Web Assembly: Overview, Security and Detection Opportunities

Pierre Chifflier

Agence Nationale de la Sécurité des Systèmes d'Information

2021

Who am I

- Pierre Chifflier
- Head of Detection Research Lab at ANSSI
- Security, compilers and languages
- Rust enthousiast (Parse all the things!)
- Suricata contributor since 2010

Agenda

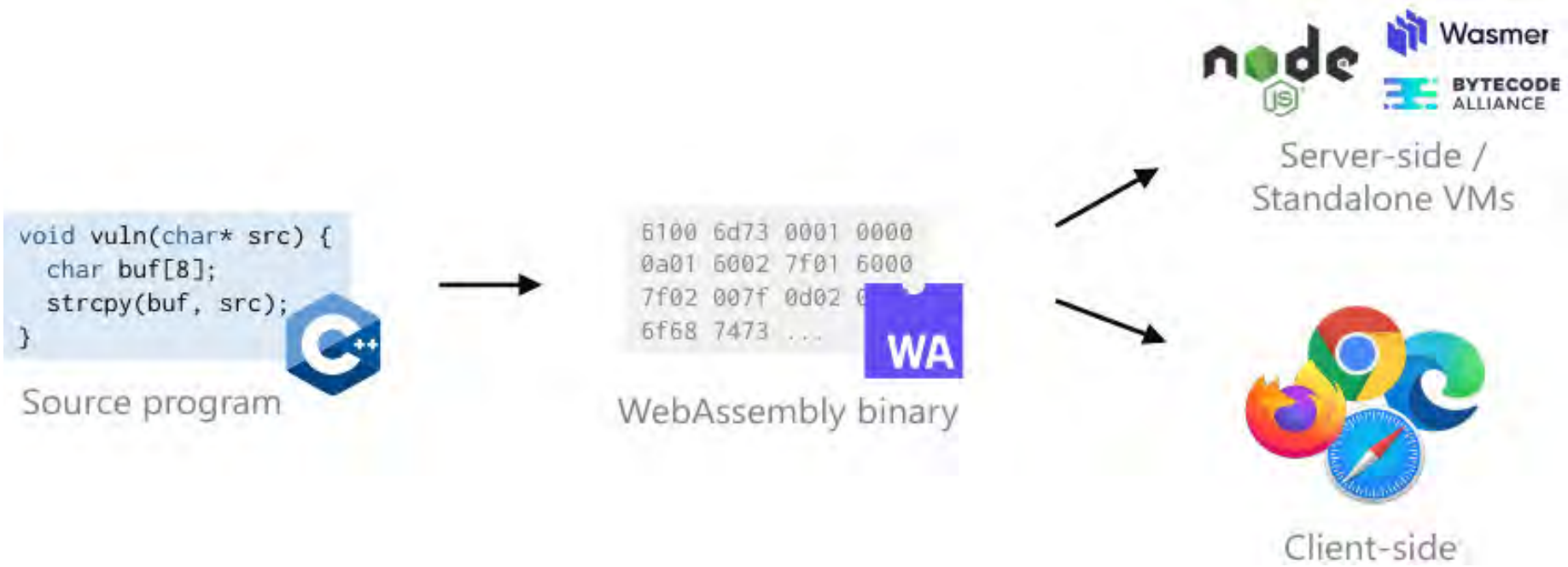
1. What is WebAssembly
2. Security of WASM Execution Environment
3. Malwares using WASM
4. WASM: challenges for detection tools
5. Conclusion

What is WebAssembly

What is WebAssembly

- WebAssembly (WASM) is a set of specifications
  - A *binary instruction format* for a stack-based *virtual machine*
  - Developped by W3C Community Group (2015)
  - Defined *mostly* for the web environment
  - Goals:
    - Provide an environment for execution of *client-side* application
    - Provide close to native code execution performance
    - Provide isolation for executed code
- *"WebAssembly is neither web nor assembly!"*

