

Technical Analysis of Code-Signed “Blister” Malware Campaign (Part 2)

cloudsek.com/technical-analysis-of-code-signed-blister-malware-campaign-part-2/

Anandeshwar Unnikrishnan

February 17, 2022

The blister is a code-signed malware that drops a malicious DLL file on the victim’s system, which is then executed by the loader via rundll32.exe, resulting in the deployment of a RAT/C2 beacon, thus allowing unauthorized access to the target system over the internet. Blister Malware campaigns have been active since 15 September 2021.

Part I of CloudSEK’s analysis provides a detailed understanding of how the loader functions. Part 2 will delve into the details of this campaign’s second stage, which is the .dll payload, and its internal working.

Dissecting the Malicious DLL – Blister Malware

As discussed in Part 1, the Blister dropper drops the malicious .dll file in the Temp directory of the user, inside a newly created folder. This malicious .dll then carries out the second stage of the campaign, in which a RAT/ agent is deployed on the system to gain unauthorized access and steal data.

The Blister dropper calls the function LaunchColorCpl, which is one of the functions exported by the .dll, via rundll32.exe.

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00003A6C	0000	00011E20	LaunchColorCpl
00000002	000037AA	0001	00011E2F	DllCanUnloadNow
00000003	00005D62	0002	00011E3F	DllGetClassObject
00000004	000039F5	0003	00011E51	DllMain
00000005	000037D1	0004	00011E59	DllRegisterServer
00000006	000037DB	0005	00011E6B	DllUnregisterServer

Functions

exported by the malicious DLL

Staging

The exported function LaunchColorCpl retrieves the staging code from the resource section of the PE file. This staging code is protected by a simple XOR encoding scheme.

	1717390E	FFD7	call edi
	17173910	8BC6	mov eax,esi
	17173912	83E0 03	and eax,3
	17173915	8A4405 E8	mov al,byte ptr ss:[ebp+eax-18]
EIP	17173919	30041E	xor byte ptr ds:[esi+ebx],al
	1717391C	46	inc esi
	1717391D	81FE E0890100	cmp esi,189E0
	17173923	72 EB	jb holorui.17173910
	17173925	8D45 DC	lea eax,dword ptr ss:[ebp-24]

Code responsible for decoding the staging code

Address	Hex	ASCII
172040E6	50 00 00 E8 35 24 E8 8F 1A D4 F8 66 72 8A E8 8F	P..è\$è..òofr.è.
172040F6	C9 F9 56 BA 98 10 BA 66 69 2B E9 8F C9 F8 F3 AB	ÉúV°..°fi+è.Éóó«
17204106	9D 10 6B 48 89 F9 68 CF 98 10 00 81 BD 10 E8 0C	..kk.ùhĩ...%è.
17204116	50 00 01 72 05 10 E8 06 DC C8 01 57 B6 10 E8 DE	j..r..è.ÜE.w]èp
17204126	70 92 C7 8F 99 93 95 63 99 1F 6D 52 C8 11 E8 04	p.Ç...c.mRE.è.
17204136	1C 8C 1F 70 66 F9 EA D6 98 10 2B 0C E4 F4 E8 66	...pfueö...+.äöef
17204146	26 1C E8 8F 5E 55 2C 8F 99 10 E8 0F 62 A5 6B F2	&.è.^U,...è.b¥kò
17204156	5D 10 01 A0 89 10 E8 67 2B 8D E8 8F 1A D4 D0 06]...èg+.è..òD.

Encoded staging

code in the resource section of the PE file

- After the iterative decoding of the staging code, the control is transferred to decoded code in the memory.
- The control flow is transferred to the staging code by calling the address in the EAX register.

