Incident Response: Analysis of recent version of BRC4

protect.airbus.com/blog/incident-response-analysis-of-recent-version-of-brc4/

Introduction

During our latest incident response case we have discovered a recent sample of Brute Ratel C4 packed with Themida. BRC4 is a powerful Command and Control (C2) tool which allows to control targeted workstations through an executable agent. The objective of Themida is to protect code against reverse engineering.

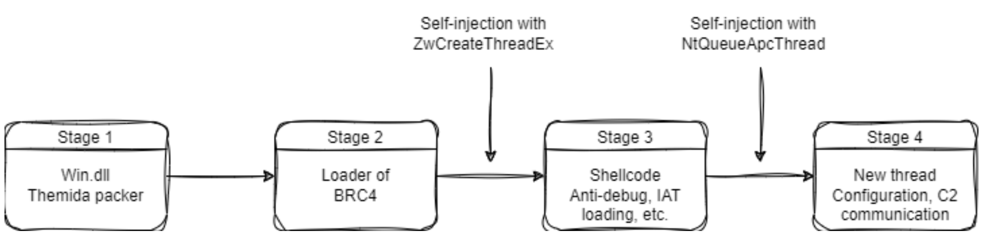
Currently, C2 tools are used by attackers as much as pentesters. So, it’s always interesting to analyse and to fully understand them in order to find a way to detect them effectively and enrich the threat hunting phase.

This sample is a DLL from an archive that has been brought to the targeted machine. It was executed with the command line present into the following event:


```
{ [-]
   Artifact: Windows.System.Powershell.PSReadline
   ClientId: [redacted]
   FlowId: F.CMDDTU65Q1STA
   Line: rundll32.exe Win.dll,main
   LineNum: 1
   OSPath: C:\Users\Administrator\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadLine\ConsoleHost_history.txt
   Stat: { [+]
   }
   Username: Administrator
   _index: artifact_Windows_System_Powershell_PSReadline
   timestamp: [redacted]
}
Show as raw text
host = [redacted]   source = velociraptor   sourcetype = artifact_windows_system_powershell_psreadline
```


The difficulties behind this sample were:
- Unpack Themida
- Defeat obfuscations and anti-debug techniques
- Understand the different stages to reach configuration and data sent

Here you’ll find a short explanation of the different stages:



This article solely focuses on obfuscation techniques, configuration extraction and how data are encrypted before they are sent to the server.

To briefly summarize, to pass the first stage, we used **ScyllaHide** plug-in on xDBG and we jumped in a specific area with read/execution rights at the time the DLL was loaded. In this area, we found symbols that allowed us to find the loader of BRC4.

The second stage is just a loader of shellcode where **ZwAllocateVirtualMemory**, **ZwProtectVirtualMemory**, **ZwCreateThreadEx** and **NtWaitForSingleObject** functions are used for self-injection. We save the shellcode with system informer to get a new starting point for the analysis.

The third stage focuses on the first part of the shellcode with obfuscations and anti-debug techniques. It introduces the last stage by self-injection with NtQueueApcThread.

The last stage is the most interesting part of the shellcode because it focuses on configuration and communication.

In the following article, I will use a first part to describe what obfuscation techniques were used in stages 2, 3 and 4 and unpacking process of Themida. I will then describe how stage 4 retrieves the configuration, uses it to cypher outgoing data, and how we can automate retrieval of this configuration.

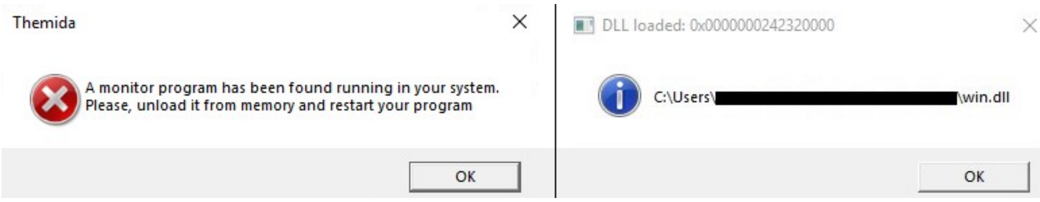
Unpack of Themida

For the unpacking part we used ScyllaHide plug-in on x64DBG with Themida x86/x64 profile.
We found two different results for two types of execution: normal execution on the left and
execution with ScyllaHide on the right:

