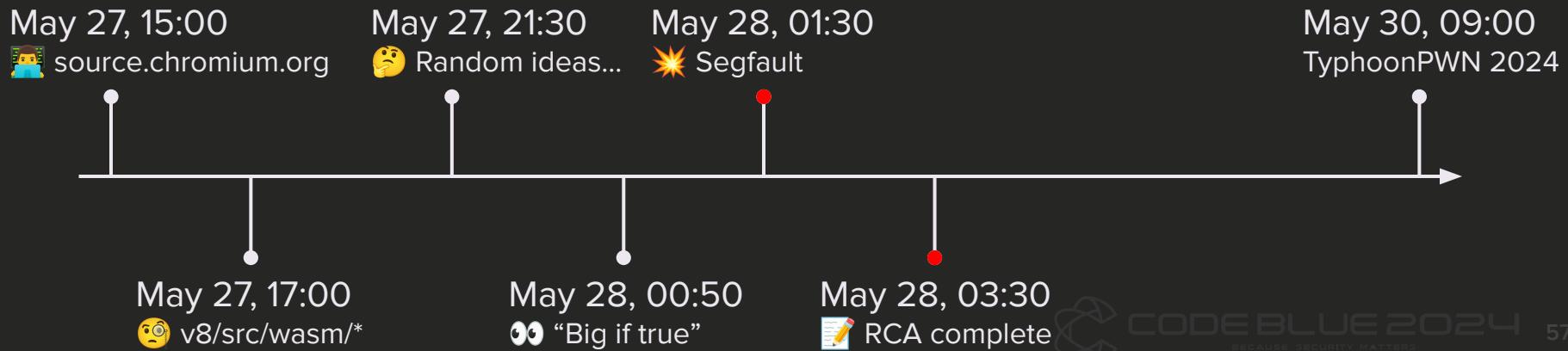
```
MaybeHandle<Object> JSToWasmObject(Isolate* isolate, Handle<Object> value,
                                     ValueType expected_canonical,
                                     const char** error_message) {
    // ...
    switch (expected_canonical.heap_representation_non_shared()) {
        // ...
        case HeapType::kAny: {
            if (IsSmi(*value)) return CanonicalizeSmi(value, isolate);
            if (IsHeapNumber(*value)) {
                return CanonicalizeHeapNumber(value, isolate);
            }
            if (!IsNull(*value, isolate)) return value;
            *error_message = "null is not allowed for (ref any)";
            return {};
        }
        // ...
        default: { // [!] ref (concrete type)
            // ...
        }
    }
}
```



“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

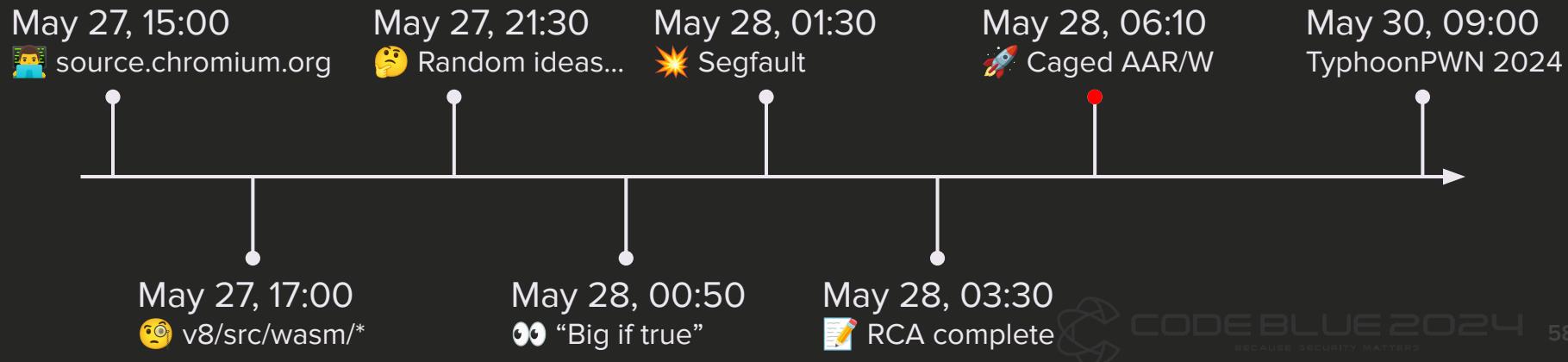
- Bug #2: Canonical type index confused as generic `HeapType!`
 - Canonical type indices are never meant to be stored on `ValueType`
- Effect: Broken typecheck on JS-to-Wasm boundary, where:
 - Intended: Typecheck against `ref T`, where `t = (n << 20) + kAny`
 - Actual: Typecheck against `ref any`
- Result: Universal WASM type confusion `any -> T`

“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024



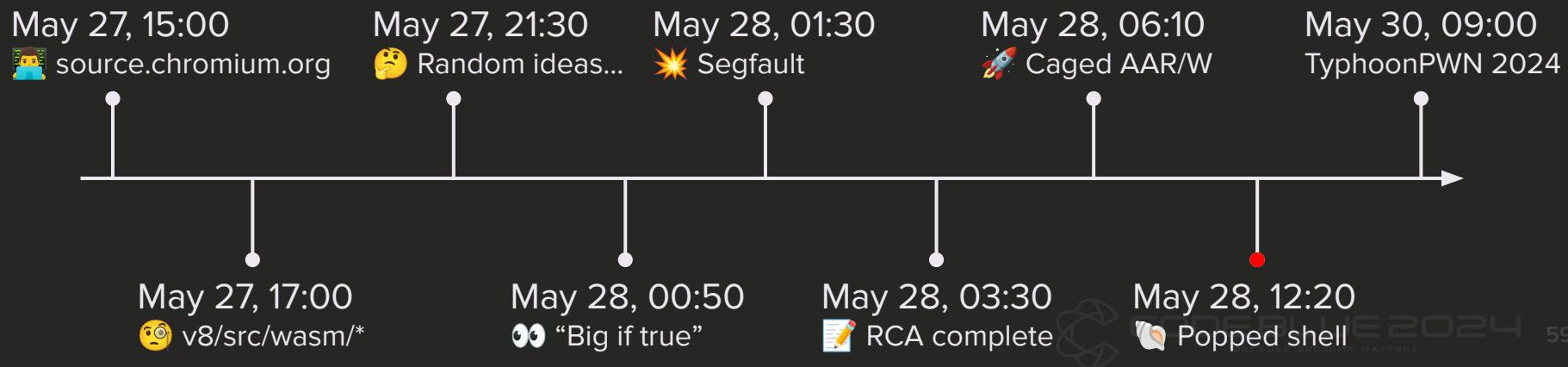
“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- In-sandbox exploit? Exactly same as CVE-2024-2887
 - Arbitrary caged RW, `addrOf()`, `fakeObj()` primitives instantly acquired



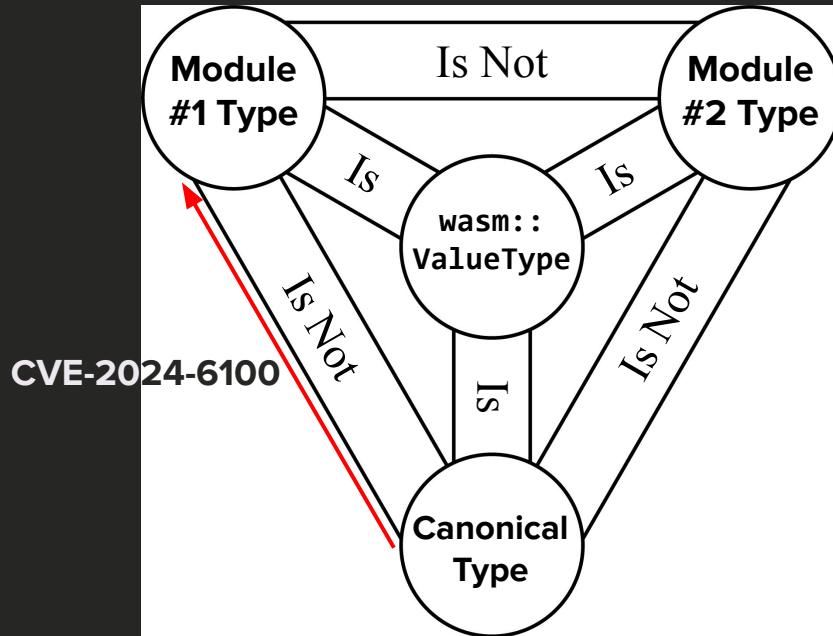
“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- In-sandbox exploit? Exactly same as CVE-2024-2887
 - Arbitrary caged RW, `addrOf()`, `fakeObj()` primitives instantly acquired
- V8 sandbox escape? Just Use PartitionAlloc™ (feat. b/40238514 from p2o2024)
 - Common misconception that V8 sandbox has no raw pointers – not with PA!



