WebAssembly Is All You Need:

Exploiting Chrome and the V8 Sandbox 10+ times with WASM

Seunghyun Lee (@0x10n)
Carnegie Mellon University

$ whoami

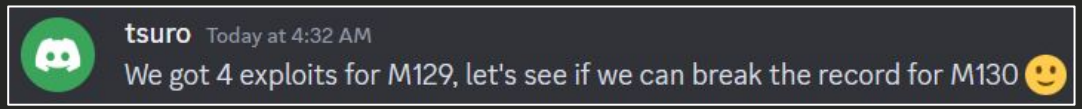
Seunghyun Lee,
a.k.a Xion (@0x10n)
Carnegie Mellon University

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

$ 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?



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

https://www.zerodayinitiative.com/blog/2024/5/2/cve-2024-2887-a-pwn2own-winning-bug-in-google-chrome

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.

$$
\begin{array}{lcl}
\mathit{rectype} & ::= & \mathsf{rec}\ \mathit{subtype}^{*} \\
\mathit{subtype} & ::= & \mathsf{sub}\ \mathsf{final}^{?}\ \mathit{typeidx}^{*}\ \mathit{comptype}
\end{array}
$$

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))))  0
    (type $t2 (struct (field i64 (ref $t1))))  1
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))  2
    (type $u2 (struct (field i64 (ref $u1))))  3
  )
  (type $v (struct (field (ref $t1))))         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++) {
      ...
      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_->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());
  }
}
```

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_->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());
  }
}
```

The Prequel: CVE-2024-2887

- Case 1: Max type count exceeded within a **recursive group**

```
<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. recursive group with 1 type
    builder.startRecGroup();
    builder.addStruct([makeField(wasmRefType(kWasmI32), true)]);
    builder.endRecGroup();

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

0 ~ 999999

~~1000000~~

```
▶ Uncaught CompileError: WebAssembly.Module(): Type definition count exceeds maximum 1000000 @+5000020   wasm-module-builder.js:1999
    at WasmModuleBuilder.toModule (wasm-module-builder.js:1999:12)
    at WasmModuleBuilder.instantiate (wasm-module-builder.js:1990:23)
    at cve-2024-2887.html:17:25
```

The Prequel: CVE-2024-2887

- Case 2: Max type count exceeded with a **standalone type**

```
<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,                     //
```

WasmGC type system

- What are generic heap types?
  - **any**: Top type of all internal non-function type (i.e. supertype of all internal type)
    - “Internal” in WASM perspective
  - **none**: Bottom type of all internal non-function type (i.e. subtype of all internal type)

WasmGC type system

- What are generic heap types?
  - **any**: Top type of all internal non-function type (i.e. supertype of all internal type)
    - “Internal” in WASM perspective
  - **none**: Bottom type of all internal non-function type (i.e. subtype of all internal type)
  - **func**: Top type of all function type
  - **nofunc**: Bottom type of all function type

WasmGC type system

- What are generic heap types?
  - **`any`**: Top type of all internal non-function type (i.e. supertype of all internal type)
    - “Internal” in WASM perspective
  - **`none`**: Bottom type of all internal non-function type (i.e. subtype of all internal type)
  - **`func`**: Top type of all function type
  - **`nofunc`**: Bottom type of all function type
  - **`extern`**: Top type of all external type
    - “External” in WASM perspective, i.e. JS objects
  - **`noextern`**: Bottom type of all external type
  - …

The Prequel: CVE-2024-2887

- Key idea for the exploit:
  - Any concrete struct type is a supertype of `none`
  - 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)))
```

Goal: Type confusion of arbitrary field type `src -> 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)))
```

Goal: Type confusion of arbitrary field type `src -> dst`

2. Push value of type `src`

   `=> Stack: src`

3. Create struct `$tSrc`

   `=> Stack: ref $tSrc`

The Prequel: CVE-2024-2887

=> Stack: src

3. Create struct `$tSrc`

=> Stack: ref $tSrc

```
case kExprStructNew: {
  StructIndexImmediate imm(this, this->pc_ + opcode_length, validate);
  if (!this->Validate(this->pc_ + opcode_length, imm)) return 0;
  PoppedArgVector args = PopArgs(imm.struct_type);
  Value* value = Push(ValueType::Ref(imm.index));
```

The Prequel: CVE-2024-2887

=> Stack: src

3. Create struct $tSrc

=> Stack: ref $tSrc = ref none

```
case kExprStructNew: {
  StructIndexImmediate imm(this, this->pc_ + opcode_length, validate);
  if (!this->Validate(this->pc_ + opcode_length, imm)) return 0;
  PoppedArgVector args = PopArgs(imm.struct_type);
  Value* value = Push(ValueType::Ref(imm.index));
```

```
constexpr ValueType kWasmNullRef = ValueType::RefNull(HeapType::kNone);
```

The Prequel: CVE-2024-2887

=> Stack: ref $tSrc = ref none

4. Type cast to ref $tDst
   a. ref none <: ref $tDst => static upcast, runtime typecheck elided

```
bool null_succeeds = opcode == kExprRefCastNull;
Value* value = Push(ValueType::RefMaybeNull(
    target_type, null_succeeds ? kNullable : kNonNullable));
if (current_code_reachable_and_ok_) {
  // This logic ensures that code generation can assume that functions
  // can only be cast to function types, and data objects to data types.
  if (V8_UNLIKELY(TypeCheckAlwaysSucceeds(obj, target_type))) {
    if (obj.type.is_nullable() && !null_succeeds) {
      CALL_INTERFACE(AssertNotNullTypecheck, obj, value);
    } else {
      CALL_INTERFACE(Forward, obj, value);
    }
```

```
// Checks if {obj} is a subtype of type, thus checking will always
// succeed.
bool TypeCheckAlwaysSucceeds(Value obj, HeapType type) {
  return IsSubtypeOf(obj.type, ValueType::RefNull(type), this->module_);
}
```

The Prequel: CVE-2024-2887

```
=> 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

The Lore: Speedrunning TyphoonPWN with variant analysis

- **May 27**: Boredom exceeded the procrastination threshold
- **May 30**: TyphoonPWN 2024*

* Organized by SSD Secure Disclosure.

The Lore: Speedrunning TyphoonPWN with variant analysis

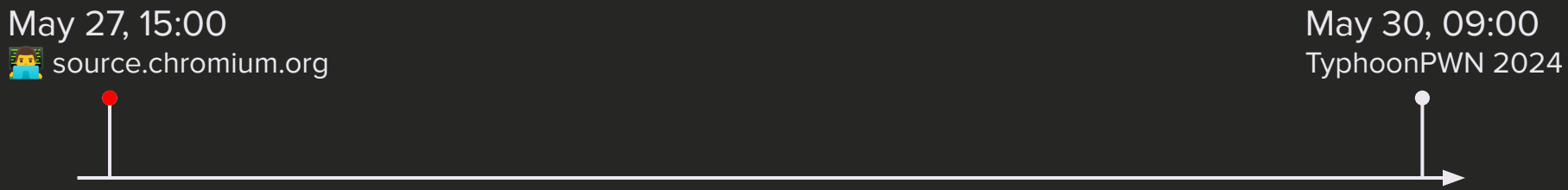
- **May 27**: Boredom exceeded the procrastination threshold
- **May 30**: TyphoonPWN 2024*
- 3-day Chrome renderer exploit speedrun

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

* Organized by SSD Secure Disclosure.

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



The Lore: Speedrunning TyphoonPWN with variant analysis

- How to find and exploit bugs “*easily*”, in the fastest way possible?
  - Not enough time to spend on stabilizing bugs / exploits
    - ⇒ Target bug classes that grant stable, powerful primitives
      - Target code that previously have been exploited with such bug classes

The Lore: Speedrunning TyphoonPWN with variant analysis

- How to find and exploit bugs “*easily*”, in the fastest way possible?
  - Not enough time to spend on stabilizing bugs / exploits
    - ⇒ Target bug classes that grant stable, powerful primitives
      - Target code that previously have been exploited with such bug classes
  - Not enough time to learn intricate subsystems / implementations
    - ⇒ Target large, complex but legible code
      - Large, complex: Difficult to write & reason about for devs
      - Legible: Simple enough for me to quickly understand the code base
    - ⇒ Target code that can be easily tested & have my understanding of the code verified

The Lore: Speedrunning TyphoonPWN with variant analysis

- How to find and exploit bugs “*easily*”, in the fastest way possible?
  - Not enough time to spend on stabilizing bugs / exploits
    - ⇒ Target bug classes that grant stable, powerful primitives
      - Target code that previously have been exploited with such bug classes
  - Not enough time to learn intricate subsystems / implementations
    - ⇒ Target large, complex but legible code
      - Large, complex: Difficult to write & reason about for devs
      - Legible: Simple enough for me to quickly understand the code base
    - ⇒ 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
    - E.g. What's the result of searching "wasm isorecursive type canonicalization"?
      - V8 commits
      - Wasm spec discussions
      - Many PL theory papers

The Lore: Speedrunning TyphoonPWN with variant analysis

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

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

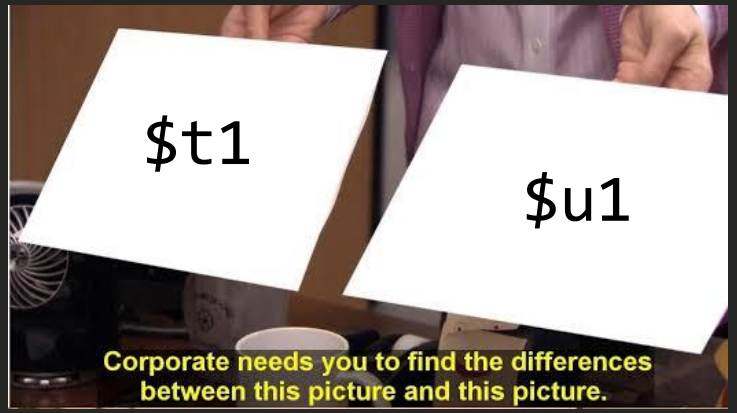
Isorecursive Type Systems

- 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))))  0
    (type $t2 (struct (field i64 (ref $t1))))  1
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))  2
    (type $u2 (struct (field i64 (ref $u1))))  3
  )
  (type $v (struct (field (ref $t1))))         4
)
```



Isorecursive Type Systems: Type Equivalence

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