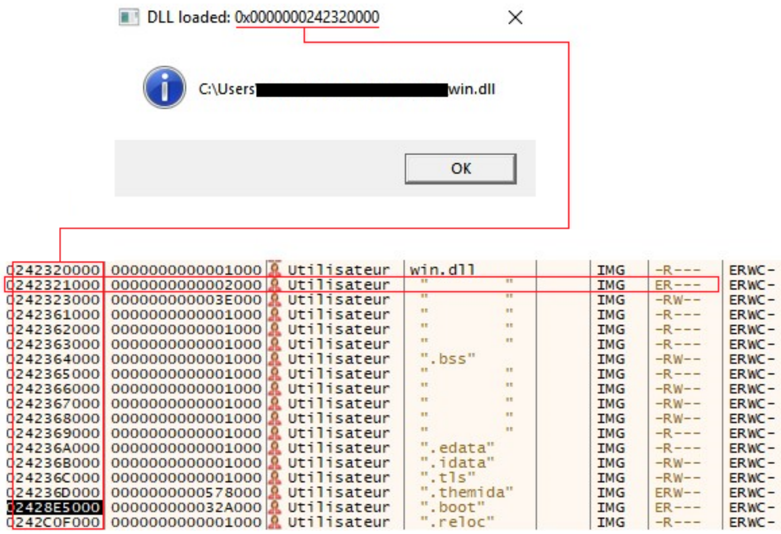
Name	Win.dll
SHA-256	4400750cbc597b7e0cec813dcaf66d00e83955a034591a5a6ba40547a045721b
File	type PE64
Packer	Themida 3.x

For the unpacking part we used ScyllaHide plug-in on x64DBG with Themida x86/x64 profile.

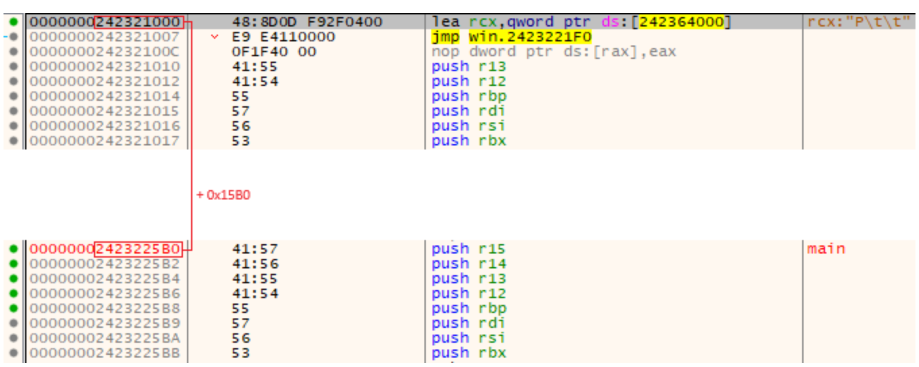
We found two different results for two types of execution: normal execution on the left and execution with ScyllaHide on the right:



After the execution with ScyllaHide plugin, we found a memory area with execution right and we jumped on it:



During the analysis of this memory area, we finally found a main function, this function is our second stage:



Obfuscation & Anti-debug

During our analysis we found obfuscations based on scraping, PEB parsing and API hashing on stage 2, 3 and 4.