The wasm::ValueType Trinity

- Note how this isn't a one-off bug – it's a huge design issue



The Sequel: CVE-2024-9859 (v8CTF M126, later found ITW)

The Sequel: CVE-2024-9859

- CVE-2024-6100: canonical index → module-specific index confusion
- Other way around – module-specific index → canonical index??

The Sequel: CVE-2024-9859

Merged [5613375](#) ▾ [wasm] Add missing type canonicalization for exceptions JS API 

Monday, Jun 10, 2024, 9:25:28 PM UTC+09:00 [SHOW ALL](#) ▾ [SIGN IN](#)

Submitted	9:25 PM
Owner	 Thibaud Michaud
Uploader	 V8 LUCI CQ
Reviewers	 Jakob Kumme... +1  V8 LUCI CQ
CC	 v8-reviews@g...  wasm-v8@g...
Repo Branch	v8/v8 main
Hashtags	wasm

[wasm] Add missing type canonicalization for exceptions JS API

When we encode a JS value in a wasm exception, canonicalize the type stored in the tag's signature first. Canonicalize it using the tag's original module by storing the instance on the tag object.

R=jkummerow@chromium.org

Bug: [346197738](#)

Change-Id: [I7575fd79c792d98e4a11c00b466700f0ab82d164](#)

Reviewed-on: <https://chromium-review.googlesource.com/c/v8/v8/+/5613375>

Commit-Queue: Thibaud Michaud <thibaudm@chromium.org>

Reviewed-by: Jakob Kummerow <jkummerow@chromium.org>

Cr-Commit-Position: refs/heads/main@{#94335}

The Sequel: CVE-2024-9859

```
case i::wasm::kRef:
case i::wasm::kRefNull: {
    const char* error_message;
    i::Handle<i::Object> value_handle = Utils::OpenHandle(*value);

    if (type.has_index()) {
        // Canonicalize the type using the tag's original module.
        i::Tagged<i::HeapObject> maybe_instance = tag_object->instance();
        CHECK(!i::IsUndefined(maybe_instance));
        auto instance = i::WasmInstanceObject::cast(maybe_instance);
        const i::wasm::WasmModule* module = instance->module();
        uint32_t canonical_index =
            module->isorecursive_canonical_type_ids[type.ref_index()];
        type = i::wasm::ValueType::RefMaybeNull(canonical_index,
                                                type.nullability());
    }

    if (!internal::wasm::JSToWasmObject(i_isolate, value_handle, type,
                                         &error_message)
        .ToHandle(&value_handle)) {
        thrower->TypeError("%s", error_message);
        return;
    }
    values_out->set(index++, *value_handle);
    break;
}
```



The Sequel: CVE-2024-9859

1. Wasm module exports exception signature (i.e. Tag) with module-specific types

The Sequel: CVE-2024-9859

1. Wasm module exports exception signature (i.e. Tag) with module-specific types
2. An exception is created with `WebAssembly.Exception()` with the export tag
 - o Typechecked with module-specific index → canonical index confusion

The Sequel: CVE-2024-9859

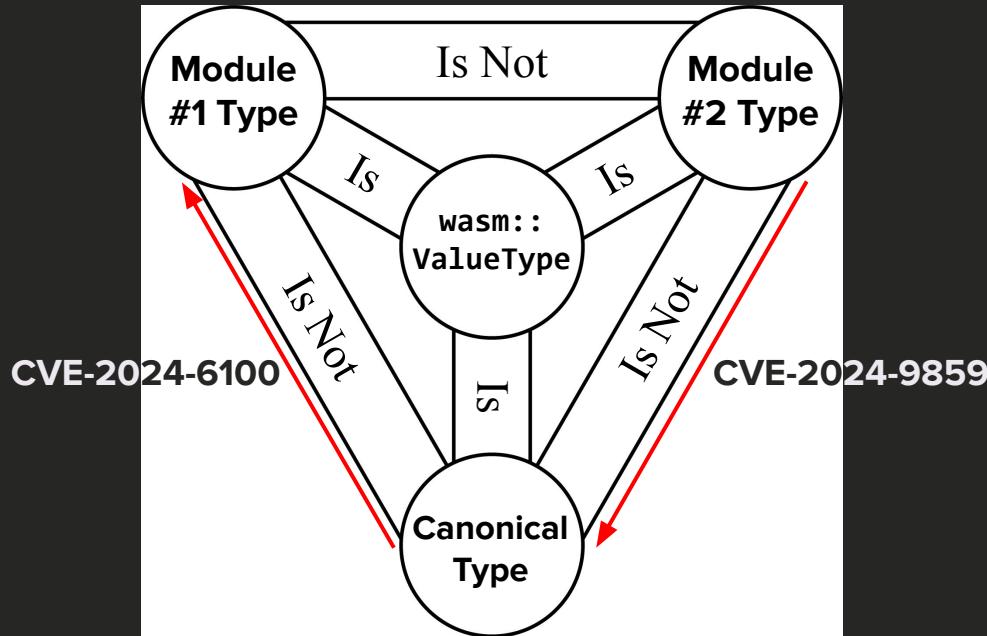
1. Wasm module exports exception signature (i.e. Tag) with module-specific types
2. An exception is created with `WebAssembly.Exception()` with the export tag
 - o Typechecked with module-specific index → canonical index confusion
3. Catch the exception within Wasm to unpack values as module-specific types
4. ...?
5. 

The Sequel: CVE-2024-9859

- Q: How did this go unknown? Where are the unit tests??
- A: Simple, those tests don't use WasmGC types
 - Different feature extension proposal: Garbage Collection vs. Exception Handling
 - Lack of integration tests between feature extensions

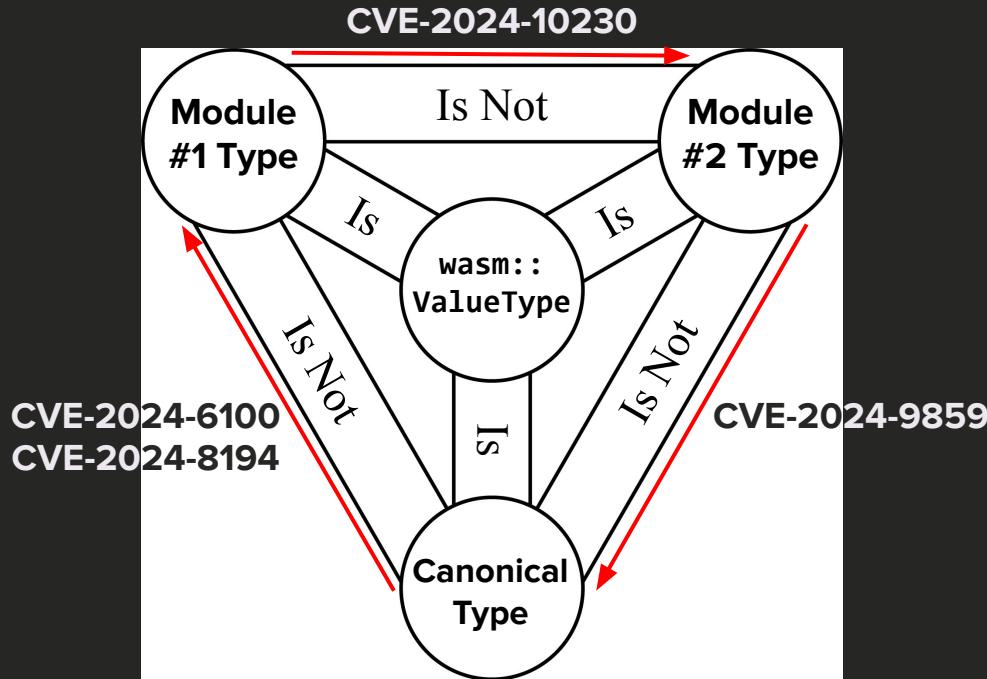
The wasm::ValueType Trinity

- Type confusion, two ways



The wasm::ValueType Trinity

- Type confusion, all ways (and not just once!)



Note that these are only the bugs that I have found || exploited.