```
(module
  (rec
    (type $t1 (struct (field i32 (ref $t2))))  0
    (type $t2 (struct (field i64 (ref $t1))))  1
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))  2
    (type $u2 (struct (field i64 (ref $u1))))  3
  )
  (type $v (struct (field (ref $t1))))         4
)
```

- 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.

https://github.com/WebAssembly/gc/blob/main/proposals/gc/MVP.md

Isorecursive Type Systems: Type Equivalence

- In plain language:
  - Represent recursive type group as type tuple `rec`
  - Replace all recursive type variables into `rec.<i>`
  - Compare this replaced type to check type equivalence
- In PL terms:

```
tie($t1) ~ (mu a. <(struct (field i32 (ref a.1))), (struct i64 (field (ref a.0)))>).0
tie($t2) ~ (mu a. <(struct (field i32 (ref a.1))), (struct i64 (field (ref a.0)))>).1
```

  - Recursive type variable `a` represents `rec`

Isorecursive Type Systems: Type Equivalence

- WASM uses iso-recursive typing rules which compares the `tie()`'d state

```
(module
  (rec
    (type $t1 (struct (field i32 (ref $t2))))  0
    (type $t2 (struct (field i32 (ref $t1))))  1
  )
  (rec
    (type $u1 (struct (field i32 (ref $u1))))  2
  )
  (type $v (struct (field (ref $t1))))         3
)
```

- None of the `tie()`'d type representation below are equivalent

```
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))))  0
    (type $t2 (struct (field i64 (ref $t1))))  1
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))  2
    (type $u2 (struct (field i64 (ref $u1))))  3
  )
  (type $v (struct (field (ref $t1))))         4
)
```

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))))  0/0
    (type $t2 (struct (field i64 (ref $t1))))  1/1
  )
  (rec
    (type $u1 (struct (field i32 (ref $u2))))  2/0
    (type $u2 (struct (field i64 (ref $u1))))  3/1
  )
  (type $v (struct (field (ref $t1))))         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: 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: Subtyping

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

Isorecursive Type Systems: Subtyping

- Canonical subtype check:
  - Canonicalize, then `sub = canonical_supertypes_[sub]` until match or 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);
}
```

Isorecursive Type Systems: Subtyping

- Canonical subtype check:
  - Canonicalize, then `sub = canonical_supertypes_[sub]` until match or end
  - Used for subtype check between module-defined reference types:

```
  DCHECK(super_heap.is_index());
  uint32_t super_index = super_heap.ref_index();
  DCHECK(super_module->has_type(super_index));
  // The {IsSubtypeOf} entry point already has a fast path checking ValueType
  // equality; here we catch (ref $x) being a subtype of (ref null $x).
  if (sub_module == super_module && sub_index == super_index) return true;
  return GetTypeCanonicalizer()->IsCanonicalSubtype(sub_index, super_index,
                                                    sub_module, super_module);
}
```

“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

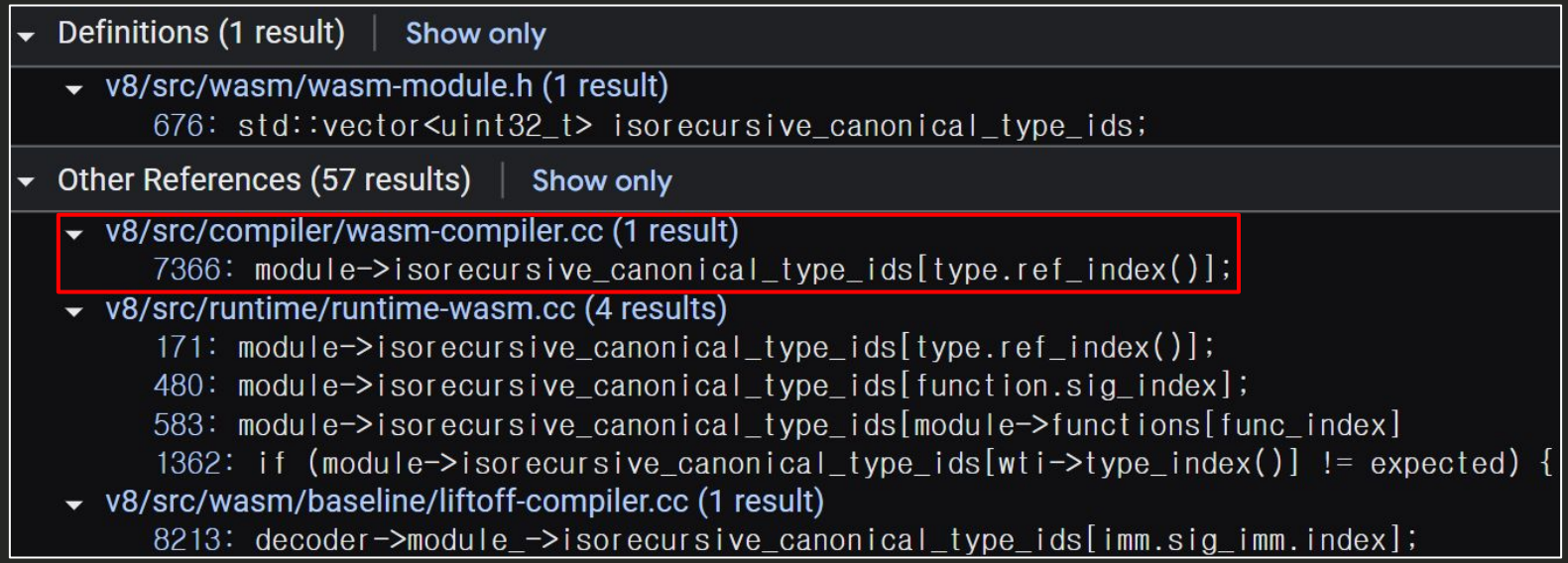
- Enough with the background - let’s find the bug
- Idea 1: `uint32_t` canonical index overflow
  - Effect: Overlapping canonical index, universal WASM type confusion
  - In reality: Requires ~200GB memory at minimum due to overheads

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

- Enough with the background - let’s find the bug
- Idea 1: `uint32_t` canonical index overflow
  - Effect: Overlapping canonical index, universal WASM type confusion
  - In reality: Requires ~200GB memory at minimum due to overheads
- 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



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

- Object typechecks 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 `ValueType`?

```
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>;
```

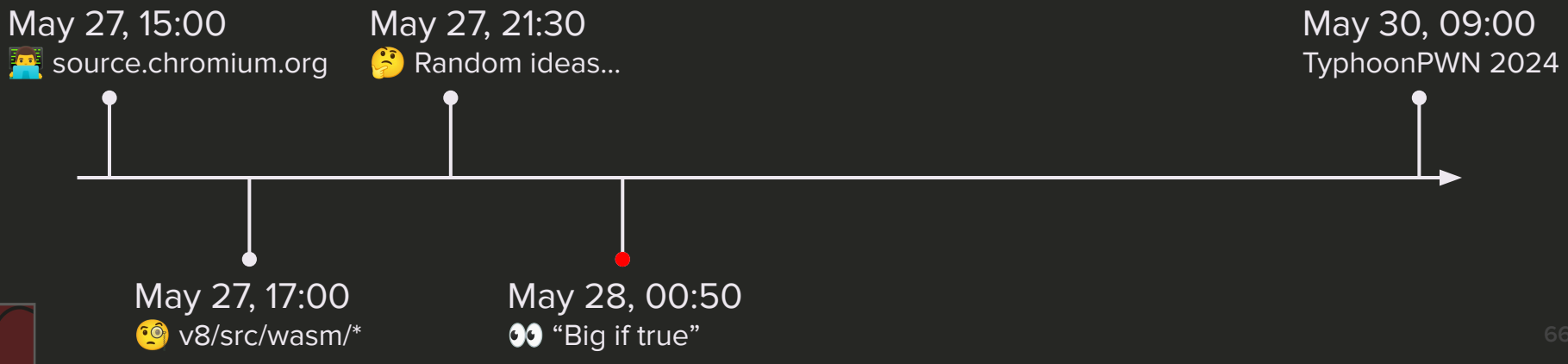
“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- `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 ~ 999,999 (= kV8MaxWasmTypes - 1)`
    - Generic heap types: `1,000,000 ~ 1,000,0xx`
    - Internal types (invalid): `1,000,0xx + 1 (kBottom)`

“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 ~ 999,999 (= kV8MaxWasmTypes - 1)`
    - Generic heap types: `1,000,000 ~ 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<<20) + k (0 <= 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

