A Chrome/Edge RCE via V8 WASM Type Confusion Pwn2Own Vancouver 2024

Manfred Paul (@_manfp)

March 20, 2024

Contents

1 Introduction

In this whitepaper I describe an exploit in the V8 JavaScript and WebAssembly engine, which allows for the execution of arbitrary shellcode inside the ”renderer” process. This includes a bypass of the V8 memory sandbox (”Ubercage”) [1], though code execution is still constrained by the process isolation-based browser sandbox (unless the browser is run with the `--no-sandbox` flag).

2 WebAssembly Type confusion

2.1 Root Cause

WebAssembly modules may contain a `type` section that defines a list of custom ”heap types”. In the base specification, this is only used to declare function types, but with the adoption of the garbage collection (GC) proposal [3], this section can additionally define *struct types*, allowing for the use of composite, heap-allocated types in WebAssembly.

Normally, structs declared in this section may only contain references to preceding structs (i.e., ones with a lower type index). To use mutually recursive data structures, a feature called

recursive type groups can be used: Then, the (potentially) mutually recursive types are not declared as individual entries in the type section, but as one recursive group containing the individual types, which are allowed to reference each other.

With this in mind, consider the function responsible for parsing the type section from the binary WebAssembly format in v8/src/wasm/module-decoder-impl.h:

```
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());
        }
      }
    }
    ...
}
```

At (1), the limit `kV8MaxWasmTypes` (currently equal to one million) is passed as a maximum to `consume_count()`, ensuring that at most this many entries are read in from the type section.

When recursive type groups were added [1] this check became quite obviously insufficient: While only `kV8MaxWasmTypes` entries of the type section can be read in, each of those can potentially be a recursive type group containing more than one individual type definition.

This insufficiency was clearly noticed at the time of this change, as together with recursive type groups a second check was added at (2). Here, for each recursive type group it is checked that the addition of the constituent types would not exceed the `kV8MaxWasmTypes` limit.

However, this second check is still not enough. While it protects the indices of each type allocated *inside* a recursive group, the presence of those groups also has implications for types declared *outside* this group, as each recursive group adds to the total count of declared types.

To make this clearer, imagine a type section consisting of two entries: One recursive group with `kV8MaxWasmTypes` many entries itself, and following that group one non-recursive type. The check at (1) is passed, as the section only has two entries. While processing the recursive group, the check at (2) is also passed, as the section has exactly `kV8MaxWasmTypes` entries. For the following single type, there is no further check: at (3) the type is simply allocated at the next free index - in this case at `kV8MaxWasmTypes` (exceeding the usual maximum of `kV8MaxWasmTypes-1`). If there was more than one non-recursive type at the end of the type section, they would similarly get assigned `kV8MaxWasmTypes+1`, `kV8MaxWasmTypes+2`, etc. as type indices.

2.2 Impact

Exceeding the maximal number of declared heap types [2] might seem like a very harmless resource exhaustion bug at first. But due to some internal details about how V8 handles WebAssembly heap types, it actually directly allows constructing some very powerful exploit primitives.

In v8/src/wasm/value-type.h the encoding of heap types is defined:

```
// 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,
    kEq,
    kI31,
    kStruct,
    kArray,
    kAny,
    kExtern,
    ...
    kNone,
```

[1] In Change-ID I69fd04ecc5611f6230c95d5c89d1c520163fffae as far as I can tell

[2] And that only up to a factor of 2 - the last recursive group has to end before index `kV8MaxWasmTypes` and can only be followed by `kV8MaxWasmTypes-1` individual types at most

```
    ...
};
```

Here, V8 actively relies on the encoding of all user-defined heap types with indices smaller than `kV8MaxWasmTypes` - larger indices are reserved for fixed, internal heap types. This results in our own type declarations actually aliasing one of these internal types, leading to many opportunities for type confusion.

2.3 Universal WebAssembly Type Confusion

To leverage this encoding ambiguity into a full type confusion, let's first consider the `struct.new` opcode, which creates a new (reference to a) struct from fields given on the stack. The relevant check on the type index can be found in v8/src/wasm/function-body-decoder-impl.h:

```
bool Validate(const uint8_t* pc, StructIndexImmediate& imm) {
    if (!VALIDATE(module_->has_struct(imm.index))) {
        DecodeError(pc, "invalid struct index: %u", imm.index);
        return false;
    }
    imm.struct_type = module_->struct_type(imm.index);
    return true;
}
```

Which for validation calls the module's `has_struct()` method from v8/src/wasm/wasm-module.h:

```
bool has_struct(uint32_t index) const {
    return index < types.size() && types[index].kind == TypeDefinition::kStruct;
}
```

As we have grown the size of the module's `types` beyond `kV8MaxWasmTypes` this check will pass without problems even if we pass something larger than this value, allowing us to create a reference of an arbitrary internal type (which actually points to the struct we can freely define).