Typos Gone Wild: CVE-2024-6779



CODE BLUE 2024
BECAUSE SECURITY MATTERS

Typos Gone Wild: CVE-2024-6779

- A short trip to Wasm Turbofan implementation to find other **truncation** issues
 - Caching logic for last accessed memory base & size

```
Node* WasmGraphBuilder::MemStart(uint32_t mem_index) {
    DCHECK_NOT_NULL(instance_cache_);
    V8_ASSUME(cached_memory_index_ == kNoCachedMemoryIndex ||  
              cached_memory_index_ >= 0);
    if (mem_index == static_cast<uint8_t>(cached_memory_index_)) {  
        return instance_cache_->mem_start;
    }
    return LoadMemStart(mem_index);
}

Node* WasmGraphBuilder::MemSize(uint32_t mem_index) {
    DCHECK_NOT_NULL(instance_cache_);
    V8_ASSUME(cached_memory_index_ == kNoCachedMemoryIndex ||  
              cached_memory_index_ >= 0);
    if (mem_index == static_cast<uint8_t>(cached_memory_index_)) {  
        return instance_cache_->mem_size;
    }
    return LoadMemSize(mem_index);
}
```

Typos Gone Wild: CVE-2024-6779

- Cached memory index confusion
 1. Access memory index **0x100**
 2. Access memory index **0** ($\text{== static_cast<uint8_t>(0x100)}$)
 - Accessed using cached memory base & length of memory index 0x100
- How does this become a security issue?

Typos Gone Wild: CVE-2024-6779

- Optimization – if `offset <= min_size`, elide comparison against current size
 - Broken invariant, remaining size `effective_size` subtraction overflow!

```
Node* mem_size = MemSize(memory->index); <= Cached size of memory[0x100]

Node* end_offset_node = mcgraph_->IntPtrConstant(end_offset);
if (end_offset > memory->min_memory_size) { <= Min size of memory[0]
    // The end offset is larger than the smallest memory.
    // Dynamically check the end offset against the dynamic memory size.
    Node* cond = gasm_->UintLessThan(end_offset_node, mem_size);
    TrapIfFalse(wasm::kTrapMemOutOfBounds, cond, position);
}

// This produces a positive number since {end_offset <= min_size <= mem_size}.
Node* effective_size = gasm_->IntSub(mem_size, end_offset_node); <= Overflow!

// Introduce the actual bounds check.
Node* cond = gasm_->UintLessThan(converted_index, effective_size);
TrapIfFalse(wasm::kTrapMemOutOfBounds, cond, position);
return {converted_index, BoundsCheckResult::kDynamicallyChecked};
```

Typos Gone Wild: CVE-2024-6779

- Great, arbitrary index OOB read/write from Wasm memory base :)
- Exploitable?

Typos Gone Wild: CVE-2024-6779

- Great, arbitrary index OOB read/write from Wasm memory base :)
- Not-so-great reasons:
 - Index limited to uint32
 - Wasm memory padded to 8GB w/ guard page for OOB trapping mechanism
- Exploitable? Unexploitable??

Typos Gone Wild: CVE-2024-6779

- Great, arbitrary index OOB read/write from Wasm memory base :)
- Not-so-great reasons:
 - Index limited to uint32
 - Wasm memory padded to 8GB w/ guard page for OOB trapping mechanism
=> **On Android, no guard page** due to signal safety issues
- ~~Exploitable? Unexploitable?? Exploitable???~~

Typos Gone Wild: CVE-2024-6779

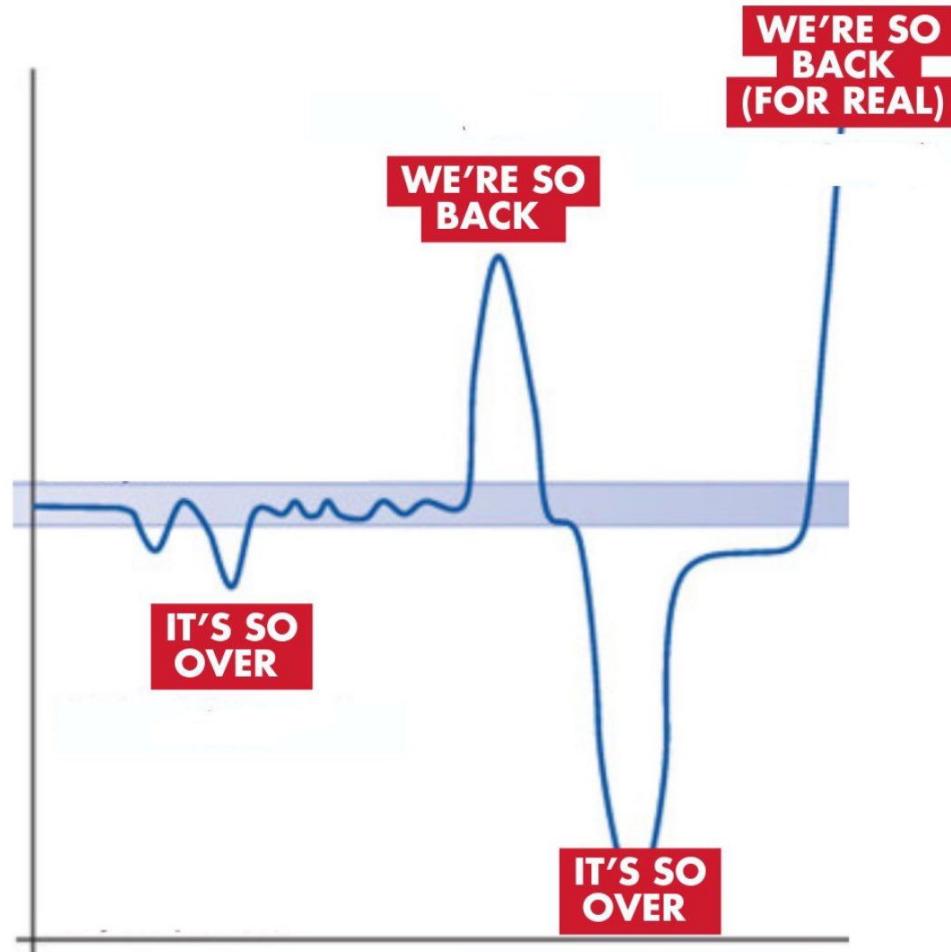
- Great, arbitrary index OOB read/write from Wasm memory base :)
- Not-so-great reasons:
 - Index limited to uint32
 - Wasm memory padded to 8GB w/ guard page for OOB trapping mechanism
 - => **On Android, no guard page** due to signal safety issues
 - => But there's nothing useful to overwrite?
It's allocated after ArrayBuffer PartitionAlloc...
- ~~Exploitable? Unexploitable?? Exploitable???~~ Unexploitable????

Typos Gone Wild: CVE-2024-6779

- Great, arbitrary index OOB read/write from Wasm memory base :)
- Not-so-great reasons:
 - Index limited to uint32
 - Wasm memory padded to 8GB w/ guard page for OOB trapping mechanism
 - => **On Android, no guard page** due to signal safety issues
 - => But there's nothing useful to overwrite?
 - It's allocated after ArrayBuffer PartitionAlloc...
 - => In “some cases”, it's between V8 cage & ArrayBuffer PartitionAlloc!!
- ~~Exploitable? Unexploitable?? Exploitable???~~ ~~Unexploitable????~~ Exploitable!

Typos Gone

- Great, arbitrary typos
- Not-so-great typos
 - Index linking
 - => With typos
 - Wasm memory
 - => On Address
 - => But the address
 - It's also a pointer
 - => In "so over"
- Exploitability?



e :)

ture is staged...

mechanism

loc!!

Exploitable!

Typos Gone Wild: CVE-2024-6779

- Conditions for Wasm memory to be allocated between V8 cage & PA
 - On Android, address is almost always fixed due to randomization bug^{*} + Zygote

Either way, uses `OS::GetRandomMmapAddr` to obtain address hint to map the virtual memory

```
#if V8_TARGET_ARCH_X64 || V8_TARGET_ARCH_ARM64  
  
    raw_addr &= uint64_t{0x3FFFFFFF000};
```

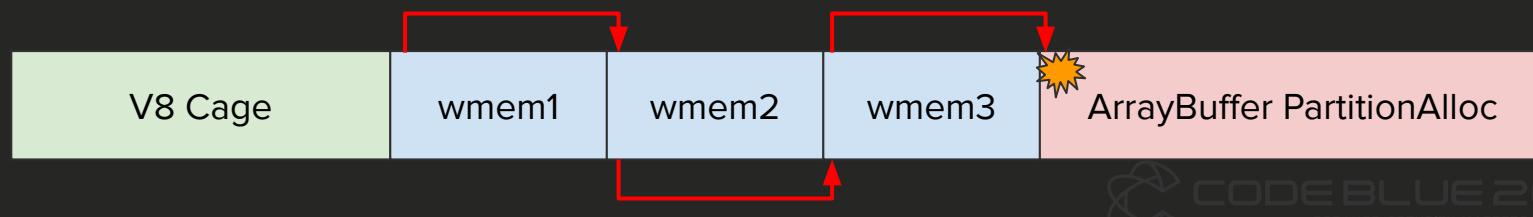
Address is random and masked to 46 bits. On Arm64, address space is 39 bits, so hint is almost certain to fail and the first free address is used => Fixed once per boot (Memory layout depends on Zygote on Android)