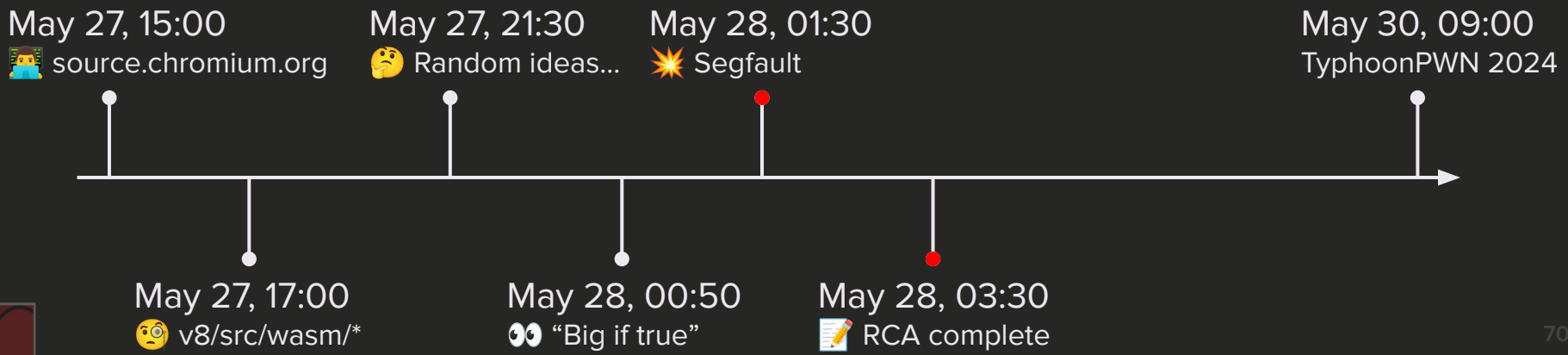
- What if `t = (n<<20) + k` (`1E6 <= k < 2^20`), i.e. a generic type index?

```
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

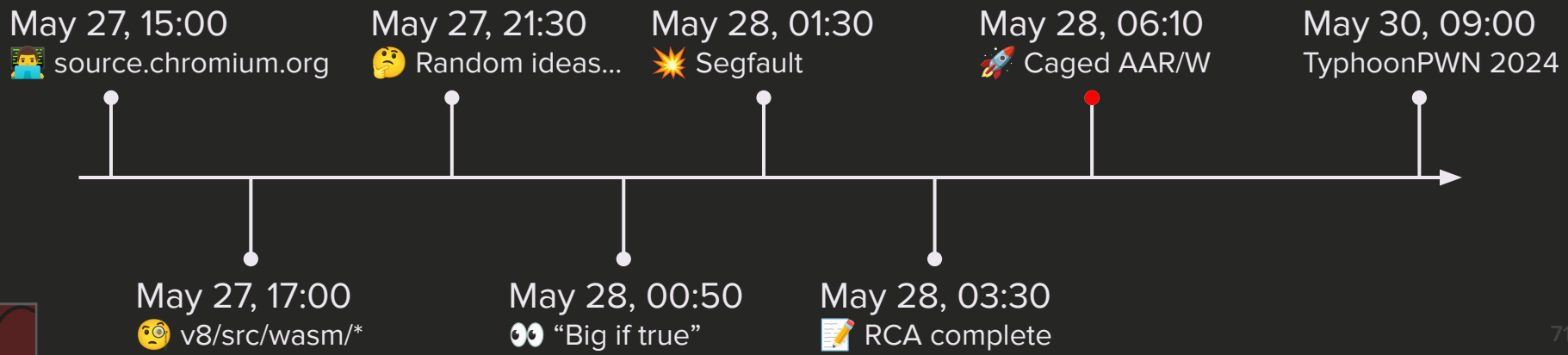
- Bug #2: Canonical type index confused as generic `HeapType`!
  - As generic `HeapTypes` use the same `ValueType`, this is indistinguishable from the very moment we use `ValueType` to store canonical type indices
- Effect: Broken typecheck on JS-to-Wasm boundary, where:
  - Intended: Typecheck against `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



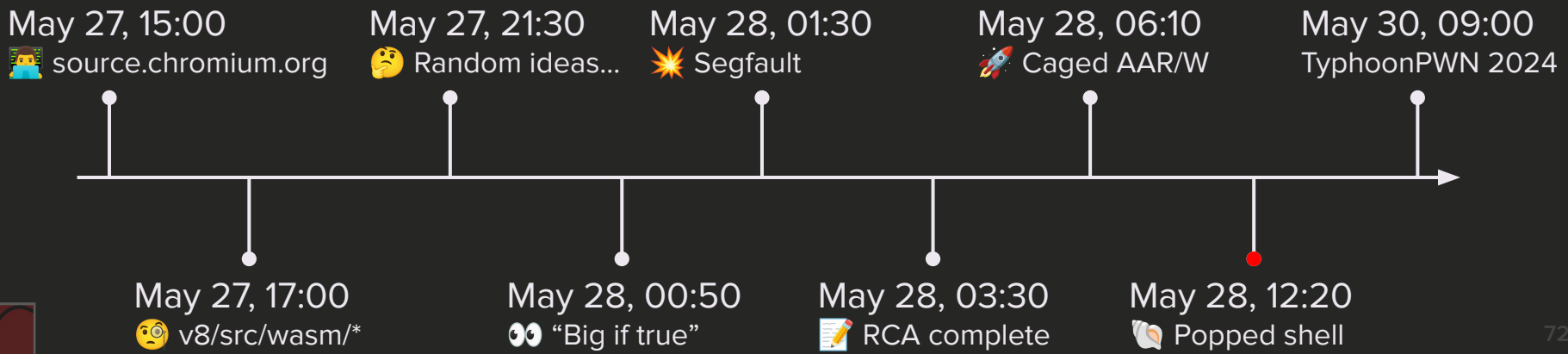
“Deja Vu”: CVE-2024-6100 @ TyphoonPWN 2024

- 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™
  - Common misconception that V8 sandbox has no raw pointers – not with PA!



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

- Fun fact: Fuzzers hit this bug repeatedly (as a DCHECK)
  - But none of the reporters nor devs were able to repro it (b/323856491)
  - The assumption is wrong – Wasm module creation is NOT side-effect free!

cl...@google.com <cl...@google.com> #4 Feb 6, 2024 05:38

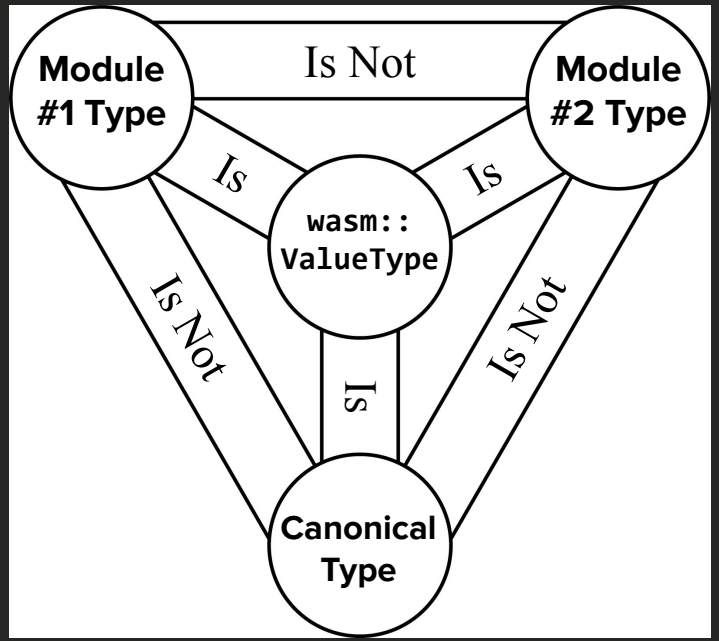
Hm, after decoding a recursion group we try to add it to the type canonicalizer, and there we encounter an invalid type. This shouldn't happen. Without a reproducer it's difficult to figure out what goes wrong where. Adding Manos as the author of type canonicalization.

al...@goodmanemail.com <al...@goodmanemail.com> #5 Feb 6, 2024 05:48

I've got hundreds of crash files where the fuzzer crashed with this dcheck, however none of them reproduce the crash for me. Makes me think the fuzzer memory is getting corrupted somehow? I can reproduce the issue quite reliably by letting the fuzzer run for some time. Perhaps I could collect more information somehow?

The `wasm::ValueType` Trinity

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



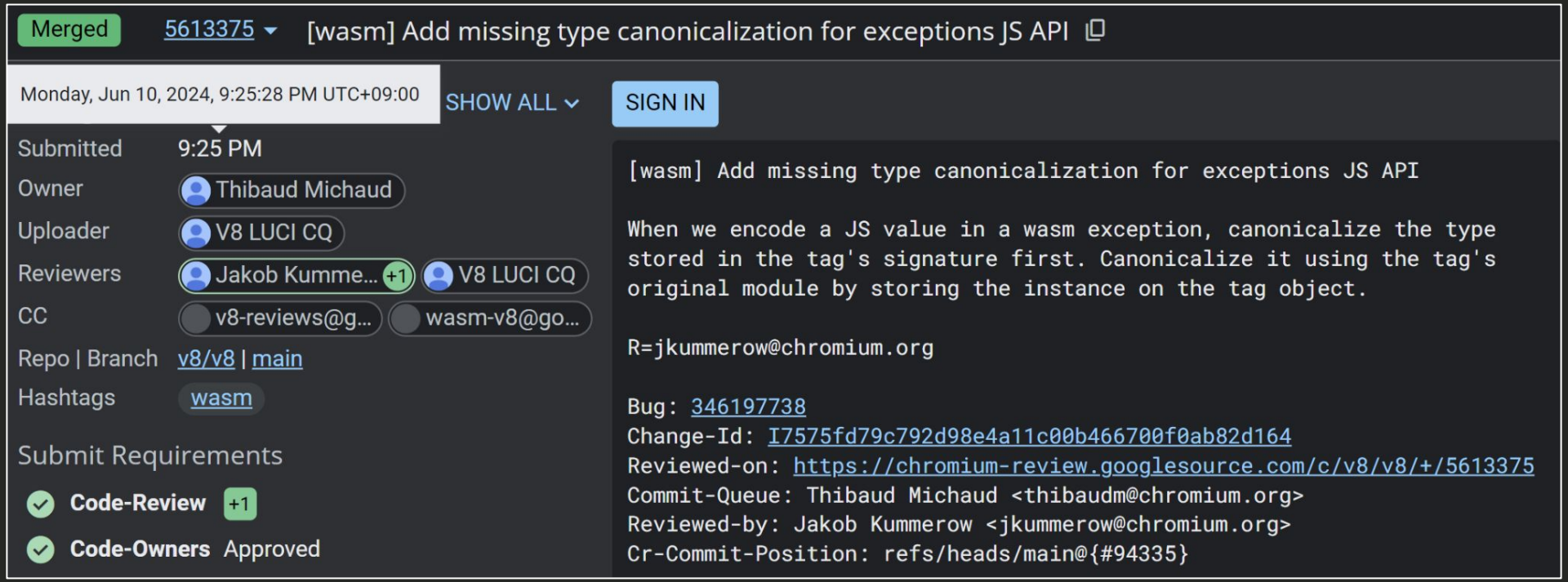
The Sequel: CVE-2024-9859

(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



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
    - Typechecked with module-specific index ➔ canonical index confusion

The Sequel: CVE-2024-9859

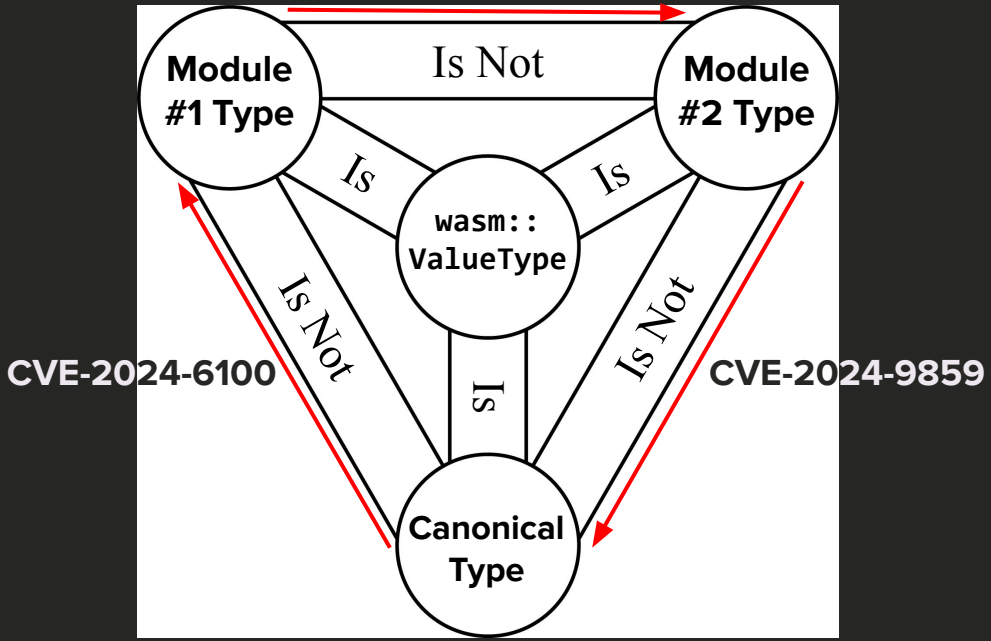
1. Wasm module exports exception signature (i.e. Tag) with module-specific types
2. An exception is created with `WebAssembly.Exception()` with the export tag
    - Typechecked with module-specific index ➡ canonical index confusion
3. Catch the exception within Wasm to unpack values as module-specific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

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

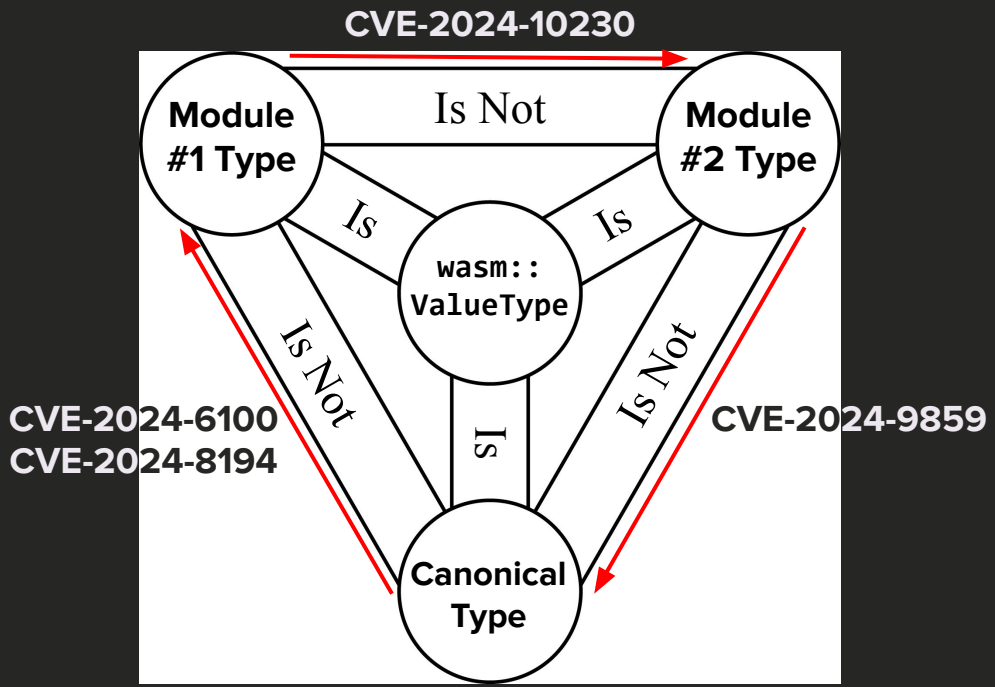
The `wasm::ValueType` Trinity

- 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

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

- 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** (`== static_cast<uint8_t>(0x100)`)
     - Accessed using cached memory base & length of memory index 0x100
- But if offset check is all done purely dynamically, this won't be a problem…?

Typos Gone Wild: CVE-2024-6779

- Optimization – if offset & index is known & statically in-bounds, elide check