	17173910	8BC6	mov eax,esi
	17173912	83E0 03	and eax,3
	17173915	8A4405 E8	mov al,byte ptr ss:[ebp+eax-18]
	17173919	30041E	xor byte ptr ds:[esi+ebx],al
	1717391C	46	inc esi
	1717391D	81FE E0890100	cmp esi,189E0
	17173923	72 EB	jb holorui.17173910
	17173925	8D45 DC	lea eax,dword ptr ss:[ebp-24]
	17173928	50	push eax
	17173929	6A 20	push 20
	1717392B	8D45 F0	lea eax,dword ptr ss:[ebp-10]
	1717392E	50	push eax
	1717392F	8D45 EC	lea eax,dword ptr ss:[ebp-14]
	17173932	50	push eax
	17173933	6A FF	push FFFFFFFF
	17173935	FFD7	call edi
	17173937	8D45 E8	lea eax,dword ptr ss:[ebp-18]
	1717393A	50	push eax
	1717393B	8D83 905A0000	lea eax,dword ptr ds:[ebx+5A90]
	17173941	FFD0	call eax
	17173943	5F	pop edi
	17173944	5B	pop ebx
	17173945	33C0	xor eax,eax

Calling the

address in the EAX register

Anti-Analysis

- The staging code is heavily obfuscated, and has a logic similar to a spaghetti code, to hinder analysis. All the calls to Windows APIs are obscured and dynamically resolved.
- The first thing that the staging code does is to make the malware go to sleep by calling the Sleep Windows API. This is a typical strategy used by most malicious codes to bypass security sandboxes and dynamic testing of security products.

009FEA18	759F9010	kernel32.759F9010
009FEA1C	000927C0	
009FEA20	00000000	
009FEA24	009FEA5C	
009FEA28	00000000	
009FEA2C	E20400E6	

Stackframe before the malware calls the Sleep

Windows API

- The hex value “927C0” is passed to `kernel32.759F9010` i.e the *Sleep function*. This value (927C0) translates to “600000” in decimal. Since the Sleep API takes arguments in milliseconds (ms), the 600000 ms get converted to 10 minutes.
- When the malware resumes from sleep, it fetches the final payload from the resource section of the PE file.

Address	Hex	ASCII
04880000	C8 33 11 25 32 0D 8A E2 53 01 C0 1E A5 CC EB 88	É3.%2..âs.A.ÿIë.
04880010	88 7E CE 2C 88 14 01 23 5E 53 96 23 80 9D A7 69	~I,...#^S.#.ÿi
04880020	66 D1 89 8D 86 DB 68 C8 A8 F2 09 22 19 60 A0 10	fñ...0kë"ö."
04880030	64 B7 56 78 5F AF 8A 42 E7 E7 1B A0 3D 72 09 97	d.Vx_.Bcc.=r..
04880040	AA DC 25 FF 21 3C EB CA FB BF 31 EE F7 F0 B3 C7	*Ûÿy!<ëËÛ¿i÷ð*Ç
04880050	4B E0 FA A0 65 96 CA 01 E7 A3 41 15 32 0B 7E F0	Kâu e.Ë.cËA.2.~ð
04880060	EF F0 79 74 A9 FA F5 A5 D3 3B 16 62 B2 B4 D8 98	iðyt@úðÿ0;.b" 0.
04880070	C7 CF 3D 18 DD D7 9B 28 FF 89 BE 3F 91 F6 2A D0	CI=.ÿx.(ÿ.%?.ö*Ð

Snippet of the protected payload stored in the memory

In the memory, the protected payload is decoded. The presence of a DOS header, in the payload bytes, confirms that the payload is in PE format and not a shellcode.

Address	Hex	ASCII
048E0000	00 00 90 00 03 00 00 00 04 00 00 00 FF FF 00 00ÿÿ..
048E0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
048E0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048E0030	00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00
048E0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..!i!Li!Th
048E0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
048E0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
048E0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$.

Decrypted payload stored in the memory

An interesting observation from this analysis, is the addition of MZ byte after the decryption process. In the above image, the initial byte is not MZ, rather the MZ byte is later added at the beginning of the payload separately. This behavior is primarily for operational security.

Address	Hex	ASCII
04D10000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
04D10010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
04D10020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04D10030	00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00
04D10040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..!i!Li!Th
04D10050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
04D10060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
04D10070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$.

Addition of the MZ byte after the decryption process

Process Hollowing

In general, process hollowing allows an attacker to change the content of a legitimate process from genuine code to malicious code before it is executed by carving out the code logic within the target process.