Typos Gone Wild: CVE-2024-6779

- Conditions for Wasm memory to be allocated between V8 cage & PA
 - On Android, address is almost always fixed due to randomization bug^{*} + Zygote
 - Both the V8 Sandbox & V8 cage is allocated with alignment of **4GiB**
 - ArrayBuffer PartitionAlloc pool is allocated with alignment of **16GiB**

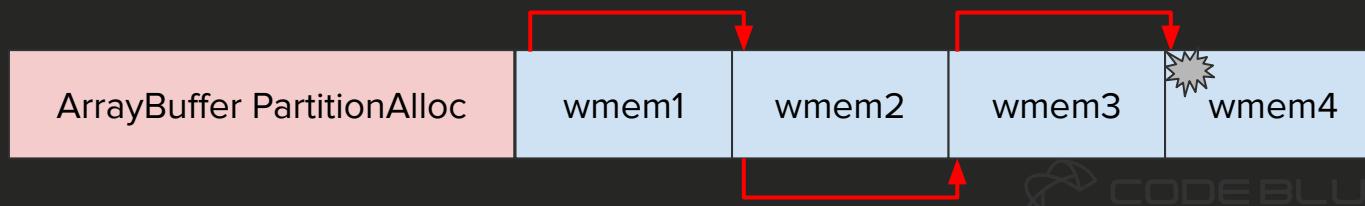
Typos Gone Wild: CVE-2024-6779

- Conditions for Wasm memory to be allocated between V8 cage & PA
 - On Android, address is almost always fixed due to randomization bug^{*} + Zygote
 - Both the V8 Sandbox & V8 cage is allocated with alignment of **4GiB**
 - ArrayBuffer PartitionAlloc pool is allocated with alignment of **16GiB**
- **75%** chance to have a gap between the V8 cage & PartitionAlloc
 - This gap can be reclaimed with Wasm memory!
 - On any cases, we can probe the layout & determine exploitability w/o crashing
 - Fill up each potential 4GiB (+1) with Wasm memory, OOB read to probe if it's before PA



Typos Gone Wild: CVE-2024-6779

- Conditions for Wasm memory to be allocated between V8 cage & PA
 - On Android, address is almost always fixed due to randomization bug^{*} + Zygote
 - Both the V8 Sandbox & V8 cage is allocated with alignment of **4GiB**
 - ArrayBuffer PartitionAlloc pool is allocated with alignment of **16GiB**
- **75%** chance to have a gap between the V8 cage & PartitionAlloc
 - This gap can be reclaimed with Wasm memory!
 - On any cases, we can probe the layout & determine exploitability w/o crashing
 - Fill up each potential 4GiB (+1) with Wasm memory, OOB read to probe if it's before PA



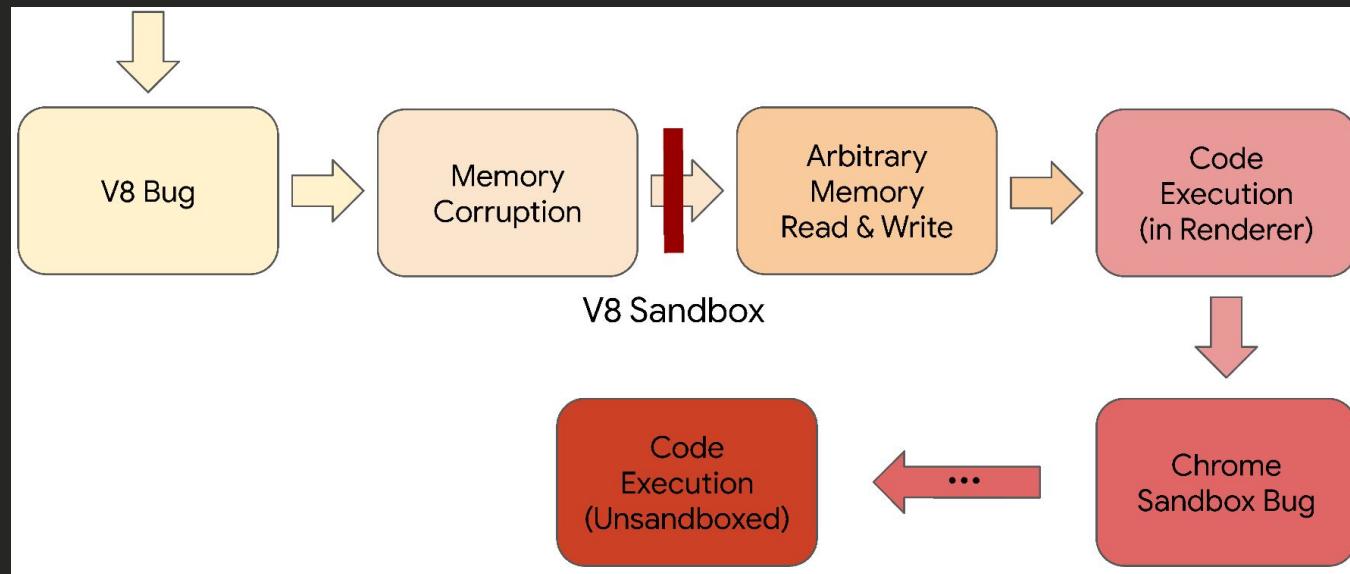
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses



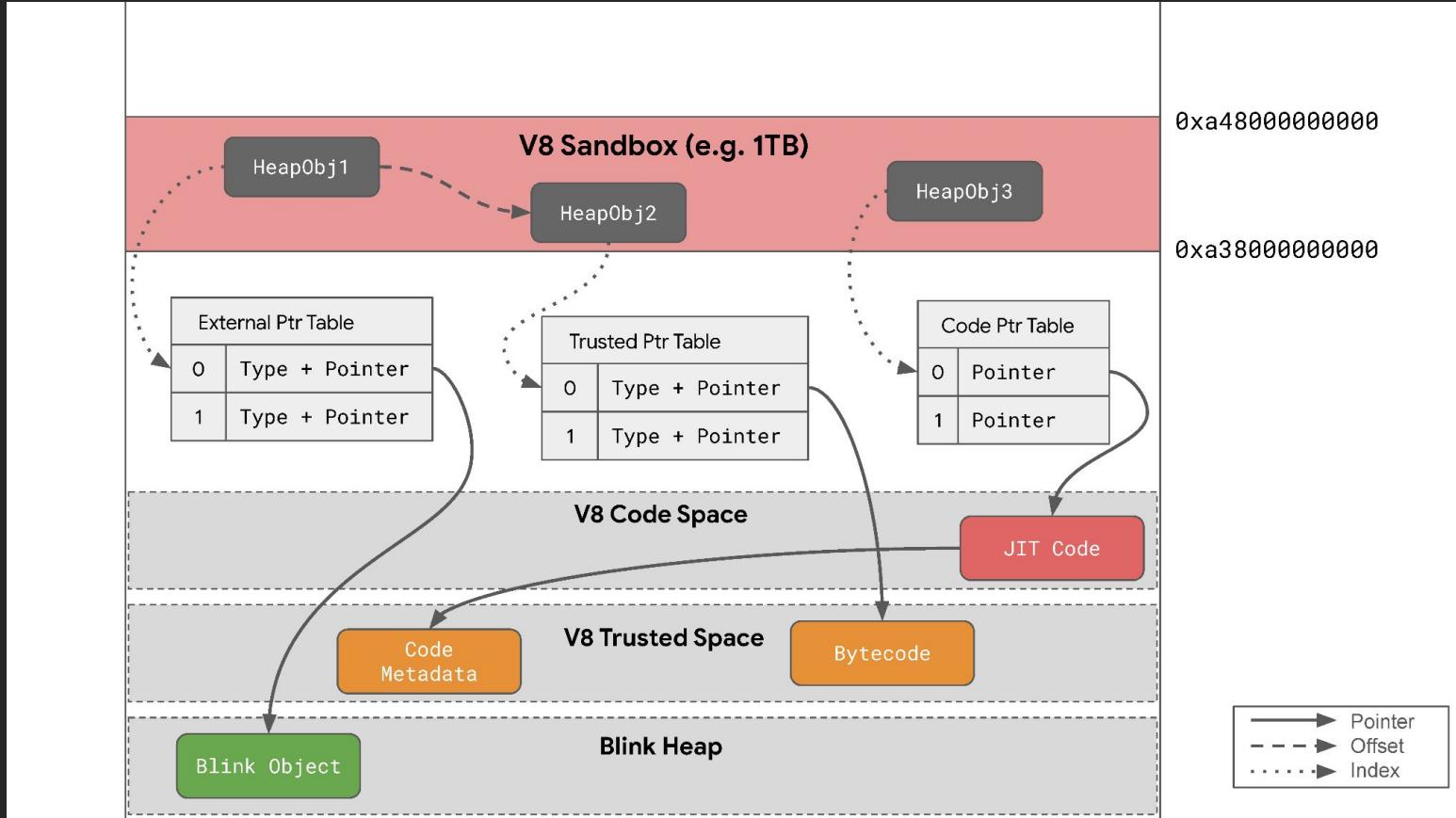
CODE BLUE 2024
BECAUSE SECURITY MATTERS

Crash Course on V8 Sandbox

- V8 Sandbox:
 - Software fault isolation mechanism to prevent memory corruptions from within the sandbox region evolving into arbitrary writes outside of sandbox



Crash Course on V8 Sandbox



“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Wasm is a goldmine of V8 Sandbox bypasses
 - What makes it so vulnerable?
 - What are the common patterns?