```
uintptr_t end_offset = offset + access_size - 1u;

if (constant_index.HasResolvedValue() &&
    end_offset <= memory->min_memory_size &&
    constant_index.ResolvedValue() < memory->min_memory_size - end_offset) {
  // The input index is a constant and everything is statically within
  // bounds of the smallest possible memory.
  return {converted_index, BoundsCheckResult::kInBounds};
}
```

Typos Gone Wild: CVE-2024-6779

- Optimization #2 – if `offset <= min size`, elide mem size comparison
  - Remaining size `effective_size` subtraction overflow!

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

Node* end_offset_node = mcgraph_->UintPtrConstant(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);

// 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

    => With memory64, this is **uint64 – fully arbitrary R/W**, but the feature is staged...
  - 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

    => With memory64, this is **uint64 – fully arbitrary R/W**, but the feature is staged…
  - 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

    => With memory64, this is **uint64 – fully arbitrary R/W**, but the feature is staged…
  - Wasm memory padded to 8GB w/ guard page for OOB trapping mechanism

    => **On Android, no guard page** due to signal safety issues

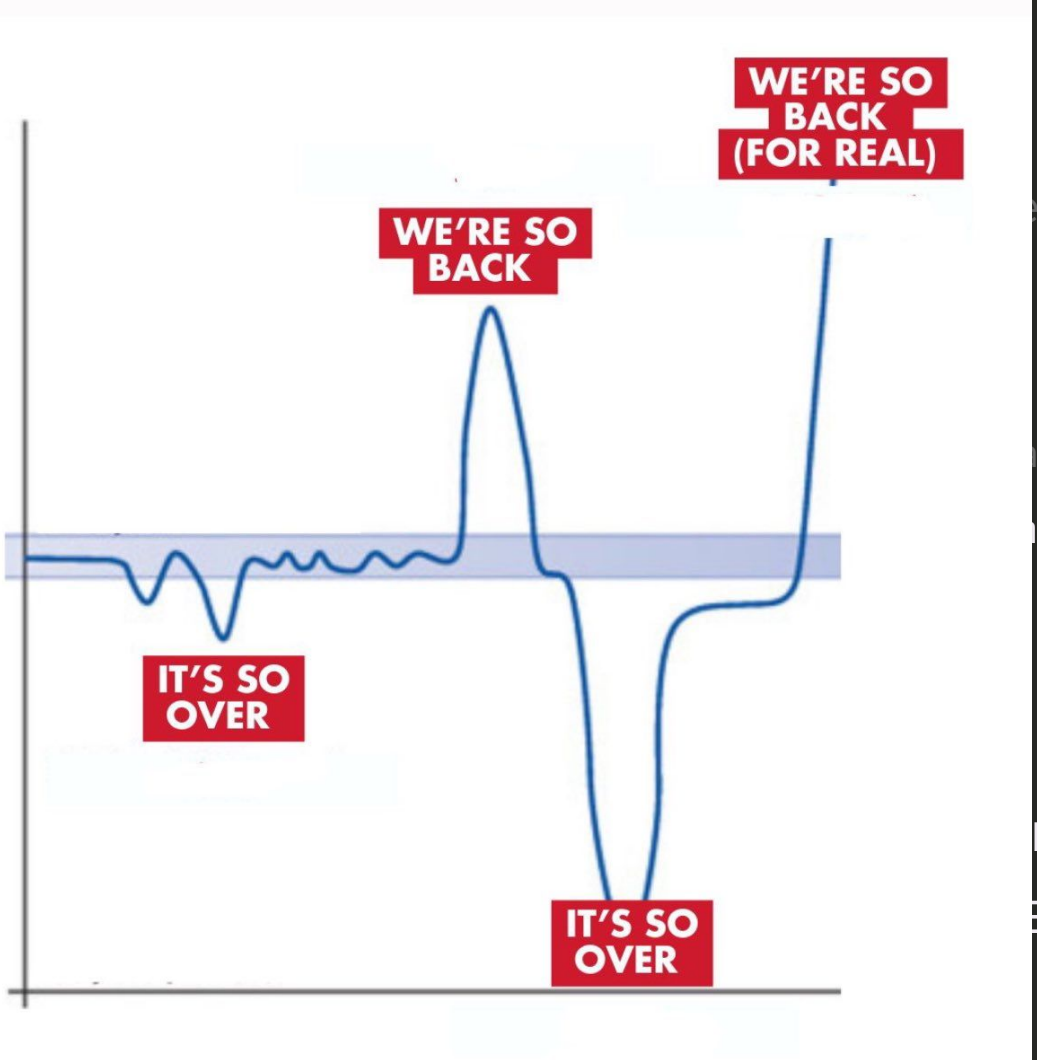
    => But there's nothing useful to overwrite?

    It's allocated after ArrayBuffer PartitionAlloc…

    => In "some cases", it's between V8 cage & ArrayBuffer PartitionAlloc!!
