

0x00000000	00 61 73 6D 01 00 00 00 00 DB 01 00	04 74 65 73	.asm.....tes	(module
0x00000010	74 2D 2D 3E 0A 3C 64 69 76 20 69 64 3D 27 6D 79		t-->.<div id='my	(type \$type0 (func (result i32)))
0x00000020	70 61 67 65 27 3E 0A 3C 73 63 72 69 70 74 20 6C		page'>.<script l	(table \$table0 0 anyfunc)
0x00000030	61 6E 67 75 61 67 65 3D 6A 61 76 61 73 63 72 69		anguage=javascri	(memory (;0;) 1)
0x00000040	70 74 20 74 79 70 65 3D 22 74 65 78 74 2F 6A 61		pt type="text/ja	(export "memory" (memory 0))
				(export "hello" (func \$func0))

How to create polyglot HTML/JS/WebAssembly module

Just a bit of context first, so last month I was at the hack.lu conference to give a workshop about **“Reversing WebAssembly module 101”** and spend some amazing time with friends. Workshop goes well, attendees were really interested and even better I received the award of the best talk/workshop of hack.lu 😊

At the bar during the conference, I was talking with **Ange Albertini (@angealbertini)** i.e. in my opinion, the **“Polyglot Grand Master and File Format Ninja”** ©. The result of this discussion is this short blogpost to share with you, Ange was right and creating polyglot file is not so complicated 😊

In this blogpost, I will first explain the **WebAssembly binary format** and its **sections**. Then, I'll demonstrate how to create a valid polyglot wasm module that contain an **html/js payload embedded** using 2 different techniques. Finally, I'll give you the link to the github repository if you want to try on your own and learn more about WebAssembly 😊

Just a quick reminder before we start, if you are interested about **WebAssembly security**, my next public 4-days [training](#) will be:

- 29 March – 01 April 2020 / 🇸🇬 Singapore ==> [SHACK/WhiskeyCon](#).
- 20 – 22 April 2020 / 🇳🇱 Amsterdam ==> [HITB](#).
- 01 – 04 August 2020 / 🇺🇸 Las Vegas ==> [Ringzer0](#).
- Onsite trainings ==> [here](#).



My awesome award from hack.lu 😊

1. What's a polyglot file?

Let's start by the definition of [Wikipedia](#):

*“a **polyglot** is a [computer program](#) or [script](#) written in a valid form of multiple [programming languages](#), which performs the same operations or output independent of the programming language used to [compile](#) or [interpret](#) it”.*

In my opinion, this definition is too restrictive and too specific to “Polyglot Programming”. A polyglot file, after being executed/parsed by different programs, will rarely lead to the same output. For example, Javascript/BMP polymorphic files has been used in the wild to [hide malicious payload](#) only if the file is interpreted as a JS file. If you want to discover more about polymorphic files, you should take a look at the following resources:

- “[Funky File Formats](#)” talk by [@angealbertini](#) – [slides](#), [video](#)
- Compilation of polyglots resources – [github](#)
- [TruePolyglot](#) project – [website](#), [github](#)

2. WebAssembly binary format, in short.

So, the first bytes of a valid WebAssembly module are the **magic number** ‘\0asm’ (i.e. null byte ASM). Following, there is a **4 bytes version number**, fixed to the value 0x1 since the release of MVP 1.0.

This preamble of **2 fields is enough to create a valid WebAssembly module** but if you change the value of this version field, some WebAssembly parsers and VMs will reject your module.

Field	Type	Description
magic number	uint32	Magic number <code>0x6d736100</code> (i.e., '\0asm')
version	uint32	Version number, <code>0x1</code>

WebAssembly module header

```

/tmp > printf "\x00\x61\x73\x6D\x01\x00\x00\x00" > mini.wasm
-----
/tmp > file mini.wasm
mini.wasm: WebAssembly (wasm) binary module version 0x1 (MVP)
-----
/tmp > wasm-validate mini.wasm && echo $?
0
-----
/tmp > wasm-validate mini.wasm -v
BeginModule(version: 1)
EndModule

```

Creating a minimal wasm module in one line

The module preamble is followed by **a sequence of sections**. Each section is identified by a 1-byte *section code* (0-11) that encodes either a **known section** or a **custom section**. Each known section is **optional and may appear at most once**.

I'll not detailed more in this blogpost which kind of information you will find in each known sections, but if you want to go deeper, start looking here:

- WebAssembly Binary Encoding – [link](#)
- Introduction to WebAssembly – [link](#)
- Understanding WebAssembly text format – [link](#)

Section Name	Code	Description
Type	1	Function signature declarations
Import	2	Import declarations
Function	3	Function declarations
Table	4	Indirect function table and other tables
Memory	5	Memory attributes
Global	6	Global declarations
Export	7	Exports
Start	8	Start function declaration
Element	9	Elements section
Code	10	Function bodies (code)
Data	11	Data segments

WebAssembly module know sections

3. HelloWorld & WebAssembly Data section

Let's start with a basic "HelloWorld". WebAssembly module can be compiled from C/C++/Rust/... source code or directly written using the [WebAssembly text representation](#) (wat/wast).