“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

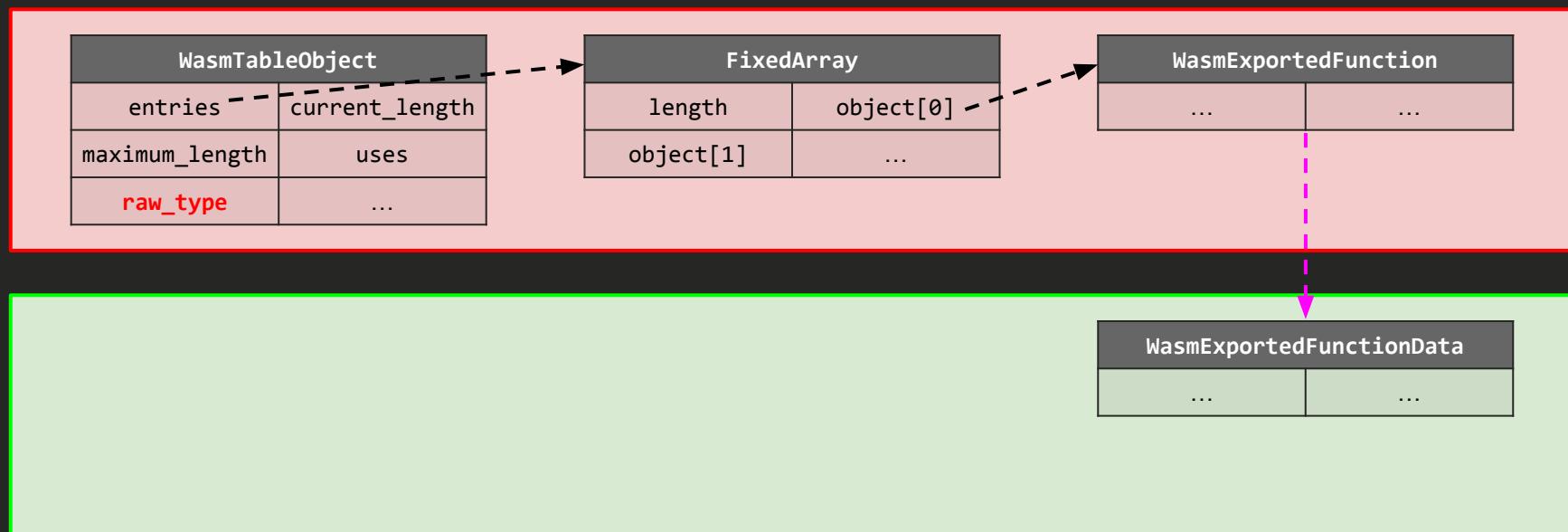
- Wasm is a goldmine of V8 Sandbox bypasses
 - What makes it so vulnerable?
 - What are the common patterns?
- Key idea:
 - Reference types are represented as full 64bit pointers at:
 - Within a Wasm function
 - Across Wasm function calls – **function signature confusion leads to v8sbx bypass!**
 - Everything is an object – memory, funcrefs, function tables, etc.
 - Anything that could be modified must not be trusted

“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Wasm is a goldmine of V8 Sandbox bypasses
 - What makes it so vulnerable?
 - What are the common patterns?
- Key idea:
 - Reference types are represented as full 64bit pointers at:
 - Within a Wasm function
 - Across Wasm function calls – **function signature confusion leads to v8sbx bypass!**
 - Everything is an object – memory, funcrefs, function tables, etc.
 - Anything that could be modified must not be trusted
 - The paradigm shift: V8 sandbox & JS is “userspace”, everything else “kernel”
 - We need to reason about “non-renderer issues” – **“double fetch” within v8sbx?**
 - “Drivers”, i.e. embedder implementations, which is difficult to reason about from V8

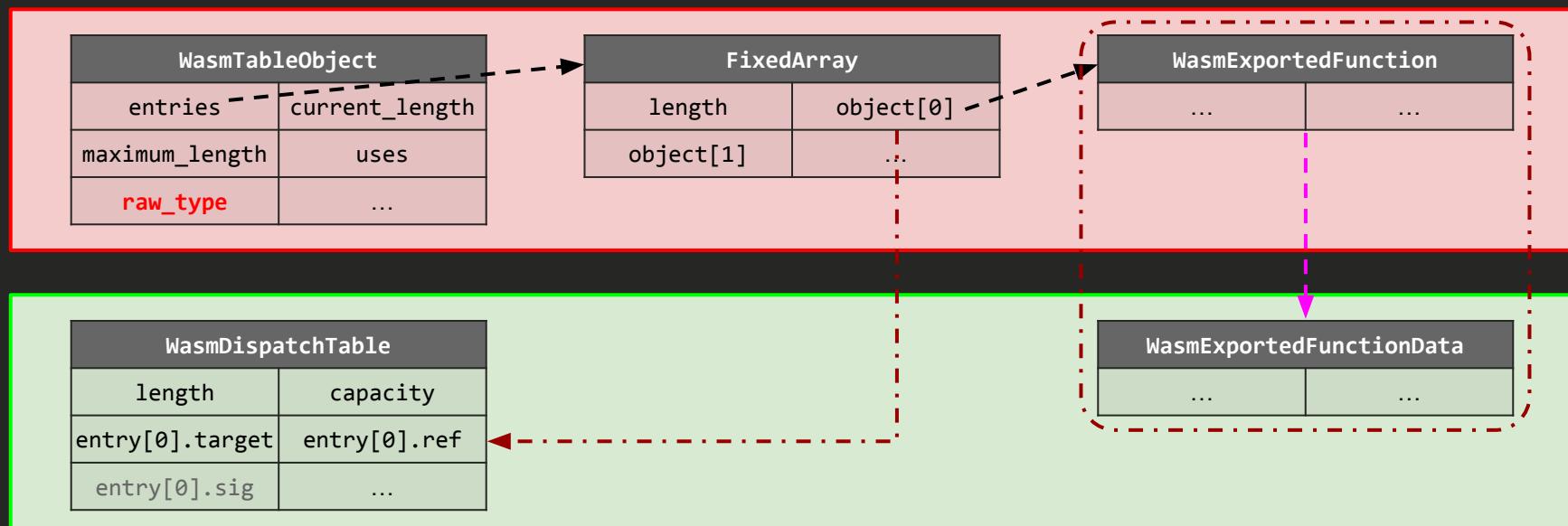
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 1: Code metadata (i.e. signatures) corruption
 - b/348793147: Missing signature check when importing function tables



“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 1: Code metadata (i.e. signatures) corruption
 - b/348793147: Missing signature check when importing function tables

