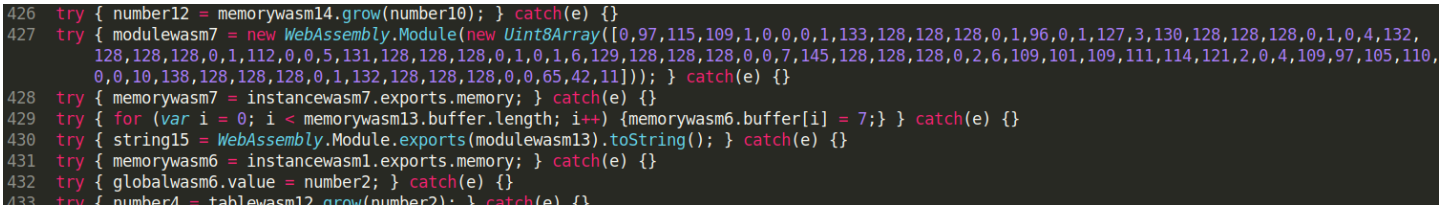


```
try { number12 = memorywasm14.grow(number10); } catch(e) {}
try { modulewasm7 = new WebAssembly.Module(new Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,
    128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,128,128,0,0,7,145,128,128,128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,110,
    0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11])); } catch(e) {}
try { memorywasm7 = instancewasm7.exports.memory; } catch(e) {}
try { for (var i = 0; i < memorywasm13.buffer.length; i++) {memorywasm6.buffer[i] = 7;} } catch(e) {}
try { string15 = WebAssembly.Module.exports(modulewasm13).toString(); } catch(e) {}
try { memorywasm6 = instancewasm1.exports.memory; } catch(e) {}
try { globalwasm6.value = number2; } catch(e) {}
```

Fuzzing JavaScript WebAssembly APIs using Dharma/Domato (on Chrome/V8)

First of all, Happy new hacking year everyone 😉

I got asked multiple time if fuzzing WebAssembly APIs of Javascript engines is complicated, so here is a short tutorial using Dharma (but you can use Domato if you prefer).

In this blogpost, I will first detailed **WebAssembly Javascript APIs supported by major browsers.** Then, I'll explains how to use Dharma to generate valid Javascript file to fuzz **WebAssembly** APIs. Finally, I'll show an easy way to execute those generated testcases over ASAN build of Chrome/V8.

Just a quick reminder before we start, if you are interested about **WebAssembly security and fuzzing WebAssembly Browsers/VMs**, my next publics trainings will be in:

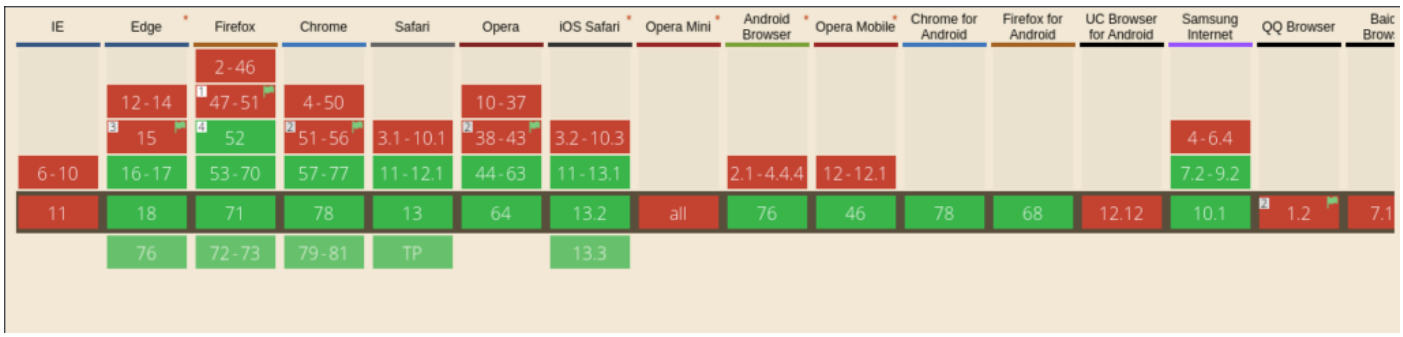
- 29 March – 01 April 2020 / 🇸🇬 Singapore ==> SHACK/WhiskeyCon.
- 20 – 22 April 2020 / 🇳🇱 Amsterdam ==> HITB.
- 01 – 04 August 2020 / 🇺🇸 Las Vegas ==> Ringzer0.
- Onsite trainings ==> here.