Here, a part of code of stage 3:

```
:000001D1CF64CBCA loc_1D1CF64CBCA:
:000001D1CF64CBCA cmp     word ptr [r8], 5A4Dh
:000001D1CF64CBD0 jnz     short loc_1D1CF64CBC6
:000001D1CF64CBD2 movsxd  rax, dword ptr [r8+3Ch]
:000001D1CF64CBD6 lea     rdx, [rax-40h]                PE scrapper
:000001D1CF64CBDA cmp     rdx, 3BFh
:000001D1CF64CBE1 ja      short loc_1D1CF64CBC6
:000001D1CF64CBE3 cmp     dword ptr [r8+rax], 4550h
:000001D1CF64CBEB jnz     short loc_1D1CF64CBC6
```

```
000001D1CF64CBED mov     edx, 82BB0EE0h
000001D1CF64CBF2 mov     rcx, r12
000001D1CF64CBF5 mov     [rsp+280h+var_140], r8
000001D1CF64CBFD call    sub_1D1CF64C1E6          PEB parsing + API Hashing func
000001D1CF64CC02 mov     r8d, 1
000001D1CF64CC08 xor     edx, edx
000001D1CF64CC0A mov     rcx, rax
000001D1CF64CC0D mov     [rsp+280h+var_D0], rax
000001D1CF64CC15 call    sub_1D1CF64C946          ID Resolver for syscall
000001D1CF64CC1A mov     rcx, [rsp+280h+var_D0]
000001D1CF64CC22 mov     [rsp+280h+var_A8], ax
000001D1CF64CC2A call    sub_1D1CF64C4B6          Syscall Builder func
000001D1CF64CC2F mov     r8, [rsp+280h+var_140]
000001D1CF64CC37 mov     edx, 14E66623h
000001D1CF64CC3C mov     rcx, r12
000001D1CF64CC3F mov     [rsp+280h+var_90], rax
000001D1CF64CC47 call    sub_1D1CF64C1E6          PEB parsing + API Hashing func
000001D1CF64CC4C mov     r8d, 1
000001D1CF64CC52 xor     edx, edx
000001D1CF64CC54 mov     rcx, rax
000001D1CF64CC57 mov     [rsp+280h+var_C0], rax
000001D1CF64CC5F call    sub_1D1CF64C946          ID Resolver for syscall
000001D1CF64CC64 mov     rcx, [rsp+280h+var_C0]
000001D1CF64CC6C mov     [rsp+280h+var_A4], ax
000001D1CF64CC74 call    sub_1D1CF64C4B6          Syscall Builder func
```

We can find multiple functions, their role is:

- Introducing anti-debug techniques
- Load modules with PEB Parsing
- Load pointers of functions with PEB parsing and API hashing
- Build syscall routine
- Resolve syscall ID with scraping

Anti-debug

An anti-debug technique involving PEB parsing is used to compare the value at **PEB+0xbc** with **0x70**. This code is encountered two times in the stage 3, it's not evident to spot it, so we must analyse the code step by step.

```
000001D1CF64CBA0 loc_1D1CF64CBA0:
000001D1CF64CBA0 mov     rdx, gs:60h
000001D1CF64CBA9 movzx   eax, byte ptr [rdx+0BCh]
000001D1CF64CBB0 and     eax, 70h
000001D1CF64CBB3 cmp     al, 70h ; 'p'
000001D1CF64CBB5 jz      loc_1D1CF64CE9D
000001D1CF64CBBB mov     r8, [rdx+18h]
000001D1CF64CBBF jmp     short loc_1D1CF64CBCA
```

Flag	Value
FLG_HEAP_ENABLE_TAIL_CHECK	0x10
FLG_HEAP_ENABLE_FREE_CHECK	0x20
FLG_HEAP_VALIDATE_PARAMETERS	0x40
Total	0x70

Load modules with PEB Parsing

The pointer to the base address of each module is found with PEB parsing. In this case the program makes a loop in the **_PEB_LDR_DATA** structure to scrape these bytes: **0x5A4D**. This technique is used to avoid calling direct functions such as **LoadLibraryA** which allows to load DLL.