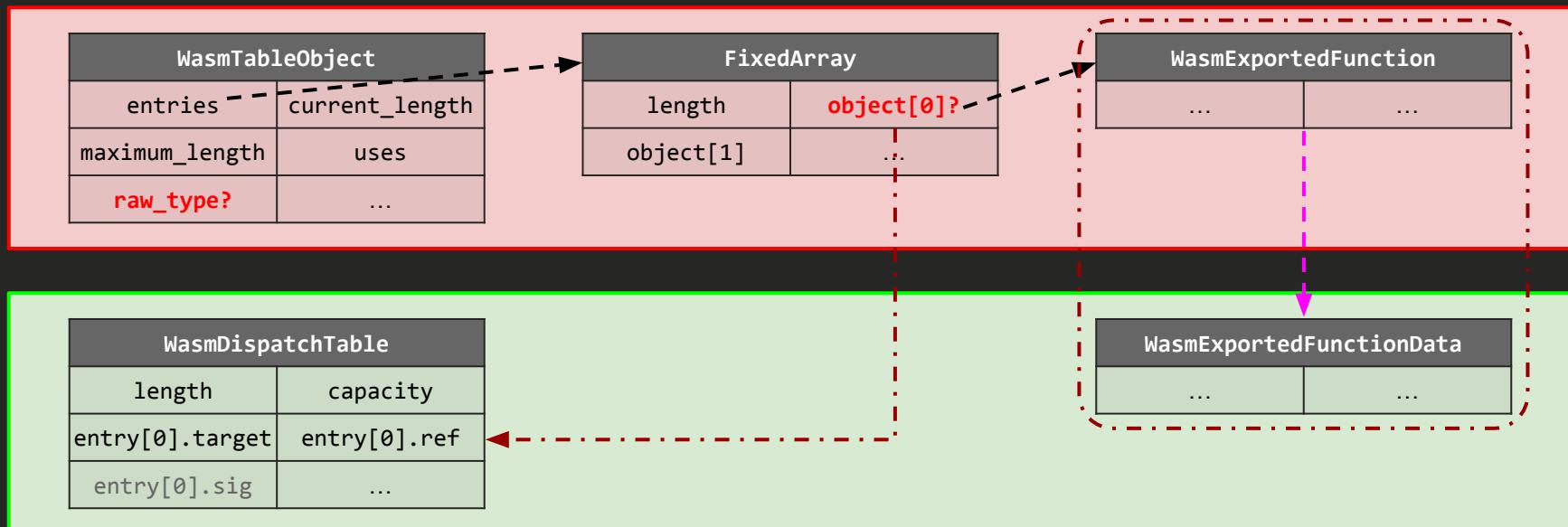


Invariant: `entry[i].sig <: table type`



“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 1: Code metadata (i.e. signatures) corruption
 - b/348793147: Missing signature check when importing function tables



Invariant: `entry[i].sig <: table type`

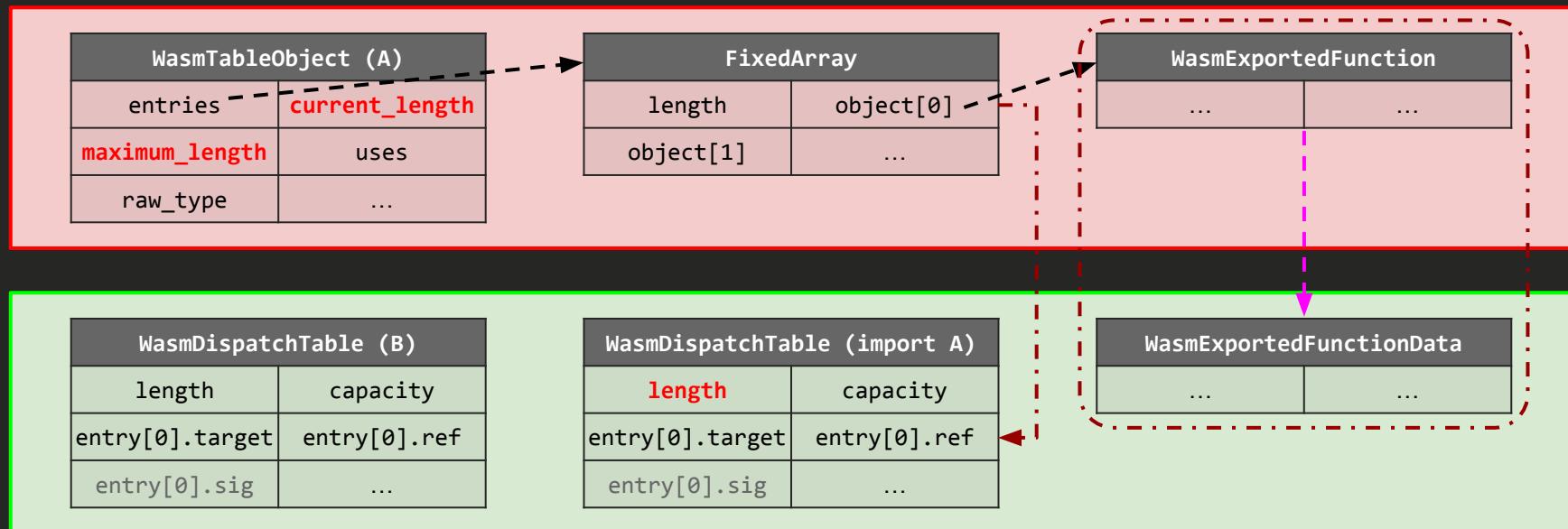


“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 1: Code metadata (i.e. signatures) corruption
 - b/348793147: Missing signature check when importing function tables
 - b/350292240: Generic func table runtime typecheck bypass via type info corruption
 - Unfixed, but public as part of exploit chain for “Typos Gone Wild: CVE-2024-6779”
 - b/352689356: Missing signature **SBXCHECK()** in Turbofan `call_ref` – wontfix’d
 - Wasm Turbofan expected to be obsolete *Soon™*
 - b/354408144: Wasm-to-JS wrapper serialized signature corruption
 - Trusted-to-untrusted reference
 - b/354355045: JS-to-Wasm **sbxcheck()** bypass
 - Trusted | Untrusted type union, fallback to fake untrusted object

“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 2: Untrusted indices
 - b/349502157: Table set SBXCHECK_LT() bypass



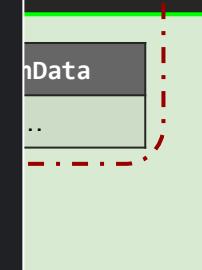
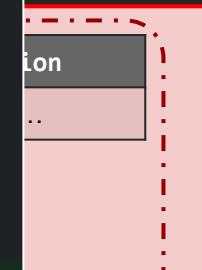
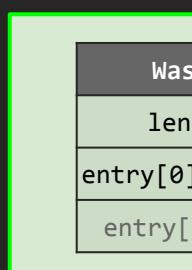
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 2: Untrusted indices

[wasm] Avoid OOB writes of the WasmDispatchTable

```
void WasmDispatchTable::Set(int index, Tagged<Object> ref, Address call_target,
                            int sig_id) {
    if (ref == Smi::zero()) {
        DCHECK_EQ(kNullAddress, call_target);
        Clear(index);
        return;
    }

    SBXCHECK_LT(index, length());
    DCHECK(IsWasmApiFunctionRef(ref) || IsWasmTrustedInstanceData(ref));
    DCHECK_EQ(ref == Smi::zero(), call_target == kNullAddress);
    const int offset = OffsetOf(index);
    WriteProtectedPointerField(offset + kRefBias, TrustedObject::cast(ref));
    CONDITIONAL_WRITE_BARRIER(*this, offset + kRefBias, ref,
                             UPDATE_WRITE_BARRIER);
    WriteField<Address>(offset + kTargetBias, call_target);
    WriteField<int>(offset + kSigBias, sig_id);
}
```



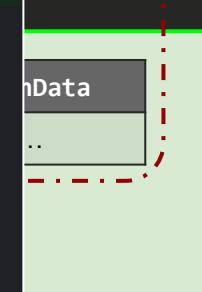
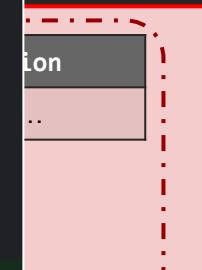
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 2: Untrusted indices