- After decrypting the final payload, the malware prepares for execution.
- This is done by creating a new process to deploy the extracted code and then performing process hollowing to execute the payload in the remote process. The staging code retrieves the *Rundll32.exe* location from the compromised system.

```
EAX 049F1F50 <ntdll.NtFreeVirtualMemory>
EBX 172040E4 holorui.172040E4
ECX 049F1F50 <ntdll.NtFreeVirtualMemory>
EDX 049F1F50 <ntdll.NtFreeVirtualMemory>
EBP 009FEA74 &"xØY"
ESP 009FE634 ",ØY"
ESI 000189E0 ".P"
EDI 76F92270 ".P"
```

Retrieval of the

location of *rundll32.exe*

A new process of *Rundll32.exe* is created via the *CreateProcessInternalW* API in the suspended state.

```
EAX 049F35C0 <ntdll.ZwSetContextThread>
EBX 172040E4 holorui.172040E4
ECX 049F35C0 <ntdll.ZwSetContextThread>
EDX 049F35C0 <ntdll.ZwSetContextThread>
EBP 009FEA74 &"xØY"
ESP 009FE63C ",ØY"
ESI 000189E0 ".P"
EDI 76F92270 ".P"
EIP 172166D8 holorui.172166D8
```

Creation of the new

rendll32.exe

- The malware uses the following Win32 APIs for process hollowing:
 - *ZwUnmapViewOfSection*
 - *ZwReadVirtualMemory*
 - *ZwWriteVirtualMemory*
 - *ZwGetContextThread*
 - *ZwSetContextThread*
 - *NtResumeThread*
- *ZwWriteVirtualMemory* is used to write malicious code into the target process.
- To make the thread of the new process point to newly written code, the attacker alters the entry point of the current thread via *ZwGetContextThread* and *ZwSetContextThread*.
- These functions are used to perform processor housekeeping activities on the data structure that stores the current context of the running thread. Process hollowing takes advantage of these features to make the process thread run the attacker code.

Step by Step Working of the DLL

The staging code allocates a new memory via `ZwAllocateVirtualMemory` to transfer the previously decrypted final payload.

```
EAX 049F1ED0 <ntdll.ZwAllocateVirtualMemory>
EBX 172040E4 holorui.172040E4
ECX 049F1ED0 <ntdll.ZwAllocateVirtualMemory>
EDX 049F1ED0 <ntdll.ZwAllocateVirtualMemory>
EBP 009FEA74 &"xöÿ"
ESP 009FE62C ",öÿ"
ESI 000189E0
EDI 76F92270 ".P"
EIP 17207598 holorui.17207598
```

Allocation of new

memory via `ZwAllocateVirtualMemory`

The payload is then copied to a newly created buffer.. Based on CloudSEK's testing on the extracted payload, one of the analyzed samples contained the *Raccoon stealer* as the final stage payload. However, other samples used *Cobalt Strike beacon* and *BitRAT* to compromise the target and gain unauthorized access.

Address	Hex	ASCII
048E0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
048E0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
048E0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
048E0030	00 00 00 00 00 00 00 00 00 00 00 00 08 01 00 00
048E0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..!!.Li!Th
048E0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
048E0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
048E0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$......
048E0080	71 5D 8C 4D 7F 0C 08 1F 7F 0C 08 1F 7F 0C 08 1F

Moving the payload to a newly created buffer

The staging code then injects the code into the newly created remote process i.e *Rundll32.exe*.

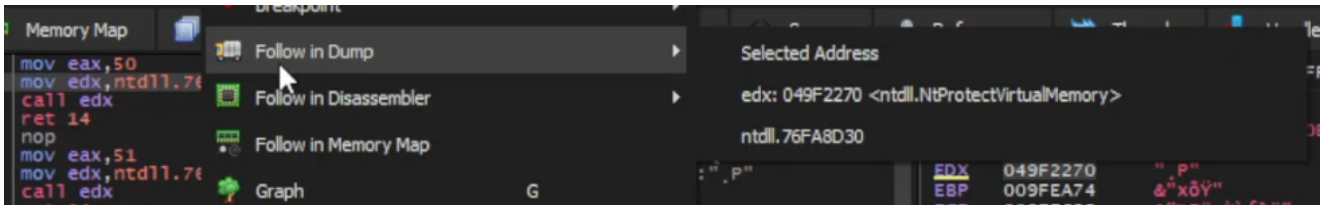
```
mov eax, 3A
mov edx, ntdll.76F...
call edx
ret 14
nop
mov eax, 3B
mov edx, ntdll.76F...
call edx
```