The following module (right-picture) will return the offset of the string **"hello from WebAssembly !"** when the function **"hello"** will be called by a VM. This offset is a **pointer to the string stored in the linear memory**, i.e. the memory shared by the wasm module and the WebAssembly VM.

As you can see, this **data section allow us to store completely arbitrary strings** inside a module, exactly what we need to inject some HTML/JS payloads.

```
$ wasm-dis hello.wasm
(module
  (type $0 (func (result i32)))
  (memory $0 1)
  (data (i32.const 16) "hello from WebAssembly !\00")
  (table $0 0 funcref)
  (export "memory" (memory $0))
  (export "hello" (func $0))
  (func $0 (; 0 ;) (type $0) (result i32)
    (i32.const 16)
  )
)
```

Disassembly - HelloWorld in WebAssembly text format

4. WebAssembly Custom section

The data section is actually not the only section that can be used to store arbitrary strings. The **custom section have been exactly designed with this goal in mind**. For example, if a developer want to store [DWARF debug information](#) inside a module, during compilation of the module, a bunch of customs sections will be embedded in the wasm file itself with different DWARF section name for each (ex: .debug_info, .debug_line, etc.).

Injection of arbitrary strings/payload using this technique will only require to **calculate correctly fields lengths to create a valid custom section**.

Custom section need to have the name & name_len field

5. Awesome HTML/JS payload needed!

HTML/JS payload to inject inside the module

Since I'm not an expert in polyglot file, I've asked [@angealbertini](#) and he kindly provided me this HTML/JS payload. In short, the payload use [InnerHTML](#) to **prevent the browser to parse the totality of the file**. It should be possible to create another valid payload without InnerHTML but no need to make your life more complicated 😊 More details about this payload and trick can be found [here](#).

Finally, this payload needs to be **embedded inside a WebAssembly module using either the data section or a custom section**.

6. Data section injection (1st technique)

Let's start with the easiest technique first. I choose to directly modify the WebAssembly text representation of the [previous](#) helloworld module and I've added an extra line to it:

```
(data (i32.const 42) "PAYLOAD_HERE")
```

This line will create a new "segment", into the data section of the module, with my payload inside. Then, I've translate this wasm text file to a wasm module (named [wasm_polyglot_data.wasm](#)) using [wat2wasm](#).

Payload injected in the data section using wasm text format

7. Custom section injection (2nd technique)

For this second technique, I just create a simple python script (available [here](#)) that take **my payload** and **my helloworld module** in order to **concatenate them into one single binary file** (named **wasm_polyglot_custom.wasm**). Only particularity, I've inject the custom section (with the html/js payload inside) at the beginning of the final wasm module i.e. just after the Webassembly header. This **custom section becomes the first section of the module** but it does not affect module validation.

Finally, you can verify the internal structure of your module with [wasmcodeexplorer](#) and you should get something similar to the following screenshot.

8. Is it working?

First, we need to check if our new WebAssembly modules are still valid:

- [wasm_polyglot_data.wasm](#)
- [wasm_polyglot_custom.wasm](#)

You can use standalone tools like [wasm-validate](#) or you can directly try to instantiate those modules with a WebAssembly VM like [wasmer](#), [wasmtime](#) or [WAVM](#). Using Javascript, you can use the [WebAssembly.validate\(\)](#) or [WebAssembly.instantiate\(\)](#) APIs.

For your verification, run a web server locally and open this script (picture on the right – source [here](#)). You should get some messages logs in the developer JS console. In short, this script will fetch our polyglot wasm module, call the exported function **hello()** and finally print the helloworld string.

JS logs when polyglot instantiated and called using WebAssembly API

Now, rename the polyglot file extension **from .wasm to .html**, open the file in your browser and you should see the following alert message !

Alert when polyglot file interpreted as an HTML file by the browser

Simple JS script calling hello wasm function

If you try to directly fetch the `wasm_polyglot.html` file using `instantiateStreaming()`, you will get this Javascript error message: `"Failed to execute 'compile' on 'WebAssembly': Incorrect response MIME type. Expected 'application/wasm'."`

You can bypass this MIME verification by fetching the module, storing the buffer content and finally compile/instantiate the module using [other WebAssembly APIs](#) 😊

9. Conclusion

I hope that you have learn and discover some new WebAssembly tricks over this blogpost ;). Of course, if you want to go deeper after the reading, you can try to apply those techniques with the data & custom section to create PDF/GIF/... polyglot WebAssembly module. Thanks again to **Ange Albertini (@angealbertini)** for the payload and advises regarding polyglot files in general !!

All the WebAssembly module, JS files and scripts shown in this blogpost are available in [this github repository](#). Final reminder, If you want to learn/discover more about **WebAssembly security**, my next public 4-days [training](#) will be:

- 29 March – 01 April 2020 / Singapore ==> [SHACK/WhiskeyCon](#).
- 20 – 22 April 2020 / Amsterdam ==> [HITB](#).
- 01 – 04 August 2020 / Las Vegas ==> [Ringzer0](#).
- Onsite trainings ==> [here](#).

Patrick Ventuzelo / [@Pat_Ventuzelo](#)



UPCOMING TRAININGS

REQUEST ON-SITE QUOTE

© 2019 - Patrick Ventuzelo | [Contact](#) | [Trainings](#)