[wasm] Avoid OOB writes of the WasmDispatchTable

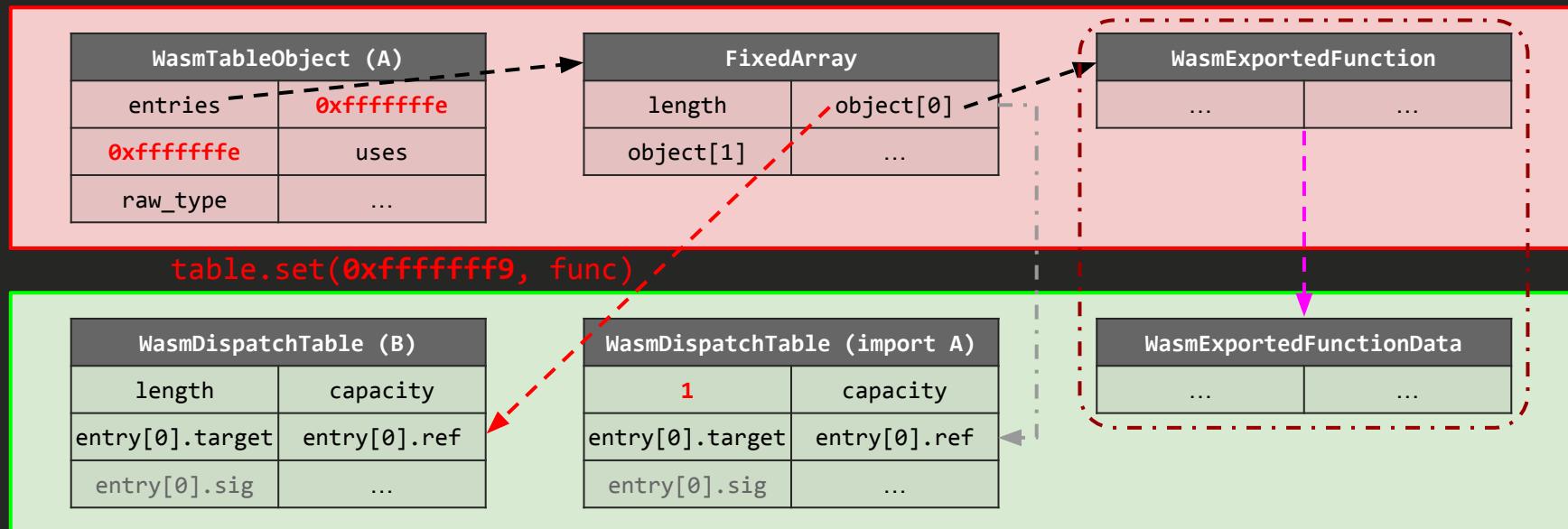
```
void WasmDispatchTable::Set(int index, Tagged<Object> ref, Address call_target,
                            int sig_id) {
    if (ref == Smi::zero()) {
        DCHECK_EQ(kNullAddress, call_target);
        Clear(index);
        return;
    } // The current length of this dispatch table.
    inline int length() const;

    SBXCHECK_LT(index, length());
    DCHECK(IsWasmApiFunctionRef(ref) || IsWasmTrustedInstanceData(ref));
    DCHECK_EQ(ref == Smi::zero(), call_target == kNullAddress);
    const int offset = OffsetOf(index);
    WriteProtectedPointerField(offset + kRefBias, TrustedObject::cast(ref));
    CONDITIONAL_WRITE_BARRIER(*this, offset + kRefBias, ref,
                             UPDATE_WRITE_BARRIER);
    WriteField<Address>(offset + kTargetBias, call_target);
    WriteField<int>(offset + kSigBias, sig_id);
}
```



“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 2: Untrusted indices
 - b/349502157: Table set SBXCHECK_LT() bypass



“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

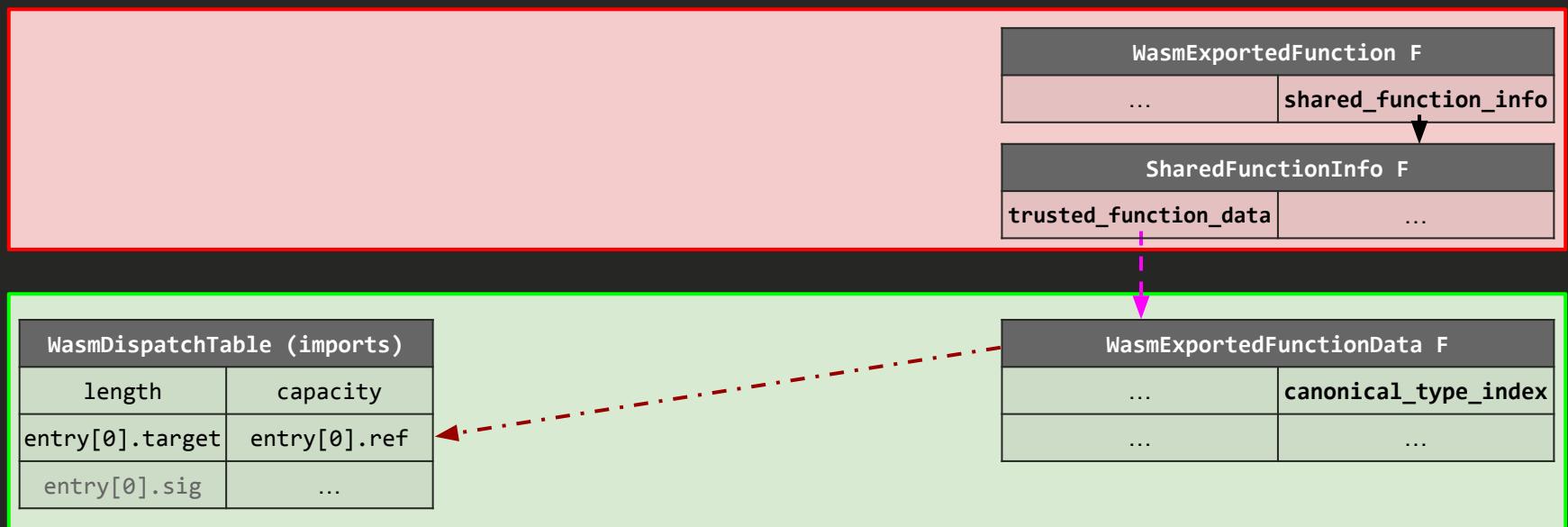
- Case 2: Untrusted indices
 - b/349502157: Table set SBXCHECK_LT() bypass
 - b/350628675: OOB access from a ProtectedFixedArray

“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Case 3: Broken invariants
 - Similar to what we’ve seen in “Typos Gone Wild: CVE-2024-6779”
- Case 4: Transplantation^{*} / Extraction of trusted handles
 - Replacing / removing references to trusted objects

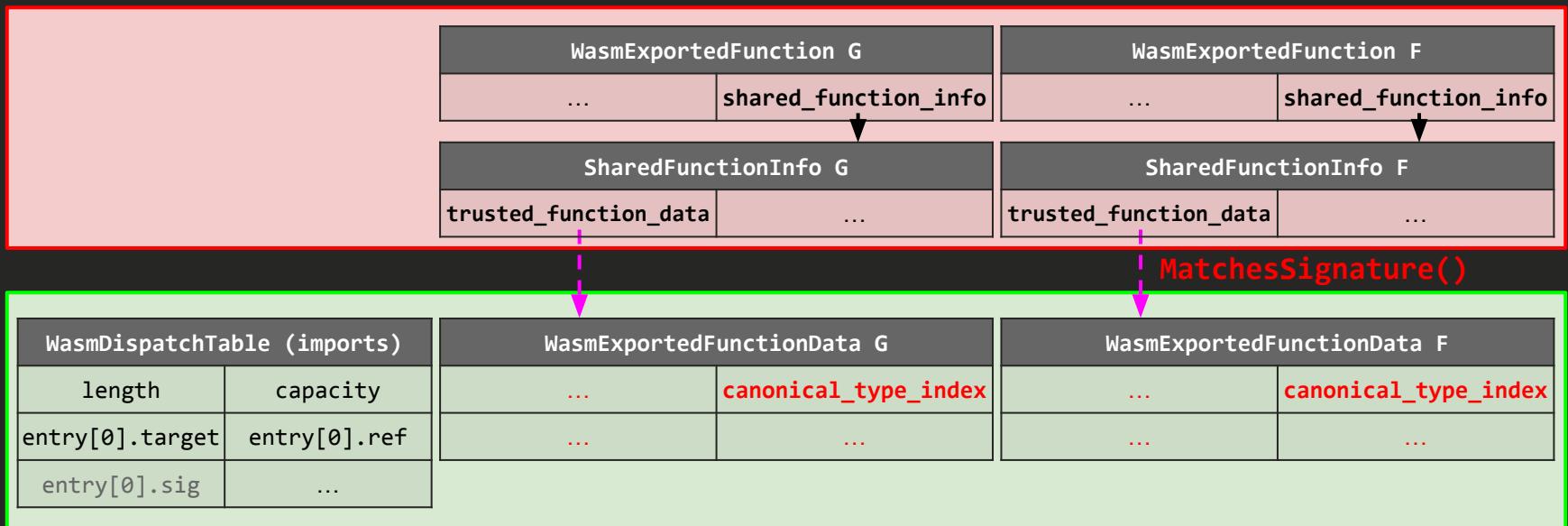
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Variant: Double fetch / TOCTOU
 - Case 1' + 4' – b/349529650: Function import signature check race