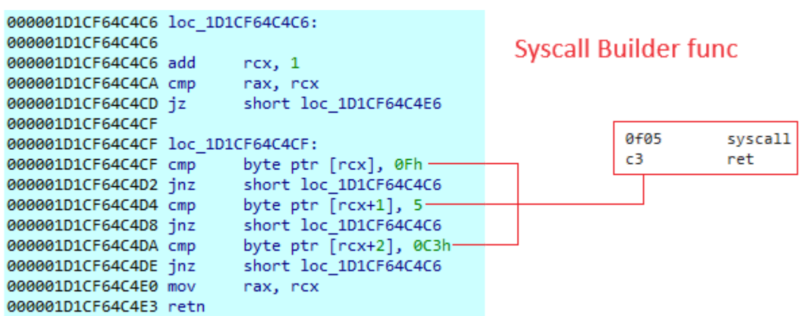
Load pointers of functions with PEB parsing and API hashing

To resolve the pointers of the functions, the program parses the PEB structure and then makes a loop in the name pointer table in **IMAGE_EXPORT_DIRECTORY** structure. A call on the hashing function is operated for each name in the table to find the correct function for the requested hash. Once the correct function is found, its pointer is obtained using the address table. You'll find an article describing the process in a more detailed way here.

Build syscall routine

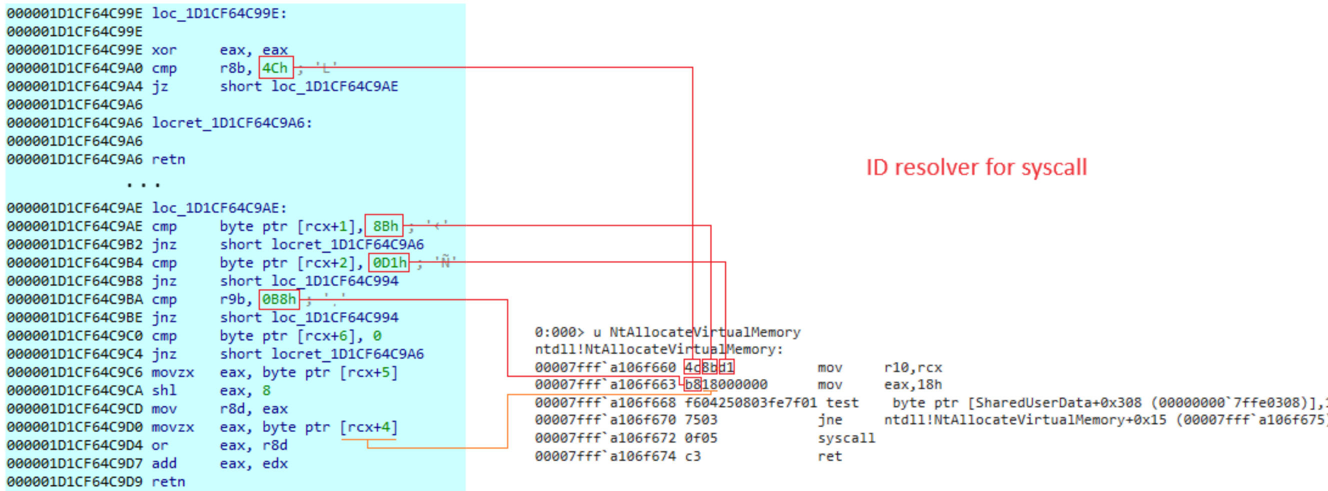
To be stealthier than its previous version, the program builds its own function to make a syscall. In an older version, there were obfuscation techniques for the loading function and ID resolution, but at the end there was a direct syscall advising us for an incoming self-injection.

In the capture below, the program builds a pointer to a custom code section to execute a syscall.