- ~~Exploitable?~~ ~~Unexploitable??~~ ~~Exploitable???~~ ~~Unexploitable????~~ Exploitable!

Typos Gone

- Great, arbit :)
- Not-so-grea
  - Index li
    => With ture is staged...
  - Wasm m echanism
    => **On A**
    => But t
    It's a
    => In "s loc!!
- ~~Exploitable?~~ 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)

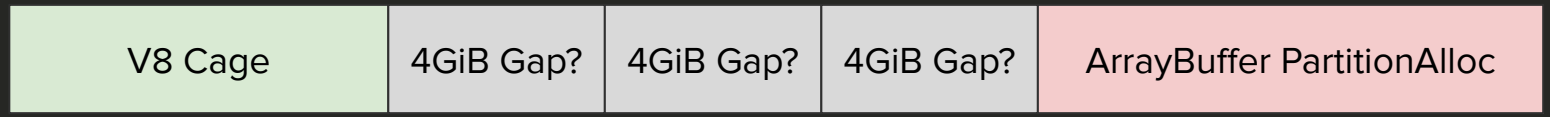
Man Yue Mo, “Controlled chaos: Predicting object addresses in Chrome (without breaking a sweat),” in POC2022.
* Fixed in https://crrev.com/c/5806587

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**

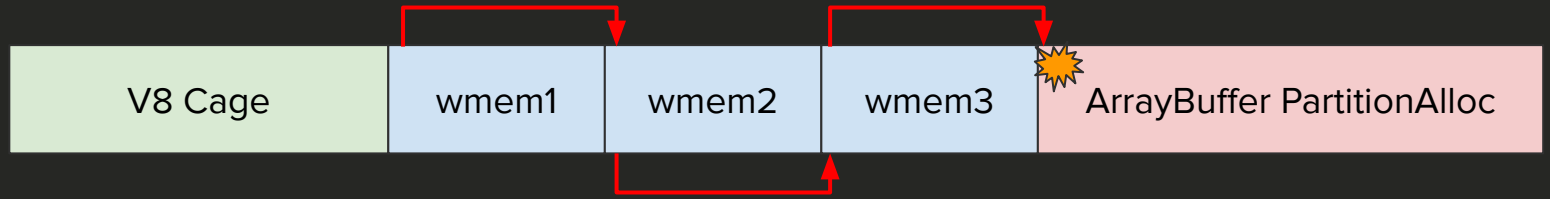
Typos Gone Wild: CVE-2024-6779

- 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



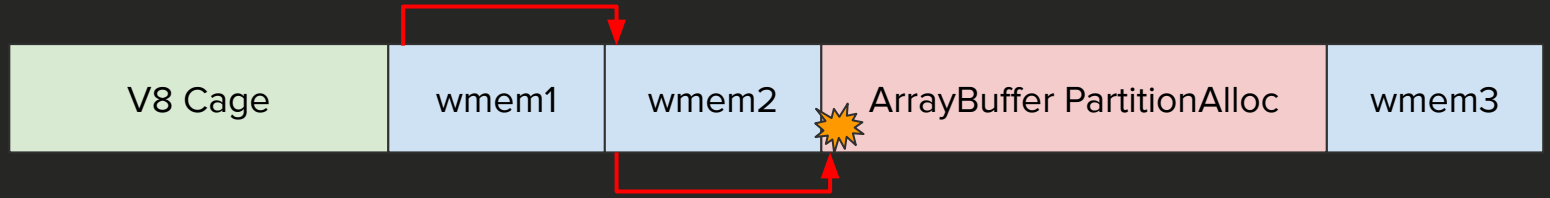
Typos Gone Wild: CVE-2024-6779

- 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



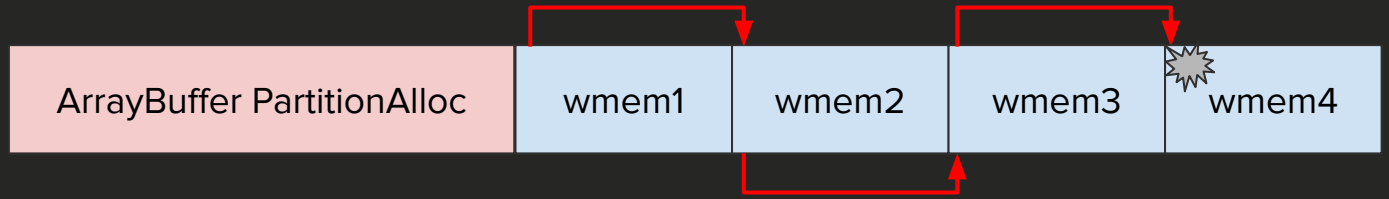
Typos Gone Wild: CVE-2024-6779

- 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



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



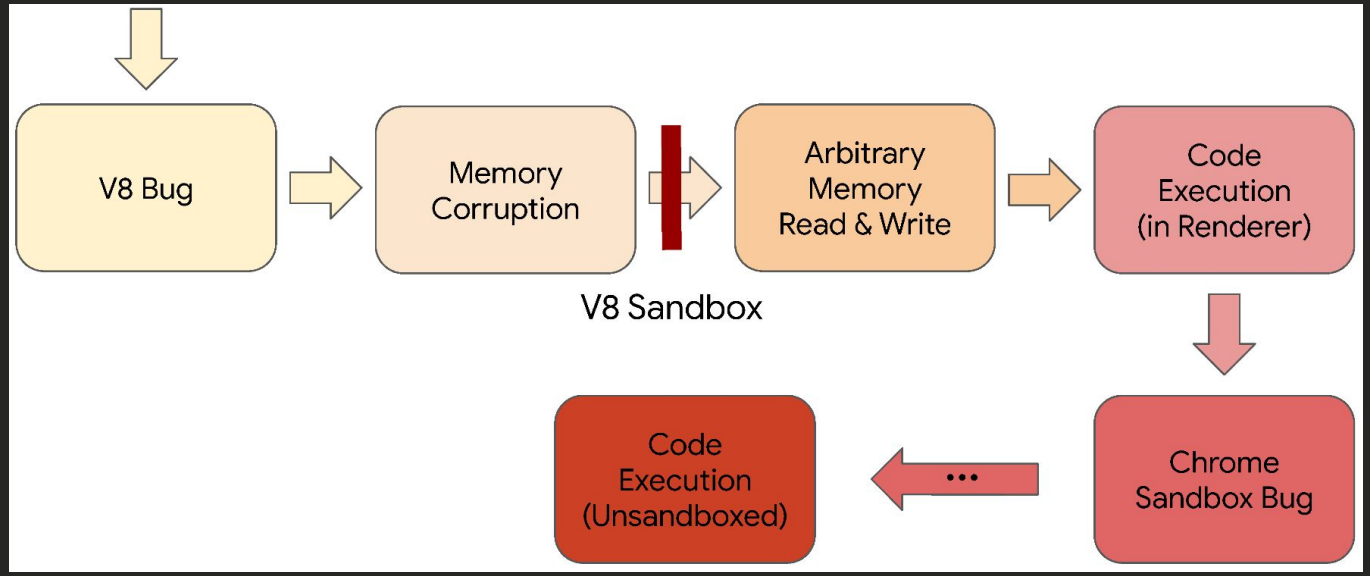
“All-You-Can-Eat”
Wasm-based V8 Sandbox bypasses

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

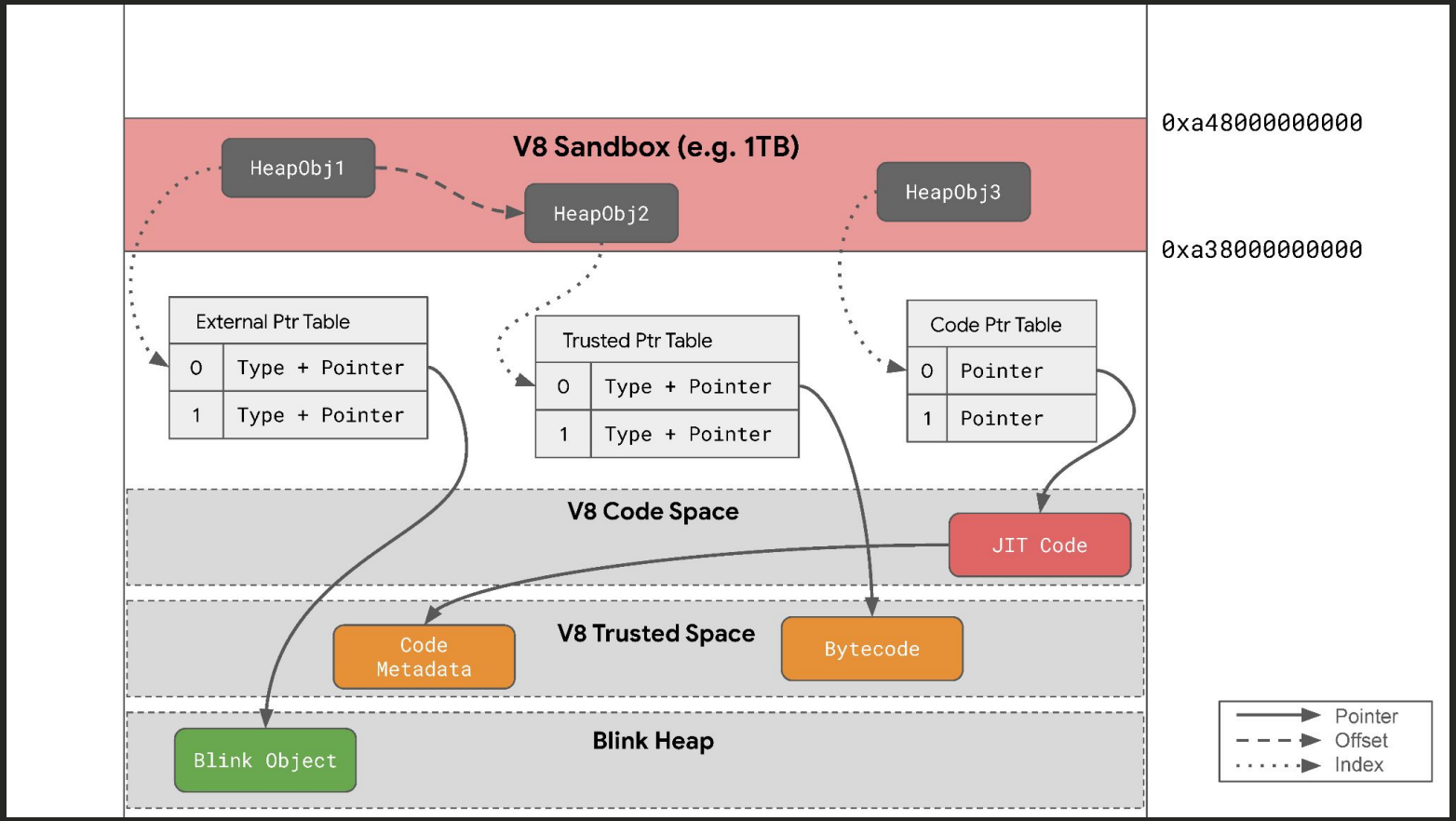
Crash Course on V8 Sandbox

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



Samuel Groß, “The V8 Heap Sandbox,” in OffensiveCon 2024.

Crash Course on V8 Sandbox



Samuel Groß, “The V8 Heap Sandbox,” in OffensiveCon 2024.