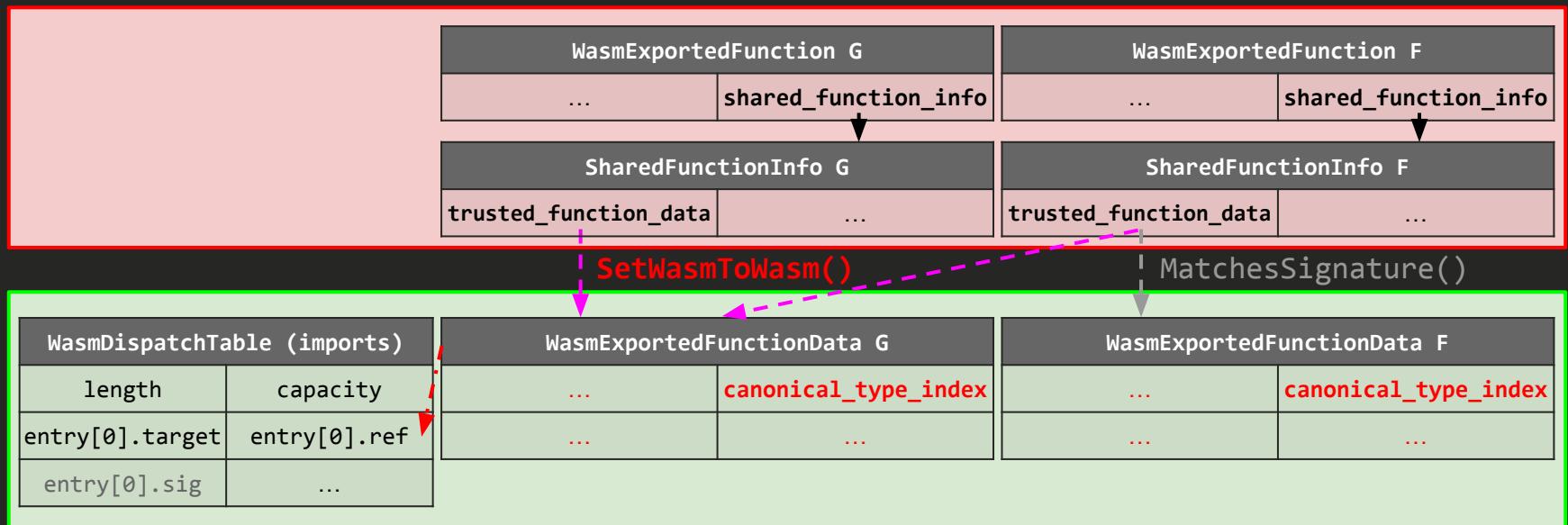
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Variant: Double fetch / TOCTOU
 - Case 1' + 4' – b/349529650: Function import signature check race



“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Variant: Double fetch / TOCTOU
 - Case 1' + 4' – b/349529650: Function import signature check race



“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Variant: Double fetch / TOCTOU
 - Case 1' + 4' – b/349529650: Function import signature check race
 - Case 3' – b/352446085: Wasm memory64 bounds check bypass via memory import race
 - Same invariant also broken in “Typos Gone Wild: CVE-2024-6779”

“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- Variant: Double fetch / TOCTOU
 - Case 1' + 4' – b/349529650: Function import signature check race
 - Case 3' – b/352446085: Wasm memory64 bounds check bypass via memory import race
 - Same invariant also broken in “Typos Gone Wild: CVE-2024-6779”
- Bugs from new feature extensions?
 - Case 4 – b/356419168: Arbitrary Wasm stack control via JSPI continuation transplant

Going Forward: Other browsers & future targets

Going Forward: Other browsers

- Firefox?
 - Blatantly wrong subtype validity check for array types: CVE-2024-8385
 - Any array types with different mutability are a subtype of each other??

```
// Checks if two arrays are compatible in a given subtyping relationship.
static bool canBeSubTypeOf(const ArrayType& subType,
                           const ArrayType& superType) {
    // Mutable fields are invariant w.r.t. field types
    if (subType.isMutable_ && superType.isMutable_) {
        return subType.elementType_ == superType.elementType_;
    }

    // Immutable fields are covariant w.r.t. field types
    if (!subType.isMutable_ && !superType.isMutable_) {
        return StorageType::isSubTypeOf(subType.elementType_,
                                         superType.elementType_);
    }

    return true;
}
```

Going Forward: Other browsers

- Firefox?
 - Blatantly wrong subtype validity check for array types: CVE-2024-8385
 - Any array types with different mutability are a subtype of each other??
 - “It’s a remarkably simple exploit.”

```
const m = new WebAssembly.Module(wasmTextToBinary(`(module
  (type $a1 (sub (array anyref)))
  (type $a2 (sub $a1 (array (mut i64))))
  (func $fakeobj (export "fakeobj") (param $v i64) (result anyref)
    (array.new $a2 (local.get $v) (i32.const 1))
    (array.get $a1 (i32.const 0))
  )
)`));
const { fakeobj } = new WebAssembly.Instance(m).exports;
```



Going Forward: Other browsers

- Firefox?
 - Blatantly wrong subtype validity check for array types: CVE-2024-8385
 - Any array types with different mutability are a subtype of each other??
- Safari?
 - WasmGC enabled by default from STP202 – still has a long way to go
 - Many bugs, from obvious type safety violations to JIT compiler bugs: CVE-2024-44221

Going Forward: Future targets

- WebAssembly is rapidly expanding:
 - Exception handling with `exnref`
 - Adds a whole new type hierarchy!
 - JSPI (JS Promise Integration)
 - You can now suspend/resume Wasm functions mid-execution!
 - Memory64
 - Memory/table indices can now be 64bit!
 - ... and many more

Going Forward: Future targets

- WebAssembly is rapidly expanding:
 - Exception handling with `exnref`
 - Adds a whole new type hierarchy!
 - JSPI (JS Promise Integration)
 - You can now suspend/resume Wasm functions mid-execution!
 - Memory64
 - Memory/table indices can now be 64bit!
 - ... and many more
- Chrome is growing too:
 - Transition from Turbofan to Turboshaft backend for Wasm
 - ~~Already transitioning via V8WasmTurboshaft Finch trial (currently at 50%)~~
 - Now fully transitioned on V8 ToT

Conclusions & Takeaways



CODE BLUE 2024

BECAUSE SECURITY MATTERS

112

Conclusions & Takeaways

- A bug is not a one-off problem – it's an indicator of a bigger problem
 - `wasm::ValueType` anti-pattern is finally almost gone (b/366180605)

Conclusions & Takeaways

- A bug is not a one-off problem – it's an indicator of a bigger problem
 - `wasm::ValueType` anti-pattern is finally almost gone (b/366180605)
- Not all bugs are created equal
 - Bugs in Wasm is generally “easier” to write an end-to-end exploit

Conclusions & Takeaways

- A bug is not a one-off problem – it's an indicator of a bigger problem
 - `wasm::ValueType` anti-pattern is finally almost gone (b/366180605)
- Not all bugs are created equal
 - Bugs in Wasm is generally “easier” to write an end-to-end exploit
- Everything (well, most of them) is exploitable if you look closely enough

Conclusions & Takeaways

- A bug is not a one-off problem – it's an indicator of a bigger problem
 - `wasm::ValueType` anti-pattern is finally almost gone (b/366180605)
- Not all bugs are created equal
 - Bugs in Wasm is generally “easier” to write an end-to-end exploit
- Everything (well, most of them) is exploitable if you look closely enough
- V8 Sandbox is still not a meaningful security boundary (yet)
 - Is the (mediocre) difficulty increase worth the (large, recurring) engineering cost?
 - VRP for bypasses increased from \$5k → \$20k, hopefully it will get stronger

Conclusions & Takeaways

- A bug is not a one-off problem – it's an indicator of a bigger problem
 - `wasm::ValueType` anti-pattern is finally almost gone (b/366180605)
- Not all bugs are created equal
 - Bugs in Wasm is generally “easier” to write an end-to-end exploit
- Everything (well, most of them) is exploitable if you look closely enough
- V8 Sandbox is still not a meaningful security boundary (yet)
 - Is the (mediocre) difficulty increase worth the (large, recurring) engineering cost?
 - VRP for bypasses increased from \$5k → \$20k, hopefully it will get stronger
- Automated testing often fails to catch up with new features
 - ... which means there's much more to take a look at!

Conclusions & Takeaways

- A bug is not a one-off problem
 - `wasm::ValueType` anti-pattern
- Not all bugs are created equal
 - Bugs in Wasm is generally harder to exploit
- Everything (well, most of it) is a process
 - Is the (mediocre) difficulty of bypassing V8's security boundaries a recurring engineering cost?
 - VRP for bypasses increases over time
- Automated testing often...
 - ... which means there's a bigger problem

P	S	TYPE
P1	S1	Vulnerability
P2	S2	Process
P1	S1	Vulnerability
P1	S1	Vulnerability
P1	S1	Vulnerability
P2	S2	Process
P1	S1	Vulnerability
P2	S2	Process
P2	S2	Process
P1	S1	Vulnerability
P1	S1	Vulnerability

if a bigger problem
the (b/366180605)
to-end exploit
u look closely enough
dary (yet)
, recurring) engineering cost?
efully it will get stronger
/ features
t!



Thank You!
Questions?