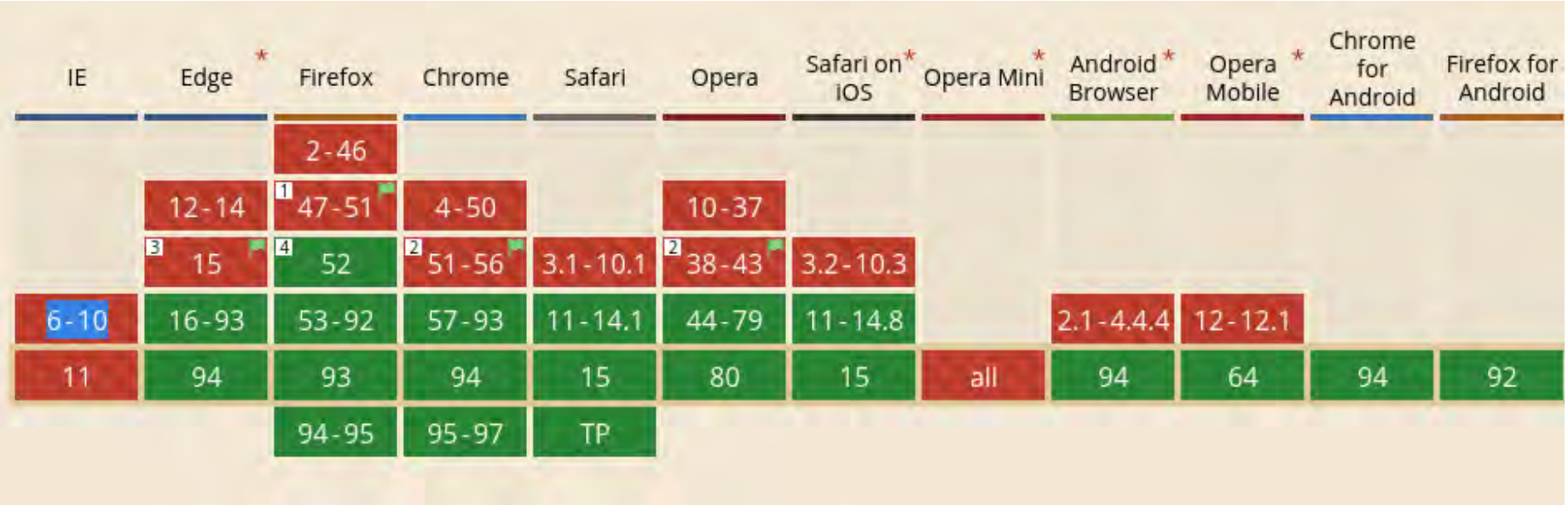
What is it, really?



(source: [2])

- Fast, portable bytecode
  - *Write once, run everywhere*
- Compiled from C, C++, Rust, Go, ...
- Supported in browsers, Node.js, standalone VMs
- Serverless apps, IoT, smart contracts, ...

WASM for browsers



IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android
		2-46									
	12-14	1 47-51	4-50		10-37						
	3 15	4 52	2 51-56	3.1-10.1	2 38-43	3.2-10.3					
6-10	16-93	53-92	57-93	11-14.1	44-79	11-14.8		2.1-4.4.4	12-12.1		
11	94	93	94	15	80	15	all	94	64	94	92
		94-95	95-97	TP							

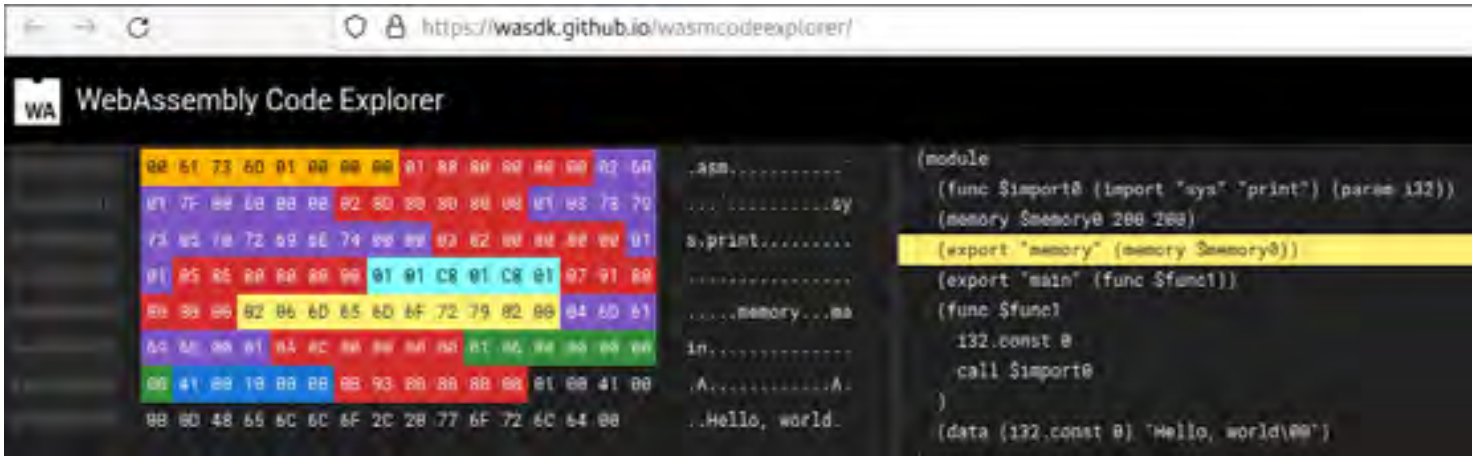
WASM support in browsers (source: https://caniuse.com/wasm)