Resolve syscall ID with scraping

The code doesn't use a direct syscall ID to be able to target enough workstations regardless of their version. The ID of a syscall depends on the version and build number of the Operating System: with this technique, it's not necessary to obtain the OS version of the targeted workstations. All ID are presented on j00ru website. On the screen below, we can see the specific section which resolved a syscall ID. The code scrapes a specific sequence of bytes to find out the correct position of the ID. This process depends on the function previously loaded with PEB parsing & API hashing:



For a better understanding, on the screen above we have the function used to find syscall ID thanks to bytes sequence on the left. On the right, we have the code of **NtAllocateVirtualMemory** which allows us to understand why these following bytes are targeted: 0x4C, 0x8B, 0xD1 and 0xB8. The bytes 0x4C, 0x8B and 0xD1 are respectively:

```
MOV R10, RCX
```

The byte just after 0xB8 is the syscall ID, in normal execution this value is moved into EAX like that:

```
MOV EAX, [SYSCALL ID]
```

To resume this part, the malware uses these techniques to be stealthier that allows it to evade detection against security software.

These techniques allow to the malware to:

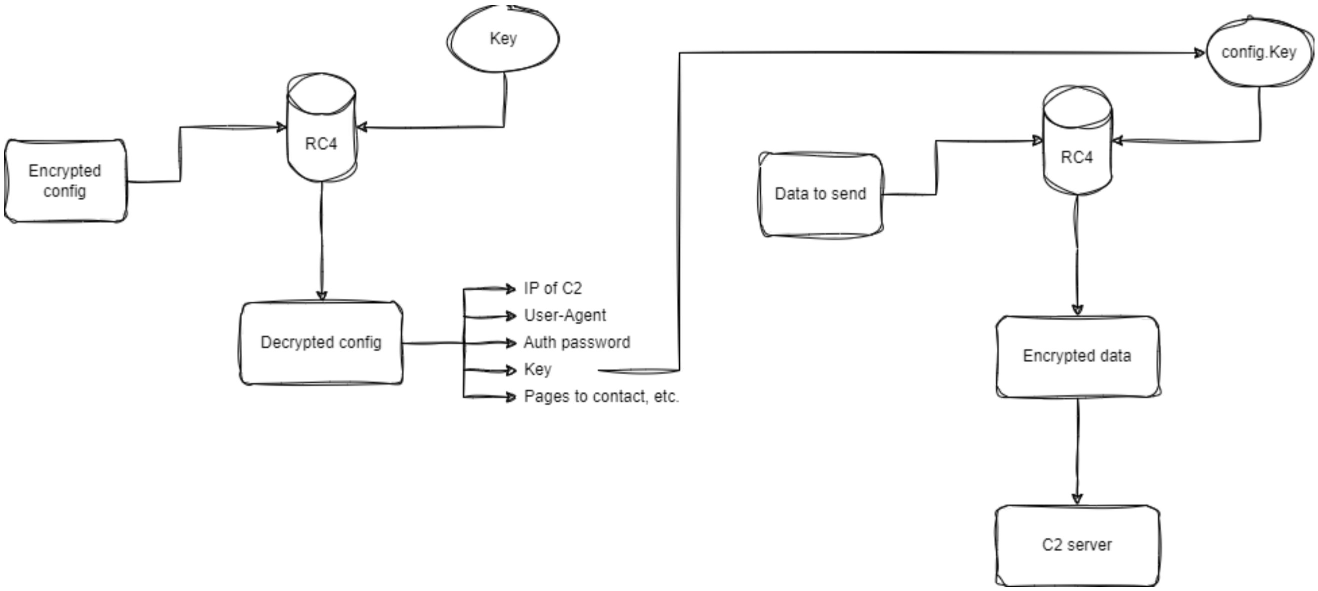
- Hide functions into Import Address Table (IAT) to evade detection
- Counter dynamic analysis by stopping the program
- Counter userland detection by IAT hooking by using direct syscall

Last stage

The last stage focuses on configuration extraction and encryption of data before they are sent to the C2 server. In addition, we can find obfuscation functions which loads DLL and function with symbols to communicate with C2 server. In our case, the BRC4 shellcode uses **ws2_32.dll** library, the **HttpSendRequest** function to send data and the **InternetReadFile** function to read data.