“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

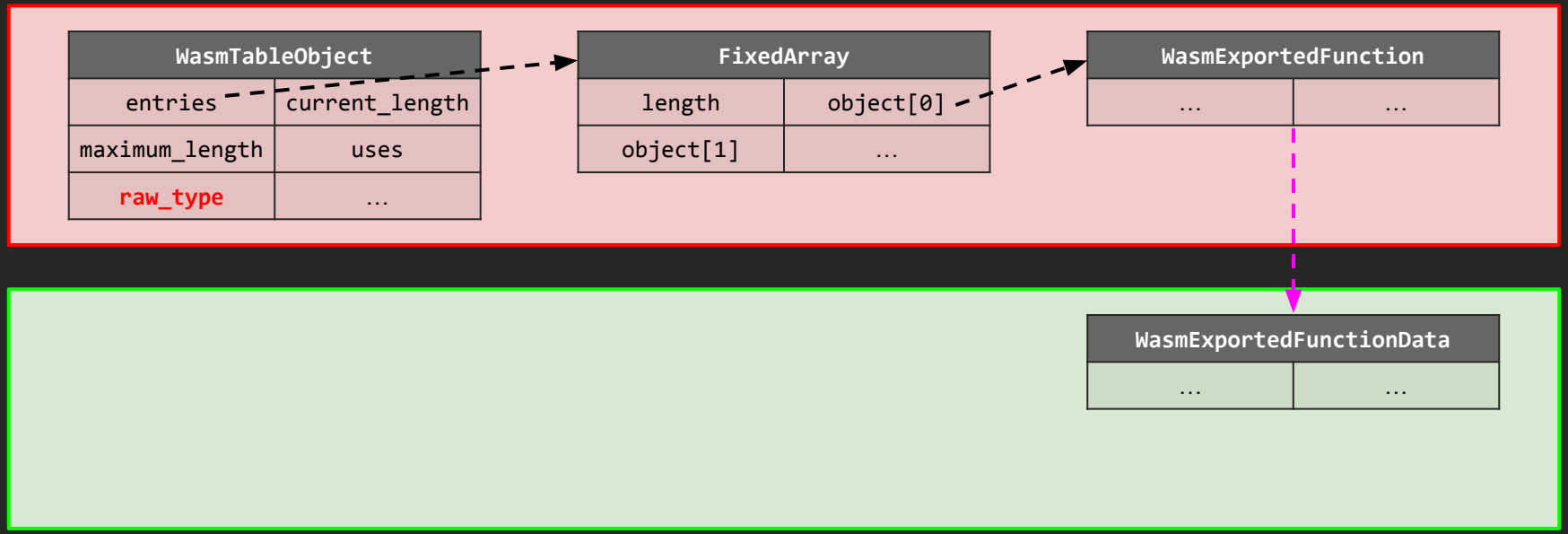
- Wasm is a goldmine of V8 Sandbox bypasses
  - What makes it so vulnerable?
  - What are the common patterns?
- Key idea:
  - Reference types are represented as full 64bit pointers at:
    - Within a Wasm function
    - Across Wasm function calls – **function signature confusion leads to v8sbx bypass**!
  - 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

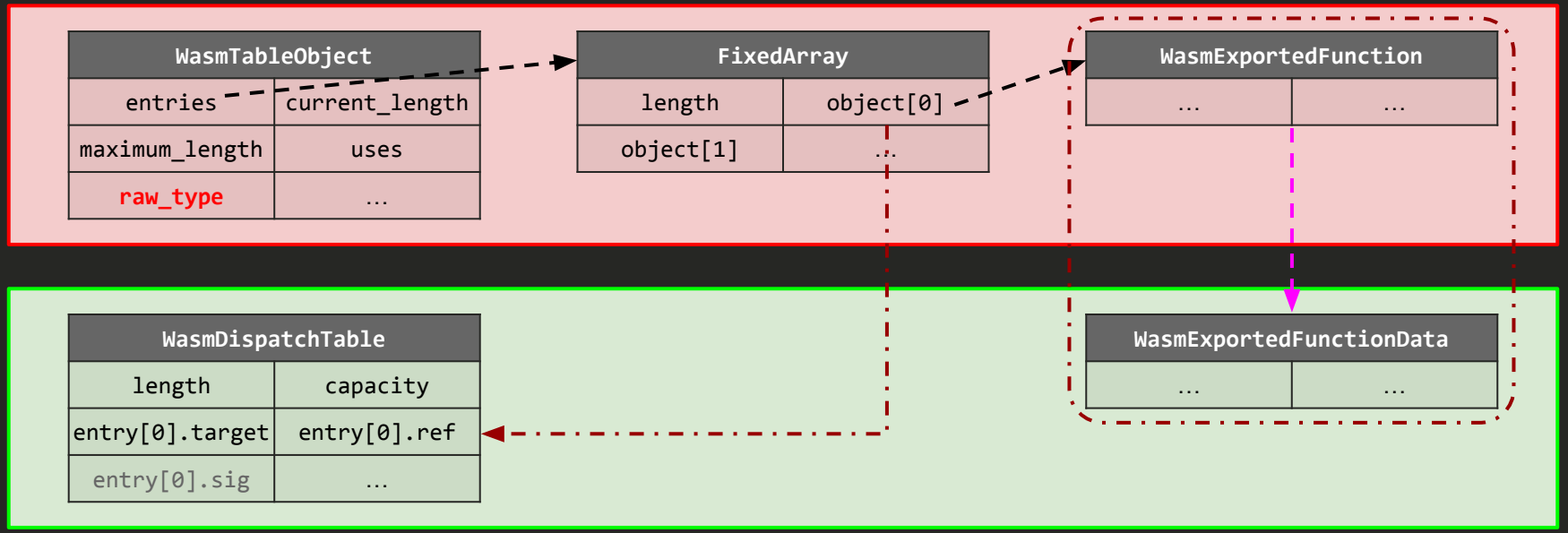
- Case 1: Code metadata (i.e. signatures) corruption
  - b/348793147: Missing signature check when importing function tables



Classification referenced from Samuel Groß’s “The V8 Heap Sandbox” talk. Some fields are implicitly omitted in the diagram.

“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

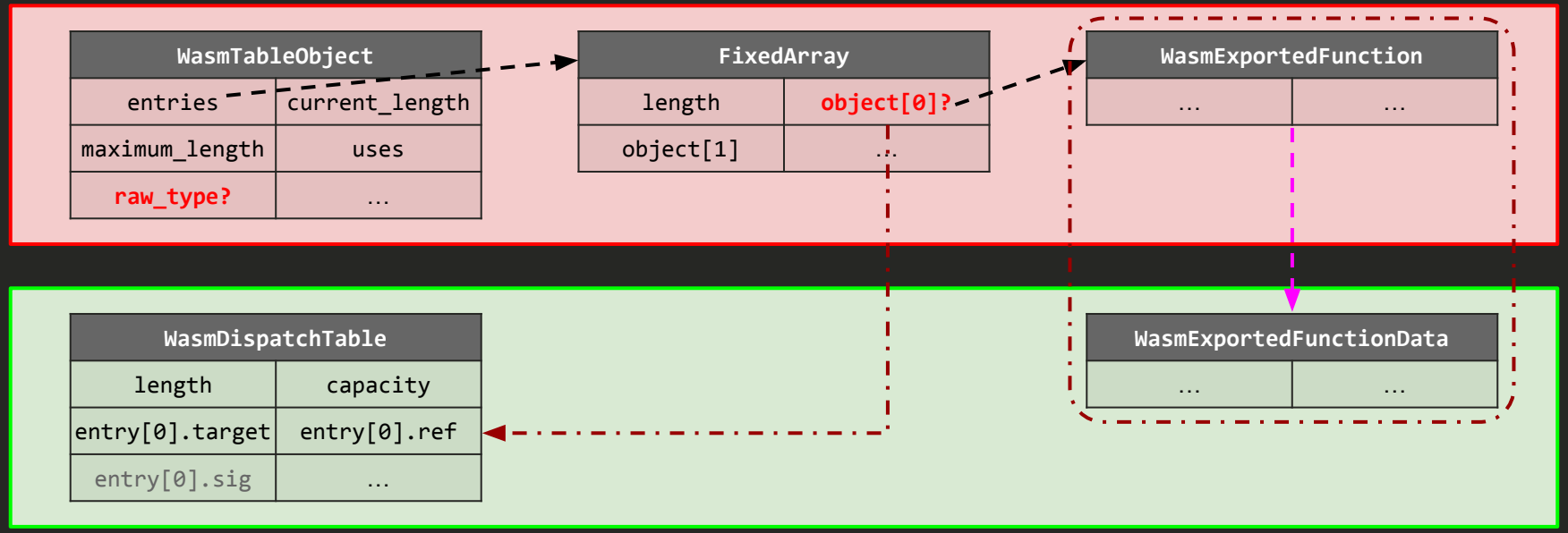


Invariant: `entry[i].sig <: table type`

Classification referenced from Samuel Groß’s “The V8 Heap Sandbox” talk. Some fields are implicitly omitted in the diagram.

“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`~~

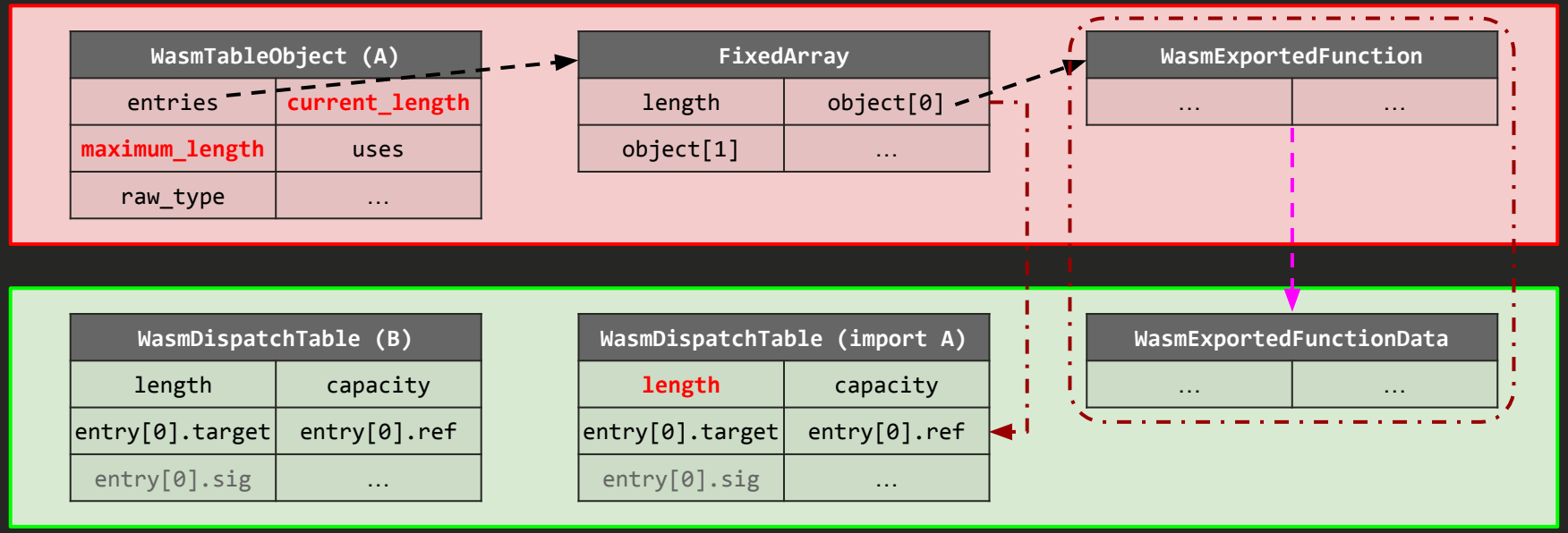
Classification referenced from Samuel Groß’s “The V8 Heap Sandbox” talk. Some fields are implicitly omitted in the diagram.

“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



Edouard Bochin, Tao Yan, Bo Qu, “Let the Cache Cache and Let the WebAssembly Assemble: Knocking’ on Chrome’s Shell,” in Black Hat USA 2024.

“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);
}
```