- All major browsers support WASM (including smartphones)
- Supported by 94% of all browser installations as of October 2021

Used for many applications

- Game engines
  - Unity3D WebGL (WebAssembly is here!)
  - Unreal engine since 4.16
  - Examples: AngryBots, Funky Karts, ...
    - See https://www.webassemblygames.com/
  - lichess
- Huge web apps: Autocad, Google Earth on Web
- Blockchain: Ethereum (#2), EOS (#5)
- Vim
- Maybe soon LibreOffice on WASM
- Also in malwares!

Binary format - overview



WASM binary structure (source: https://wasdk.github.io/wasmcodeexplorer/)

- Compact
- Easy to verify
- Simple module structure

Field	Type	Description
magic number	`uint32`	`0x6d736100 (\0asm)`
version	`uint32`	`0x1`

Web Assembly Text (WAT)

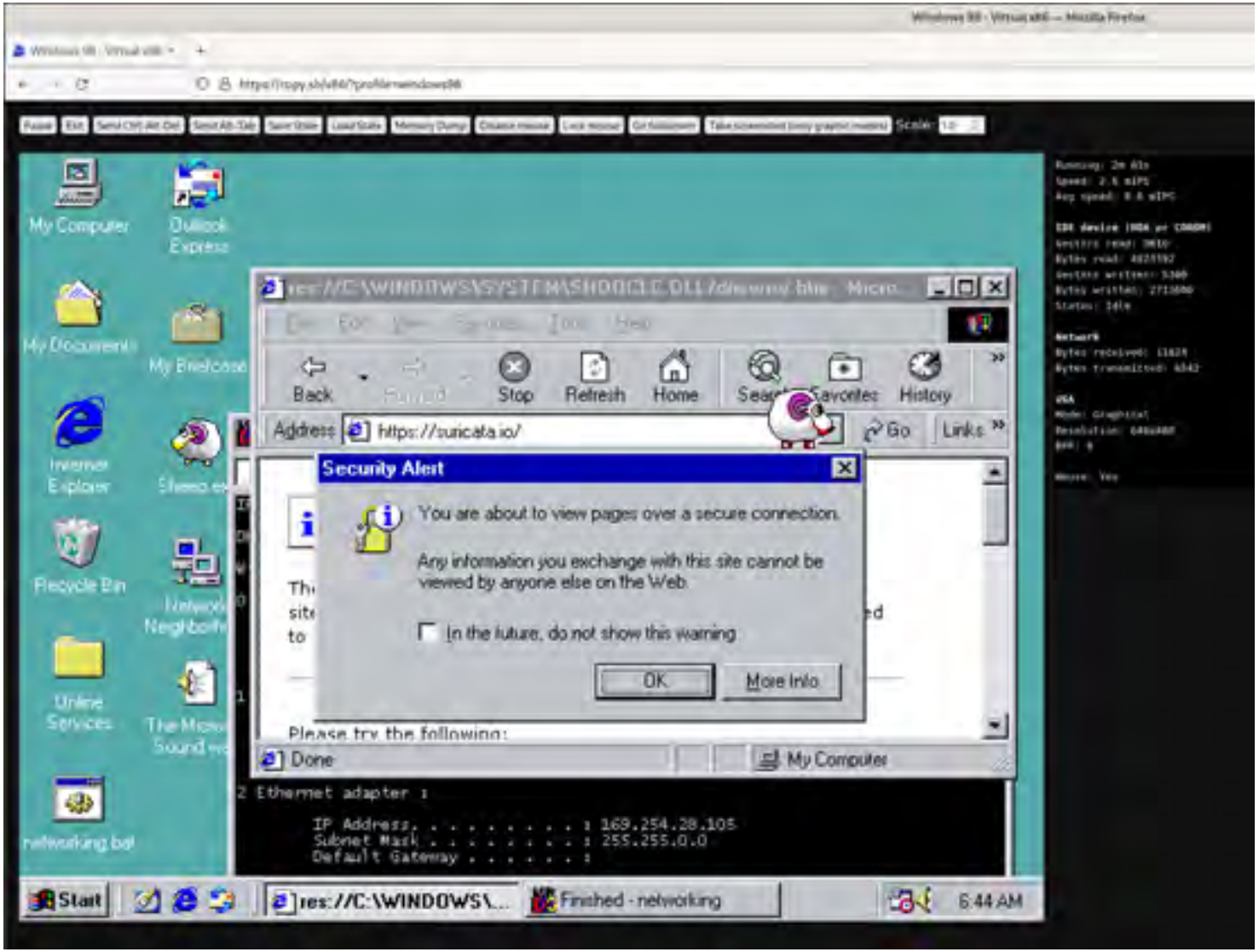
```
fn fibo(n: u32) -> u32 {
  if n <= 1 { return 1; }
  fibo(n-1) + fibo(n-2)
}
```

```
(func $fibo (type 9) (param i32) (result
    i32)
 01 7f          | local[0] type=i32
 41 01          | i32.const 1
 21 01          | local.set 1
 02 40          | block
 20 00          |   local.get 0
 41 02          |   i32.const 2
 49             |   i32.lt_u
 0d 00          |   br_if 0
 41 01          |   i32.const 1
 21 01          |   local.set 1
 03 40          |   loop
 20 00          |     local.get 0
 41 7f          |     i32.const 4294967295
 6a             |     i32.add
 10 01          |     call 1 <fibo>
 20 01          |     local.get 1
 6a             |     i32.add
 21 01          |     local.set 1
 20 00          |     local.get 0
 41 7e          |     i32.const 4294967294
 6a             |     i32.add
 22 00          |     local.tee 0
 41 01          |     i32.const 1
 4b             |     i32.gt_u
 0d 00          |     br_if 0
 0b             |   end
 0b             | end
```