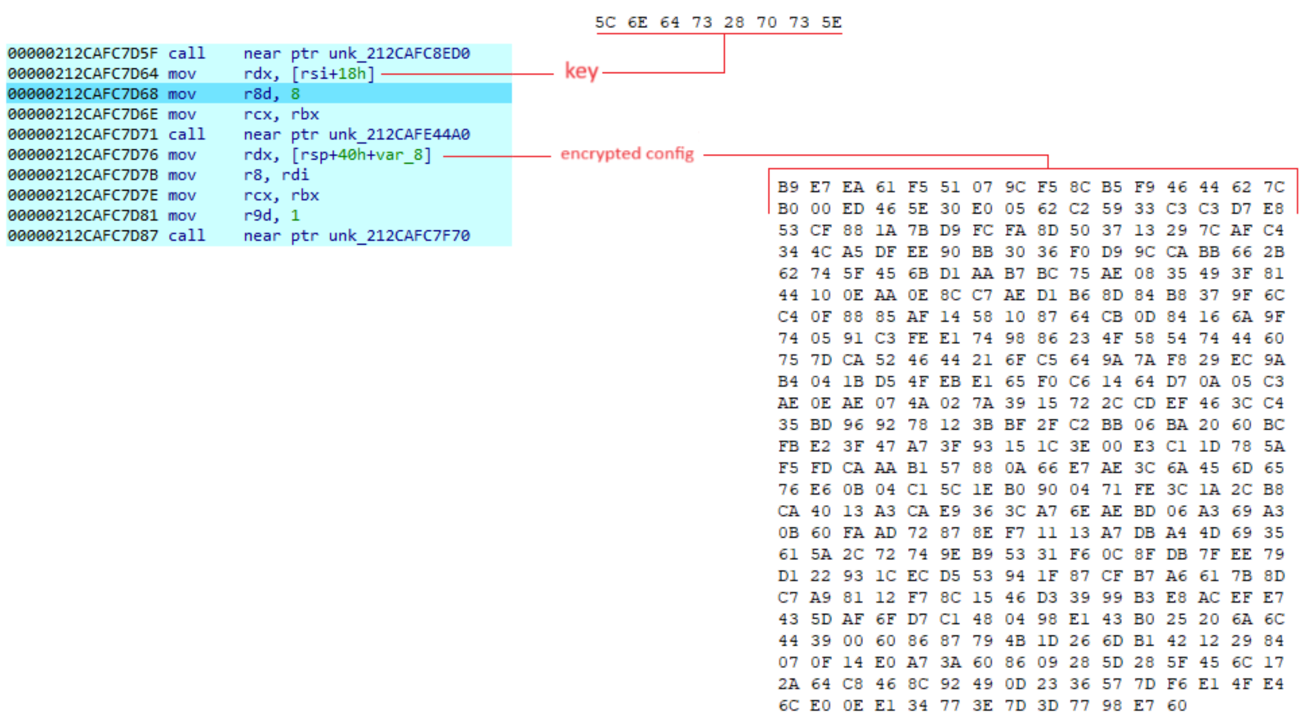
During this stage, we can retrieve all BRC4 principal functions using the **NOP** value (0x90), because each function is separated from the other by **NOP** instructions. The number of **NOP** depends on the size of the functions, since the purpose of the **NOP** instructions is to correctly align the stack.

Before continuing, here a diagram explaining the process between the configuration and the encryption of data:



Configuration extraction

Like its previous version, the configuration is encrypted in RC4. The key and the encrypted configuration are found at this precise moment during the last stage:



Shortly afterward, we can retrieve the configuration in clear text. Because it takes a lot of time to reverse all the code until this moment, we prepared a configuration extractor based on specific patterns in the program's memory. The conditions to fulfil to be able to use our extractor program are:

- Get the BRC4 shellcode in the loader stage (real first stage, because Themida packer is an added stage by our adversary in this instance).
- Get this shellcode loader here for our configuration extraction process.
- Get the configuration extractor on Airbus Protect GitHub.