Edouard Bochin, Tao Yan, Bo Qu, “Let the Cache Cache and Let the WebAssembly Assemble: Knocking’ on Chrome’s Shell,” in Black Hat USA 2024.

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

- Case 2: Untrusted indices
  - [wasm] Avoid OOB writes of the WasmDispatchTable

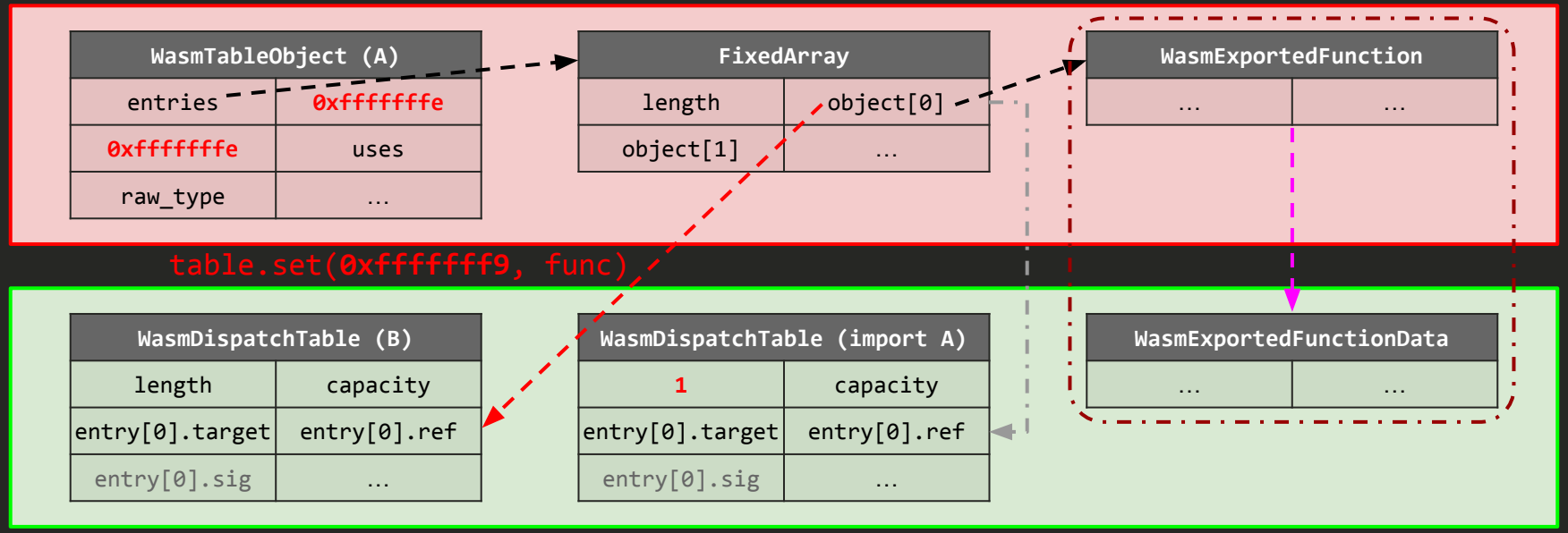
```
void WasmDispatchTable::Set(int index, Tagged<Object> ref, Address call_target,
                            int sig_id) {
  if (ref == Smi::zero()) {
    DCHECK_EQ(kNullAddress, call_target);
    Clear(index);
    return;
  }

  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);
}
```

```
// The current length of this dispatch table.
inline int length() const;
```


Edouard Bochin, Tao Yan, Bo Qu, “Let the Cache Cache and Let the WebAssembly Assemble: Knocking’ on Chrome’s Shell,” in Black Hat USA 2024.

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

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



Edouard Bochin, Tao Yan, Bo Qu, “Let the Cache Cache and Let the WebAssembly Assemble: Knocking’ on Chrome’s Shell,” in Black Hat USA 2024.