---

1. WebAssembly & Web-Browser CVEs

Long story short, WebAssembly (abbreviated *Wasm*) is a new portable, size- and load-time-efficient **binary instruction format for the web**. It has been designed by members of people representing the four browsers, Chrome, Edge, Firefox, and WebKit. WebAssembly is now **supported/enabled by default by every major browsers** (both on Desktop and Mobile) and WebAssembly module are **executed through their respective javascript engines** such as v8, spidermonkey, jsc, etc.

As you can see on caniuse.com, in January 2020, around 88% of all internet users can run WebAssembly modules on their browsers.



Support of WebAssembly across browsers (01/2020)

Adding supports of a new binary format designed to be loaded and executed by Javascript engines implied of course some huge modification on the source code i.e. from a researcher point of view, a new attack surface to discover 😉

Notably, some security issues related to WebAssembly APIs has been found in V8, WebKit and Firefox by Natalie Silvanovich (@natashenka) but also in the closed-source Xiaomi Mi6 Browser by @fluoroacetate with CVE-2019-6743.

If you want to discover more about existing WebAssembly browser CVEs, take a look at those links:

- **The Problems and Promise of WebAssembly** by *natashenka* – blogpost, slides
- **A collection of JavaScript engine CVEs with PoCs** – github
- **Apple Safari Wasm Section Exploit CVE 2018-04-16** by *MWR-labs* – write-up

---

2. WebAssembly JavaScript APIs

WebAssembly JavaScript APIs are composed of **multiple methods** and WebAssembly **object constructors** used to created and instantiate wasm modules, such as:

- `WebAssembly.instantiate()` method.
- `WebAssembly.compile()` method.
- `WebAssembly.validate()` method.
- `WebAssembly.Global()` object.
- `WebAssembly.Module()` object.
- `WebAssembly.Instance()` object.
- `WebAssembly.Memory()` object.
- `WebAssembly.Table()` object.

Those APIs are our first targets for fuzzing since they are **involved in multiple CVEs** and they are **common to every browser/JavaScript engine** implementation. More details about those APIs and their syntaxes can be found here:

- **WebAssembly** by *Mozilla MDN* – link
- **WebAssembly Web API** by *W3C* – link
- **Understanding the JS API** – link

---

3. Create Dharma/Domato WebAssembly APIs grammars

Dharma is a generation-based, context-free grammar fuzzer created in 2015 by *Christoph Diehl* from Mozilla Security team. The goal of this tool is to **generate files (like Javascript and/or HTML)** based on a **given grammar description.** This concept look maybe familiar to you if you already played with Domato by *Ivan Fratric* from Google Project Zero.

Personally, I prefer the grammar syntax of Dharma but if you are a Domato adept, converting the following grammar to Domato shouldn't be difficult 😉


```
%%% ######## WebAssembly.Global() values ###########
%%% ## https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Global

WasmTypeInt :=
    "i32"
    "i64"

WasmTypeFloat :=
    "f32"
    "f64"

GlobalDescriptorFloat :=
    {value: +WasmTypeFloat+, mutable: +common:bool+}

GlobalDescriptorInt :=
    {value: +WasmTypeInt+, mutable: +common:bool+}

GlobalParameters :=
    +GlobalDescriptorFloat+, +common:decimal_number+
    +GlobalDescriptorInt+, +common:integer+

GlobalWasmMethods :=
    !globalwasm!.value = !number!;
    !globalwasm!.value = +common:number+;
    !number! = !globalwasm!.value;
    !number! = !globalwasm!.valueOf();
    !string! = !globalwasm!.toString();
```