Simplicity

- No threads, no SIMD, no exceptions, no garbage collection
- No access to outside world, for ex the DOM in web environment

Is it working? Is it fast?



WASM x86 emulator running Windows 98 in firefox (source: https://copy.sh/v86/?profile=windows98)

Security of WASM Execution Environment

Security

WASM is designed to run untrusted code:

- Linear Memory
  - No pointers, only indices
- Control-Flow Integrity
  - Function call by index (and not by address)
  - No return address
- Applications are isolated from the host and from each other
  - No access to host functions, syscalls
  - No access to files and I/O
- Very simple API: only integers can be used as arguments!
- No runtime nor standard library for executed module

Is WebAssembly the chosen one?

WASM modules security

The security of a WASM Module is defined by:

- The Module code
  - Programming errors, compiler errors, malicious intent, ...
- The VM isolation
  - implementation errors
- The features allowed by specifications

Not so simple

WebAssembly specifications are (currently) very simple

However:

- Additional specifications are proposed for threads, I/O (WASI), ...
- Browsers / Users are pushing for more features in specifications
  - All of these are intended for performance, not security

Compilation

However:

- Compiling from a source language means
  - Natural obfuscation/loss of information
  - Some errors kinds (programmic logic, integer overflows, insecure conversions etc.) will be transposed by the compiler
- Complex passing of arguments for host $\rightarrow$ function calls:
  - Passing buffers/strings requires some magic to allocate data in guest memory
  - These are complex/error-prone operations
- Compilers do not insert modern protections (write XOR execute, stack canaries, etc.)
  - Environment is supposed to make them unneeded

JIT compilation

WASM is usually compiled ahead-of-time (AOT) or just-in-time (JIT)

- Compilation is simple
- VM properties make verification of some properties rather easy

JIT compilation

WASM is usually compiled ahead-of-time (AOT) or just-in-time (JIT)

- Compilation is simple
- VM properties make verification of some properties rather easy

However:

- Some implementations remove the bounds checks after bytecode verification
- The security of the execution depends on the quality of this verification
- This exposes WASM modules to type confusion vulnerabilities (and more)
- This re-introduces potential memory problems (stack/heap overflows, etc.) [2]
- Modern execution protections are not inserted here either

Examples of implementation problems

See [9] for details:

- CVE-2018-4121 WebKit: WebAssembly parsing does not correctly check section order
- CVE-2017-5116 V8 engine exploit
- CVE-2018-4222 Info leak in WebAssembly compilation
- CVE-2018-6092 V8: integer overflow when processing WASM locals
- ...