The configuration extractor is a python program which uses the Ctypes library to execute our shellcode through a loader. The handle of the new process is used to obtain memory areas with **VirtualQueryEx**. Here, we focus on memory areas with these specific conditions:

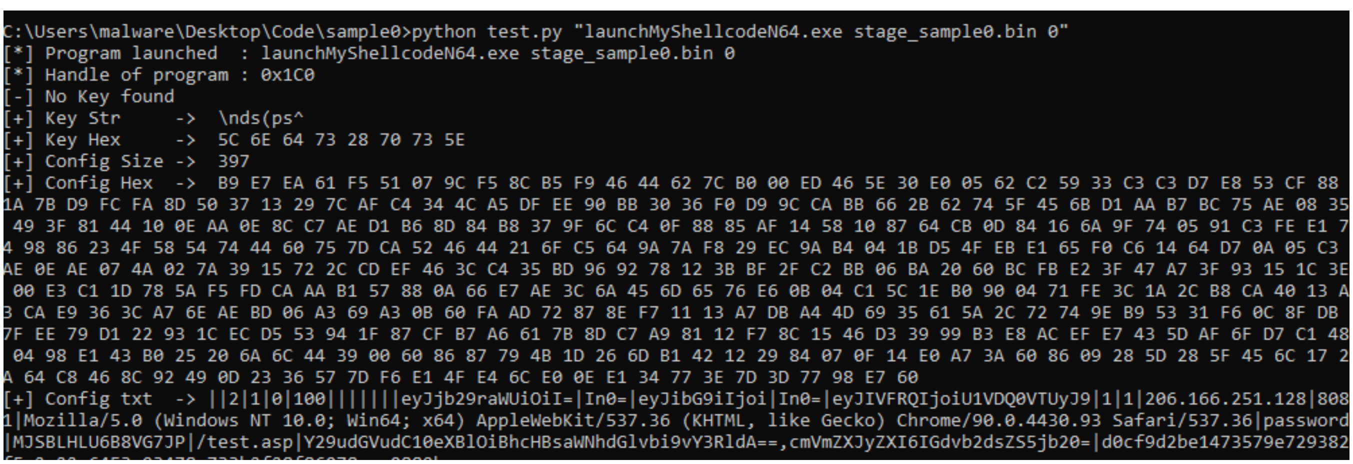
```
if mbi.State == MEM_COMMIT and (
 mbi.Protect == PAGE_READWRITE or
 mbi.Protect == PAGE_READONLY ):
```

Each matched memory area is saved in the same dump file. At the end of this function, the dump is used to extract the key and the configuration by using these regexes:

```
regex_sequences_forKey = [
 r"(00){16}([1-9A-Fa-f]{1}[0-9A-Fa-f]{1}){8}([0-9A-Faf]{2}){8}(00){8}(..0001......)(00..)",
 r"(00){16}([1-9A-Fa-f]{1}[0-9A-Fa-f]{1}){8}([0-9A-Fa-f]{2}){8}(00){8}([0-9A-Faf]{2}){6}(0001)",
 r"(00){16}([1-9A-Fa-f]{1}[0-9A-Fa-f]{1}){8}([0-9A-Fa-f]{2}){8}(00){8}([0-9A-Faf]{2}){6}(0010)"
 ]
regex_sequences_forConfig = r"(4883e4f04831c0505468)"
```

We performed some tests on samples discovered on Virus Total with Yara rules created for this specific version of BRC4. These Yara rules are available in the Yara section of this article. Here, the results of our extractor on five samples (each test is operated on the shellcode extracted from a DLL file):