Dharma grammar for WebAssembly.Global object

The most time consuming part is to read all the specification and/or APIs descriptions in order to create valid wasm objects and methods calls. I will not detailed Dharma grammar syntax in this blogpost but you can find a complete grammar cheatsheet on the official github repository of Dharma. Once your grammar seems acceptable, you can generate multiple files using this command:

```
dharma -grammars dharma/wasm.dg -count 100 -format js -seed 1337 -storage output_folder
```

Your output folder should now contain multiple JavaScript files similar to the following picture.

```
try { number12 = memorywasm14.grow(number10); } catch(e) {}
try { modulewasm7 = new WebAssembly.Module(new Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,
    128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,128,128,0,0,7,145,128,128,128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,110,
    0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11])); } catch(e) {}
try { memorywasm7 = instancewasm7.exports.memory; } catch(e) {}
try { for (var i = 0; i < memorywasm13.buffer.length; i++) {memorywasm6.buffer[i] = 7;} } catch(e) {}
try { string15 = WebAssembly.Module.exports(modulewasm13).toString(); } catch(e) {}
try { memorywasm6 = instancewasm1.exports.memory; } catch(e) {}
try { globalwasm6.value = number2; } catch(e) {}
try { number4 = tablewasm12.grow(number2); } catch(e) {}
try { number4 = globalwasm13.valueOf(); } catch(e) {}
try { number2 = memorywasm2.buffer.length - 1; } catch(e) {}
try { number13 = tablewasm10.grow(number12); } catch(e) {}
try { memorywasm1 = instancewasm6.exports.memory; } catch(e) {}
try { array6 = WebAssembly.Module.imports(modulewasm1); } catch(e) {}
try { array2 = WebAssembly.Module.customSections(modulewasm4, "debug"); } catch(e) {}
try { string15 = globalwasm10.toString(); } catch(e) {}
try { globalwasm3.value = number3; } catch(e) {}
try { instancewasm12.exports.main(); } catch(e) {}
```

Snippet of WebAssembly APIs calls generated by our dharma grammar

I just published a simple WebAssembly grammar in this github repository if you need something to start 😉 I also invited you to read the following blogposts to discover how other researchers are using Dharma or Domato 😉

- **Vulnerability Discovery Against Apple Safari** by *ret2systems* – link
- **Implementing fuzz logics with dharma** by *Mat Powell* – link
- **Domato Fuzzer's Generation Engine Internals** by *Jaewon Min* – link
- **Fuzzing PHP with Domato** by *Andrew Kramer* – link
- **Using dharma to rediscover node.js out-of-band write in utf8 decoder** by *NibbleSecurity* – link

---

4. JavaScript engines built with ASan (Chrome/V8)

In this blogpost, I will only target Chrome/V8 because it's the best way for you (readers) to reproduce blogpost's steps at home **without spending hours in compilation/debugging**. Just a quick reminder, AddressSanitizer (ASan) is a **memory error detector based** on compiler instrumentation (LLVM) and used to detect multiple kind vulnerabilities (UaF, HBoF, etc.)

Here is the main reasons why I choose Chrome/V8 as a first choice:

- Google provide pre-built Chrome binaries **built with AddressSanitizer** i.e. no compilation for me today 😉
- One of the binary provided is d8, the V8's own developer **command line shell**.
- You can specify to d8 (through cmd line option) if you want to **activate under-development WebAssembly features** (like anyref, threads, simd, etc.)