(Not so) Good Ideas

- WASM code cannot call external functions (e.g. JS)
  - Unless *imported* in Module
  - Arguments are restricted (as usual)
- Which is good thing for isolation
- Until a framework provides a Get Out of Jail Free card:
  - `Emscripten` provides a `emscripten_run_script("text")` function
  - *.. to run the specified JavaScript from C/C++ using the browsers "eval()" function*
  - And other methods: see also `EM_JS()` and `EM_ASM()`

Consequences: easy code obfuscation, risks of JavaScript injection

```
#include <emscripten.h>

EM_JS(void, call_alert, (), {
  alert('hello world!');
  throw 'all done';
});
```

Overall Security of the WASM VM

WASM specifications define an isolated execution environment for untrusted code

- The temptation of ~~opening the gates of Hell~~ adding lots of new features and performance improvements is increasing
  - For ex, accessing the HTML DOM directly
- The specifications describe an ideal: real implementations differ a lot [7, 3]
- The provided isolation clearly depends on the implementation quality
  - But most implementations are driven by performance first
- This should not exempt the compiler from adding runtime verifications

Malwares using WASM

WASM for Malwares

- Loading and executing WASM code is very easy from JavaScript
- The portability of WASM bytecode, combined to binary compilation, makes it very interesting for malwares
- Performances and universal support makes it especially interesting for cryptominers [1]

Malwares using WASM

Quote from a 2019 study from Technische Universität Braunschweig [4] presented in DIMVA:

> *We found 48 unique samples on 913 sites in the Alexa Top 1 Million. (. . .) 56%, the majority of all WebAssembly usage in the Alexa Top 1 Million is for malicious purposes*

CoinHive: (il)legitimate web miner

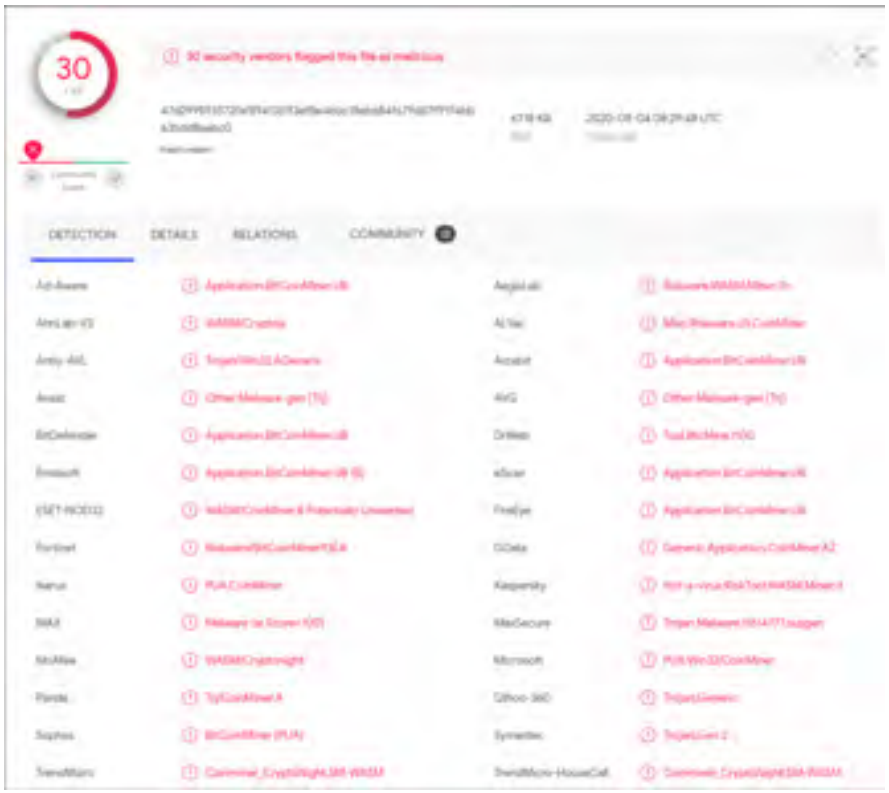
- CoinHive: a Monero cryptominer
  - Offers website owners a means to generate revenue outside of hosting ads
  - "Our miner uses WebAssembly and runs with about 65% of the performance of a native Miner."
- Emerged in 2017, quickly raised to number 1 Most Wanted Malware [6]