On the other hand, consider now the handling of the `ref.cast` instruction:

```
      case kExprRefCast:
      case kExprRefCastNull: {
        ...
        Value obj = Pop();

        HeapType target_type = imm.type;
        ...
          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);
            }
          }
        ...
      }
```

Here, a type check elimination is performed: If `TypeCheckAlwaysSucceeds` returns `true`, then no actual type check is emitted, and the value is simply reinterpreted as the target type.

The function `TypeCheckAlwaysSucceeds` then ultimately calls `IsHeapSubtypeOfImpl` defined in v8/src/wasm/wasm-subtyping.cc:

```
V8_NOINLINE V8_EXPORT_PRIVATE bool IsHeapSubtypeOfImpl(
    HeapType sub_heap, HeapType super_heap, const WasmModule* sub_module,
    const WasmModule* super_module) {
  if (IsShared(sub_heap, sub_module) != IsShared(super_heap, super_module)) {
    return false;
  }
  HeapType::Representation sub_repr_non_shared =
      sub_heap.representation_non_shared();
  HeapType::Representation super_repr_non_shared =
      super_heap.representation_non_shared();
  switch (sub_repr_non_shared) {
    ...
    case HeapType::kNone:
      // none is a subtype of every non-func, non-extern and non-exn reference
      // type under wasm-gc.
      if (super_heap.is_index()) {
        return !super_module->has_signature(super_heap.ref_index());
      }
      ...
  }
  ...
}
```

This means that if our declared type index aliases the constant `HeapType::kNone`, the type check will always be elided if we cast to any non-function, non-external reference.

In combination, we can use this to turn any reference type into any other by the following steps:

1. In the type section define a structure type with a single entry of type `ref any`, and make it have a type index equal to `HeapType::kNone` by the bug described in section 2.1.
2. Have a (non-null) reference value of any type on the top of the stack and call `struct.new` with the type index equal to `HeapType::kNone`. This will succeed, as `has_struct()` finds the struct that was declared in the previous step.
3. Also, declare a struct (with a normal type index lower than `kV8MaxWasmTypes`) with a single member of the target reference type. Call `ref.cast` with the type index equal to this struct's one. This will not perform any type check, as the input value is at this point understood to be reference type `HeapType::kNone`.
4. Finally, read back the reference stored in the struct by executing `struct.get`.

This arbitrary casting of reference types allows *transmuting* any value type into any other by referencing it, changing the reference type, then dereferencing it - a universal type confusion.

In particular, this directly contains nearly all usual JavaScript engine exploitation primitives as special cases: [3]

- Transmuting `int` to `int*` and then dereferencing results in an arbitrary read
- Transmuting `int` to `int*` and then writing to that reference results in an arbitrary write
- Transmuting `externref` to `int` is the `addrOf()` primitive
- Transmuting `int` to `externref` is the `fakeObj()` primitive[4]

Note however that these "arbitrary" reads and writes are still contained in the V8 memory sandbox, as all involved pointers to heap-allocated structures are tagged, compressed pointers inside the heap cage, not full 64-bit raw pointers.

3 Integer Underflow leading to V8 Sandbox Escape

The primitives described in section 2.3 allow for freely manipulating and faking most JavaScript objects - however, all of this happens inside the limited memory space of the V8 sandbox. "Trusted" objects like WebAssembly instance data cannot yet be manipulated.

An often-used object for JavaScript engine exploits is the `ArrayBuffer` (and its corresponding views, i.e. typed arrays), as it allows for direct, untagged access to some region of memory.

To prevent access to pointers outside the V8 sandbox, sandboxed pointers are used to designate a typed array's corresponding backing store. Similarly, an `ArrayBuffer`'s length field is always loaded as a "bounded size access", inherently limiting its value to a maximum of $2^{35}-1$.

However, in recent JavaScript the handling of typed arrays has become quite complex due to the introduction of resizable `ArrayBuffer`s ("RABs") and their sharable variant, growable `SharedArrayBuffer`s ("GSABs")[2]. Both of these variants feature the ability to change their length after the object has been created (with the shared variant being restricted to never shrink). In particular, for typed arrays with these kinds of buffers, the array length can never be cached, but has to be recomputed on each access.