- Pre-built **are fresh** (up to date) and **easy to download** using Google gsutil tool and the following command.
- **`gsutil cp $(gsutil ls "gs://chromium-browser-asan/linux-release/asan-linux-release-*.zip" | tail -1) .`**

If you are looking to compile other JavaScript engines with ASan, check instructions in the links bellow:

- Building **WebKit/JSC** with ASan – link
- **Firefox/Spidermonkey** ASan builds + compilation intructions – link
- **V8** build tests with ASan – link

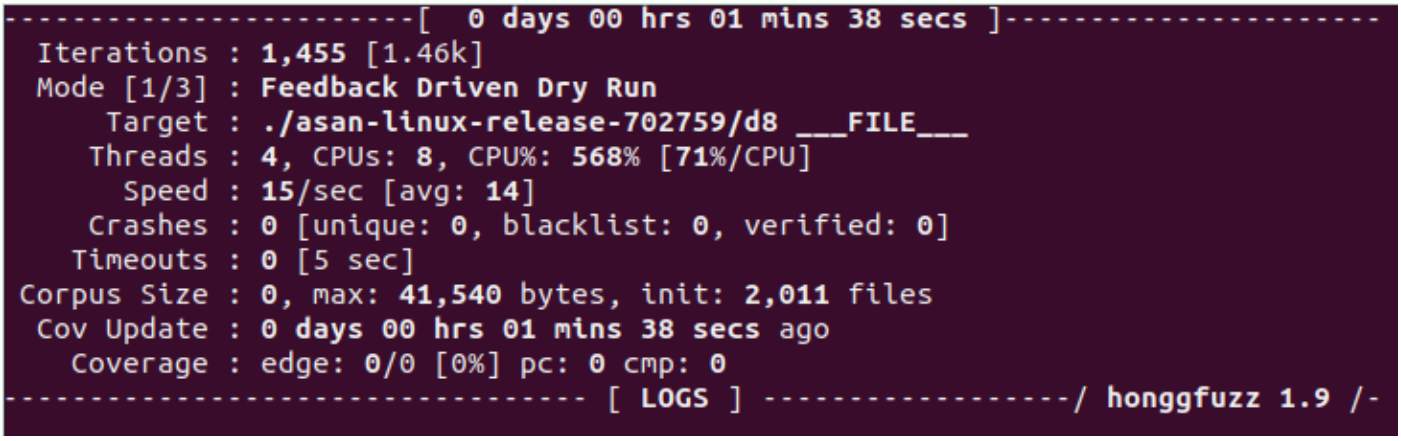
---

5. Fuzzing & monitoring - the lazy way

Last but not least, we need to provide our JS files to d8. An easy way to start can be to create a **simple bash script** (like this one) that will loop and:

- **execute d8** binary **for each js file** generated by Dharma
- **monitor the returned signal value** of the program
- **store the testcase** for analysis **if the signal is not null.**

This logic here is really basic and not optimal off course. If you don't want to wrote some shell script, I can only suggest you to **try the lazy way** i.e. generate a lot of test-cases and let honggfuzz deal with them.



Honggfuzz dry-run with ASan d8 build and dharma generated files

Honggfuzz is really **easy-to-use** and awesome fuzzer developed and maintained by Robert Swiecki from Google. During honggfuzz "dry-run", all given files will be **executed in different threads, monitored for crashes and stored if relevants**. That also mean that only using the following command-line, you let honggfuzz handle everything and can go to sleep 😉

```
honggfuzz -t 5 -n 4 -i input_wasm_js/ -- ./d8 ___FILE___
```

Note: I noticed during my research that someone also integrate dharma directly into honggfuzz. I let you the check here, but personally I was not able to make it work :s

---

5. Going deeper & Conclusion

Congratz, you are now able to **fuzz V8 WebAssembly APIs using generation-based fuzzers.** The previous dharma grammar is really specific to WebAssembly APIs and should be improved to handle more generic JavaScript methods/objects (Array, UIntArray, Number, etc.). Also, another grammar should be **created to generate valid WebAssembly module bytecode**, stored inside ArrayBuffer or TypedBuffer and provided to the WebAssembly.Module() constructor.

Grammar and script shown in this blogpost are available in this github repository. Final reminder, If you want to learn/discover more about **WebAssembly security (both reversing and fuzzing)**, my next public trainings will be in:

- 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