CoinHive spreading

- JS injected in Wordpress/Drupal websites
- Downloads WASM binary, starts mining and uses a coinhive WebSocket proxy
- Sample: 47d299593572faf8941351f3ef8e46bc18eb684f679d87f9194bb635dd8aabc0

Detection of CoinHive

- Names of exported functions
  - `_cryptonight_create`
  - `_cryptonight_destroy`
  - `_cryptonight_hash`
- Signatures based on those names
  - Yara, Snort, Suricata, AV
- Detection is very fragile!
  - Malware only has to change function names



Cryptonight VT detection

CoinHive Copycats

- Discontinued in 2019
- Other copycats followed: (JSECoin, Crypto-Loot, AFMiner, Coinhave)
  - Less obvious URLs
  - Obfuscated code and strings
- More details in [8]

Other malware types

- Content obfuscation
  - Malvertising campaigns, ...
- VM Escape/Browser exploitation
- XSS, KeyLoggers, ...

WASM: challenges for detection tools

Detection target for IDS

- WASM modules for browsers
- Files sent over TLS 1.3/Quic/HTTP3/...

New Challenges

- Binary Disassembly: very easy
- Code analysis: same cost as reverse engineering

Binary Analysis "on-the-fly"?

- WASM binaries tend to be very big
  - Compiler has to embed the language runtime
- Static and/or Dynamic Analysis required
- Lots of cross-language calls and FFI (for ex JS $\leftrightarrow$ WASM)
- Obfuscation is possible
  - Data: string literals, white-box encryption, Mixed Boolean-Arithmetic expressions, ...
  - Code: CFG flattening, Code Virtualization, indirect calls, ...

Is this the job of the IDS?

IDS Detection items: WASM files

WASM file detection:

- Usual extension `.wasm`
- File magic `00 61 73 6d`

Not all WASM files are malwares

- Content inspection:
  - YARA rules
  - Function names (if not removed)
  - Other items will require to inspect the WASM module!
    - Suspicious functions
    - Control flow properties
    - ...

IDS Detection items: other elements

Other available options:

- JavaScript wrapper:
  - `new WebAssembly.Instance(...)`
  - `WebAssembly.instantiate(...)`
  - `WebAssembly.instantiateStreaming(...)`

For cryptominers:

- Side-effects: keywords in TLS connections, WebSockets, etc.
- Known crypto-mining protocols, hosts or malware variants
- Block lists (for ex https://malware-research.org/coinblockerlists/)

We Need More Tools

Tools are either experimental or emerging:

- WABT: The WebAssembly Binary Toolkit
- Hacky Integration in IDA, Ghidra, etc.
- Analysis frameworks: Octopus, Wasabi, Manticore
- Academic work: MINOS [5]

Using these tools (or new tools) from the IDS would require a possibly complex integration

Conclusion

Thanks for listening

- WASM is already here, and is increasingly attracting malwares
- New code, old worries
- Analyzing binaries is tricky for an IDS

Will WebAssembly become the new Flash?

Références

Références

[1] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna.
Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense.
In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, page 1714–1730, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Daniel Lehmann, Johannes Kinder, and Michael Pradel.
Everything Old is New Again: Binary Security of WebAssembly.
In USENIX Security Symposium, 2020.

[3] Tyler Lukasiewicz and Justin Engler.
WebAssembly: A New World of Native Exploits on the Browser.
BlackHat USA, 2018.

[4] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck.
New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild.
In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, 2019.

[5] Faraz Naseem Naseem, Ahmet Aris, Leonardo Babun, Ege Tekiner, and Arif Selcuk Uluagac.
MINOS: A Lightweight Real-Time Cryptojacking Detection System.
In NDSS, 2021.

Références (cont.)

[6] Check Point.
September 2018's Most Wanted Malware: Cryptomining Attacks Against Apple Devices On The Rise.
2018.

[7] Natalie Silvanovich.
The Problems and Promise of WebAssembly.
BlackHat USA, 2018.

[8] Patrick Ventuzelo.
Analyze & Detect WebAssembly Cryptominer.
FIRST conference, 2019.

[9] Tiejun Wu and Guangyuan Zhao.
WASM Security Analyze And Reverse Engineering.
Botconf, 2018.