“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`

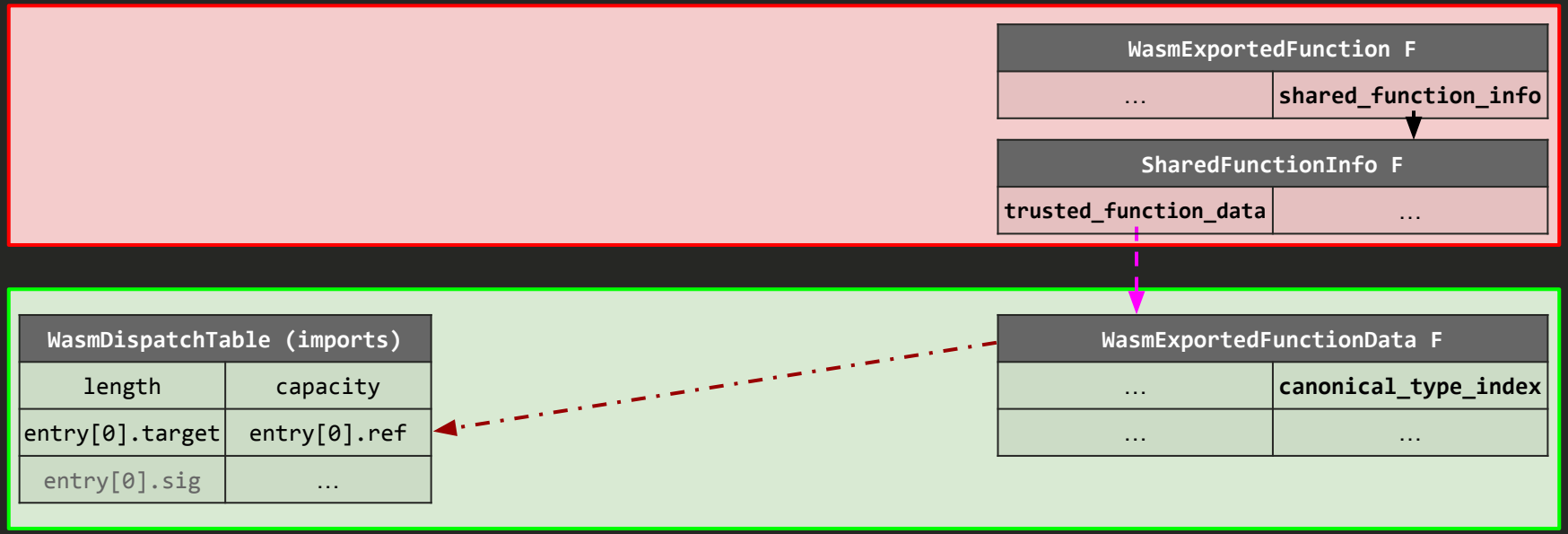
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- 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

* Bochin et al. refers to the transplantation technique as “Field Confusion”.

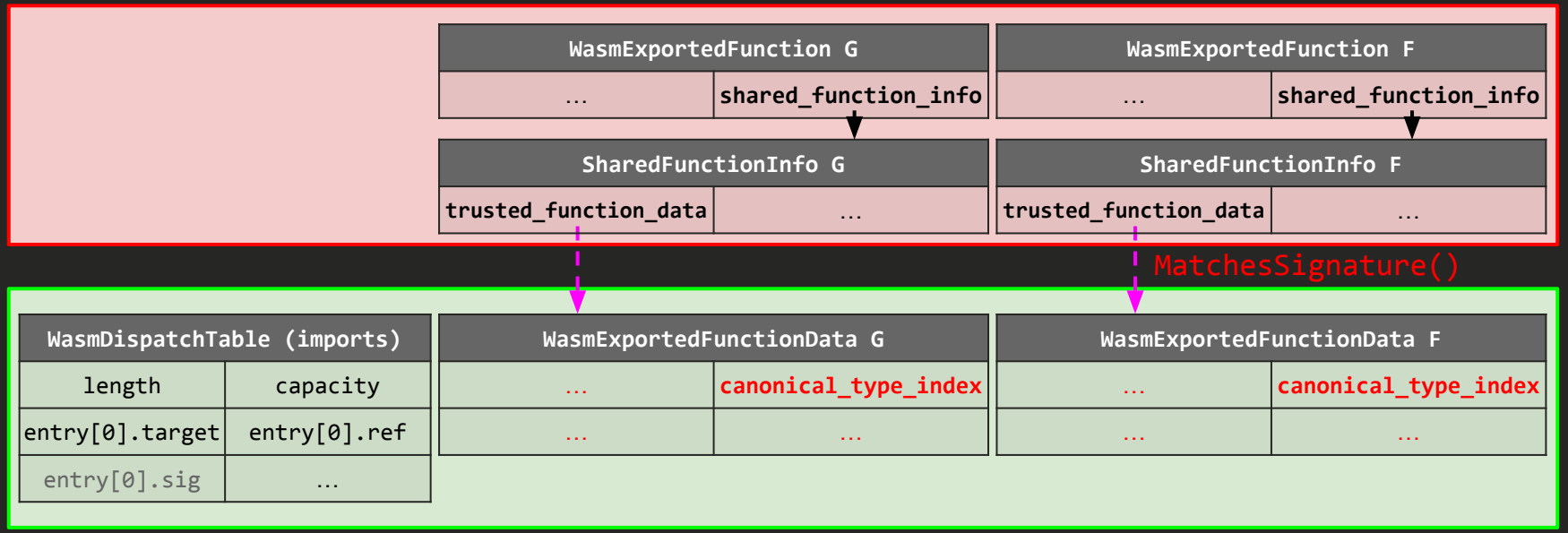
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

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



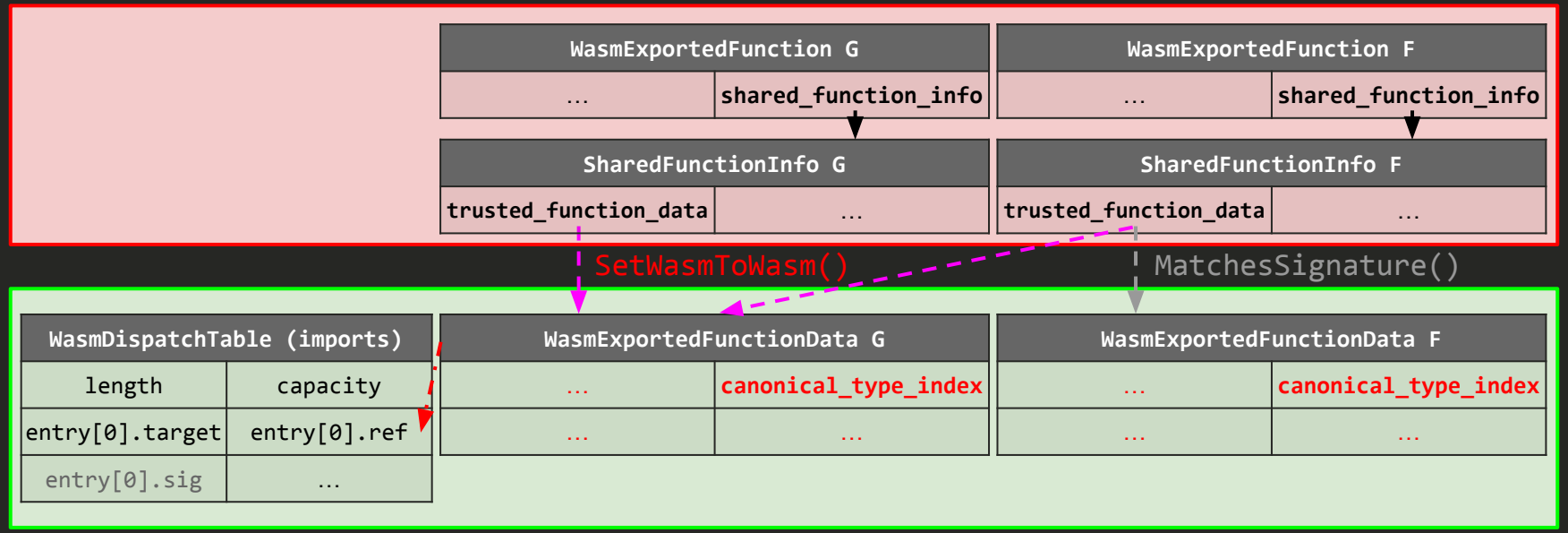
“All-You-Can-Eat” Wasm-based V8 Sandbox bypasses

- 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 import race
    - Also seen 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 import race
    - Also seen 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??
- 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
    - Already transitioning via `V8WasmTurboshaft` Finch trial (currently at 50%)

Conclusions & Takeaways

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
    - "Easy": Stable, deterministic, straightforward, ...
  - Once you have your exploit framework, just "plug-and-play" new bugs

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
    - "Easy": Stable, deterministic, straightforward, ...
  - Once you have your exploit framework, just "plug-and-play" new bugs
- 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
    - "Easy": Stable, deterministic, straightforward, ...
  - Once you have your exploit framework, just "plug-and-play" new bugs
- 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?

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
    - "Easy": Stable, deterministic, straightforward, ...
  - Once you have your exploit framework, just "plug-and-play" new bugs
- 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?
- Automated testing often fails to catch up with new features
  - ... which means there's much more to take a look at!

Conclusions & Takeaw

- A bug is not a one-off pr… …f a bigger problem
  - `wasm::ValueType` ant… …e (b/366180605)
- Not all bugs are created …
  - Bugs in Wasm is gener… …to-end exploit
    - "Easy": Stable, deter…
  - Once you have your ex… …and-play" new bugs
- Everything (well, most of … …u look closely enough
- V8 Sandbox is still not a … …dary (yet)
  - Is the (mediocre) difficu… …, recurring) engineering cost?
- Automated testing often … …y features
  - … which means there's … …t!

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

Thank You!
Questions?