Selected Address
edx: 049F2110 <ntdll.ZwWriteVirtualMemory>
ntdll.76FA8D30

EDX 049F2110
EBP 009FEA74 &"xöÿ"

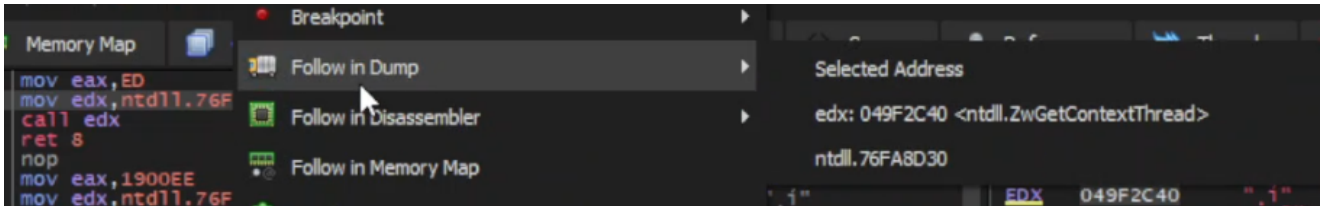
Code injections into the newly created *rendll32.exe*

Later, the memory protections are changed to appropriate ones for the execution of the residing code via `NTPProtectVirtualMemory`.



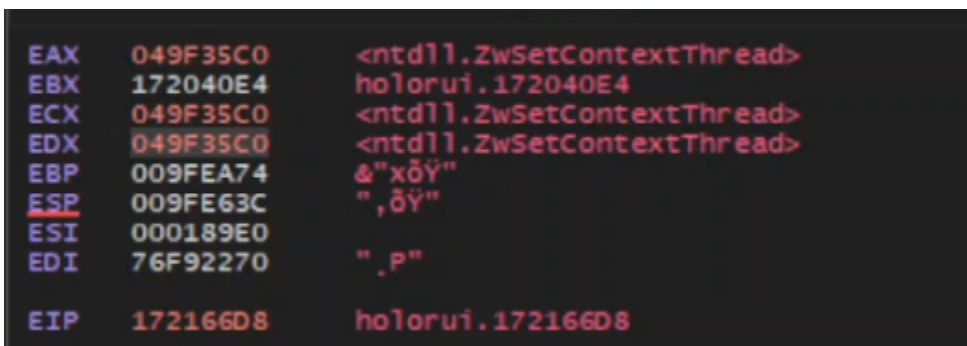
Alteration of the memory protections

The thread context is retrieved via *ZwGetContextThread* API to change the entry point of the thread to execute the payload injected into the remote process.



Addition of the MZ byte after the decryption process

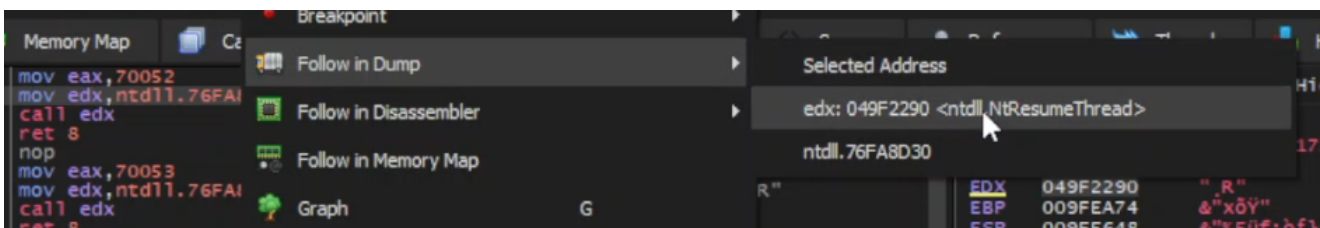
The *ZwSetContextThread* is used to modify the thread entry point to that of the newly copied PE file.



Modification of the

thread entry point to the copied PE file

At the final stage of process hollowing, the suspended thread of the *Rundll32.exe* is resumed via *NtResumeThread*. Then the *Rundll32.exe* process starts executing the malicious code hollowed into it by the malware.