Additionally, `ArrayBuffer`s also feature an *offset* field, describing the start of the data in the actual underlying backing store. This offset has to be taken into account when computing the length.

Let's now look at the code responsible for building a `TypedArray`'s length access in the optimizing Turbofan compiler. It can be found in v8/src/compiler/graph-assembler.cc (most non-RAB/GSAB cases and the code responsible for dispatching are omitted for simplicity):

```
TNode<UintPtrT> BuildLength(TNode<JSArrayBufferView> view,
                            TNode<Context> context) {
    ...
    // 3) Length-tracking backed by RAB (JSArrayBuffer stores the length)
    auto RabTracking = [&]() {
```

[3] For simplicity, C-like types and pointer syntax are used here. Actually, `int*` should be thought of as "a reference to a struct with a single member of type `i32`", "dereferencing" as "calling `struct.get 0`", etc.

[4] While casting from `HeapType::kNone` to an `externref` was not allowed, remember that we are actually operating on one more level of indirection - transmuting to `externref` involves casting to *a reference to a struct* containing one `externref` member

```
      TNode<UintPtrT> byte_length = MachineLoadField<UintPtrT>(
          AccessBuilder::ForJSArrayBufferByteLength(), buffer, UseInfo::Word());
      TNode<UintPtrT> byte_offset = MachineLoadField<UintPtrT>(
          AccessBuilder::ForJSArrayBufferViewByteOffset(), view,
          UseInfo::Word());

      return a
          .MachineSelectIf<UintPtrT>( // (1)
              a.UintPtrLessThanOrEqual(byte_offset, byte_length))
          .Then([&]() {
            // length = floor((byte_length - byte_offset) / element_size)
            return a.UintPtrDiv(a.UintPtrSub(byte_length, byte_offset),
                                a.ChangeUint32ToUintPtr(element_size));
          })
          .Else([&]() { return a.UintPtrConstant(0); })
          .ExpectTrue()
          .Value();
    };

    // 4) Length-tracking backed by GSAB (BackingStore stores the length)
    auto GsabTracking = [&]() {
      TNode<Number> temp = TNode<Number>::UncheckedCast(a.TypeGuard(
          TypeCache::Get()->kJSArrayBufferViewByteLengthType,
          a.JSCallRuntime1(Runtime::kGrowableSharedArrayBufferByteLength,
                           buffer, context, base::nullopt,
                           Operator::kNoWrite)));
      TNode<UintPtrT> byte_length =
          a.EnterMachineGraph<UintPtrT>(temp, UseInfo::Word());
      TNode<UintPtrT> byte_offset = MachineLoadField<UintPtrT>(
          AccessBuilder::ForJSArrayBufferViewByteOffset(), view,
          UseInfo::Word());
      // (2)
      return a.UintPtrDiv(a.UintPtrSub(byte_length, byte_offset),
                          a.ChangeUint32ToUintPtr(element_size));
    };
    ...
}
```

For arrays backed by a resizable ArrayBuffer, we can see at (1) that the length is computed as `floor((byte_length - byte_offset) / element_size)`. Crucially, there is an underflow check: If `byte_offset` exceeds `byte_length`, then 0 is returned instead.

Curiously though, in the case of a GSAB-backed array, the same overflow check is missing. Thus, if `byte_offset` is larger than `byte_length`, an underflow occurs and the subtraction wraps around to something close to the maximum unsigned 64-bit integer 2^{64}. As both of these fields are found in the (by now) attacker-controlled array object, we can easily make this occur using the sandboxed arbitrary read/write primitives from section 2. This results in access to the whole 64-bit address space, as the length computed by this function is used to bound any

typed array accesses (in JIT-compiled code).

4 Exploitation

Using the previous two bugs, exploitation becomes fairly straight-forward. The primitives described in section 2.3 directly give arbitrary reads and writes in the V8 memory sandbox, which can then be used to manipulate a growable `SharedArrayBuffer` to have an offset greater than its length. A previously JIT-compiled read/write function can then be used to access and overwrite data in the whole address space, including the compiled bytecode of a WebAssembly module located on an RWX page.

References

[1] Samuel Groß. *V8 Sandbox*. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8/. 2021.

[2] Shu-yu Guo. *In-Place Resizable and Growable ArrayBuffers*. https://github.com/tc39/proposal-resizablearraybuffer.

[3] *WebAssembly Specification, Release 2.0 + tail calls + function references + gc (Draft 2024-03-06)*. Version 2.0. https://webassembly.github.io/gc/core/_download/WebAssembly.pdf. W3C, Mar. 6, 2024. URL: https://webassembly.github.io/gc/core/index.html.