Resuming the suspended thread

In the clean-up process, the staging code uses *NtFreeVirtualMemory* to release the allocated memory, which holds the payload assembly, one by one.

```

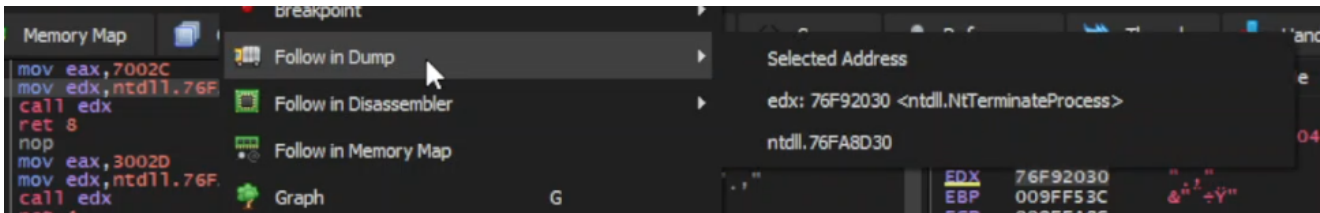
EAX 049F1F50 <ntdll.NtFreeVirtualMemory>
EBX 172040E4 holorui.172040E4
ECX 049F1F50 <ntdll.NtFreeVirtualMemory>
EDX 049F1F50 <ntdll.NtFreeVirtualMemory>
EBP 009FEA74 &"xöÿ"
ESP 009FE634 ",öÿ"
ESI 000189E0
EDI 76F92270 ".p"

```

Clean-up process

releasing the allocated memory

The current process used for staging is terminated via the `NtTerminateProcess`.



Termination of the current process

Blister Malware – Maintaining Persistence





- The Blister malware achieves persistence on the target system by creating an “Ink” file named `proamingsGames` in the `C:\Users\\AppData\Roaming\Microsoft\Windows\Start Menu\Startup` directory.
- Whenever the user logs in, `explorer.exe` executes any file in the `Startup` folder. As a result, when the user signs into the account, following the boot process, the malware runs as a child process of `explorer.exe`.

Name	Date modified	Type	Size
proamingsGames		Shortcut	1 KB

Ink file produced in the Startup directory

The target for the Ink file is set as

`C:\ProgramData\proamingsGames\proamingsGames.dll, LaunchColorCpl`. Here, the malware copies the `Rundll32.exe` as `proamingsGames.exe` and the malicious .dll (initially into `C:\ProgramData\proamingsGames` directory) is dropped in the `Temp` folder.

Name	Date modified	Type	Size
 proamingsGames.dll		Application exten...	1,114 KB
 proamingsGames		Application	61 KB

Contents of the proamingsGames.dll file

Every time that the system powers up and the user logs in, the lnk file runs a malicious `.dll` through a renamed instance of `Rundll32.exe`.

Conclusion

Given that threat actors are actively using valid code-signing certificates in Windows systems, to avoid detection by antivirus software, it is essential for network and endpoint security products to be updated with the malwares' latest Indicators of Compromise (IoCs). The latest IoCs for the Blister Malware are enumerated in [Part 1 of the technical analysis](#).

Author Details



Anandeshwar Unnikrishnan

Threat Intelligence Researcher , [CloudSEK](#)

Anandeshwar is a Threat Intelligence Researcher at CloudSEK. He is a strong advocate of offensive cybersecurity. He is fuelled by his passion for cyber threats in a global context. He dedicates much of his time on Try Hack Me/ Hack The Box/ Offensive Security Playground. He believes that “a strong mind starts with a strong body.” When he is not gymming, he finds time to nurture his passion for teaching. He also likes to travel and experience new cultures.

-
-



Hansika Saxena

Total Posts: 2

Hansika joined CloudSEK's Editorial team as a Technical Writer and is a B.Sc (Hons) student at the University of Delhi. She was previously associated with Youth India Foundation for a year.

-
-

x



Anandeshwar Unnikrishnan

Threat Intelligence Researcher , [CloudSEK](#)

Anandeshwar is a Threat Intelligence Researcher at CloudSEK. He is a strong advocate of offensive cybersecurity. He is fuelled by his passion for cyber threats in a global context. He dedicates much of his time on Try Hack Me/ Hack The Box/ Offensive Security Playground. He believes that “a strong mind starts with a strong body.” When he is not gymming, he finds time to nurture his passion for teaching. He also likes to travel and experience new cultures.

-
-

Latest Posts



- **n not malicious**
(or am I?)