Case sample – 4400750cbc597b7e0cec813dcaf66d00e83955a034591a5a6ba40547a045721b



sample 1 – 780b2b715aa33e8910479a671469ad27cc88a7ed513b83e43cf7a6a16f613013

```
C:\Users\malware\Desktop\Code\sample1>python test.py "launchMyShellcodeN64.exe stage_sample1.bin 0"
[*] Program launched  : launchMyShellcodeN64.exe stage_sample1.bin 0
[*] Handle of program : 0x1B8
[+] Key Str     ->  |\l^d\{|
[+] Key Hex     ->  7C 5C 6C 5E 64 5C 7B 7C
[+] Config Size ->  330
[+] Config Hex  ->  DE B3 47 CA C7 D1 7F 62 0A 43 31 F6 1F AE F8 CF B7 45 E2 8B E3 5F 5B 72 6D 4A FB 90 9B 25 3A 80 39 A9 3A
D5 02 CA 43 88 92 C0 D4 CA 83 8A E1 F7 24 E6 3C 9F 2C B7 D7 C2 69 0D B2 12 CF 2F 59 3B 05 B2 89 70 72 F2 06 E0 81 5A 39 B0 FE
 AD 0D 98 73 84 56 C1 1E A0 39 D1 00 7F A6 53 88 26 73 F1 03 A4 F1 3B CB A5 3A DE 95 D8 63 86 BB D6 DD E0 11 4A 22 44 14 77 9
B 25 DA CD 25 09 91 23 79 B3 F1 75 56 B1 D9 15 AA 03 F7 74 57 C0 33 FD D5 4D 44 63 06 C3 B4 6E E4 1F E1 3B C0 EF 91 46 B4 FE
8F 44 EB 1D D5 A0 AD 41 95 55 FA B2 86 5A 4E C7 98 29 4B A3 9C C6 AD 51 2F A0 8A C7 A2 DA BD 81 8E 92 01 44 F8 AF 9C 73 75 31
 E0 38 B2 2B 12 03 3D A2 0F 2B D0 AB DD 1A ED 65 97 16 A6 AA 76 D1 7B AB E9 58 7A A1 04 28 8A 51 16 82 D2 92 5B 6E 0C 70 F3 B
0 2B 92 DC 30 2F 3D 15 A9 7D 23 F3 AE 59 66 D7 16 8D D7 9D B9 84 E6 58 F0 98 6A 7A F4 9A BA 80 67 03 75 05 FA 84 F1 7C 48 29
3A BF 9A E8 5A 8C 5A 2A 76 75 AD 0F AB 83 46 83 0B D4 15 15 12 EC 63 A4 10 E3 6C AB D9 AB A8 EA 48 0F D3 D7 70 18 A2 4E C5 AC
 CC FD 41
[+] Config txt  -> ||0|60|40|100||||||||||||0|0|213.215.163.51|8080|Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/188
.16 (XTML, like VBScript) Chrome/89.6.4167.11 Safari/123.11|DVQDPV4QA5JM21UQ|78PCI7F6T2V68GSU|/update.php,/update_manager.htm
l||abf002b4ecb7c52a0d609c63754f3a5be8e3094ba638580e8f8a5b51fdbbd3f194b47d5205206bdfb0f267fbea559891
```

sample 2 – 04b47b5492f5b2086e4a6b3f2bef73eb15a51140a86bcd05417d00bf6875ffb6

```
C:\Users\malware\Desktop\Code\sample2>python test.py "launchMyShellcodeN64.exe stage_sample2.bin 0"
[*] Program launched  : launchMyShellcodeN64.exe stage_sample2.bin 0
[*] Handle of program : 0x1B8
[+] Key Str     ->  a::^>q(|
[+] Key Hex     ->  61 3A 3A 5E 3E 71 28 7C
[+] Config Size ->  294
[+] Config Hex  ->  34 4E 06 9F 06 4F E4 DE 6A 7F 76 3F FA 49 4C 7D 33 69 E3 BE 26 A9 8C D6 26 4D CF A9 0F FF 1C 8B C6 E
1 62 31 92 59 BC CF 3A 93 65 12 78 47 91 DF 4D 03 0B 4E E2 58 23 92 82 8A E0 05 D4 CA BC 4E 85 F1 AD 8B 78 36 B2 82 18 7
1 C6 71 89 E4 E0 EE 75 7B 06 ED F7 80 81 55 63 D2 42 15 07 1E B3 57 7B 29 BF 57 D9 47 C5 A4 4B EF C7 9B EA BD 69 4E 9B 4
E 92 D9 7D 52 EF 5D 5C 93 34 89 4D FA F2 D2 09 DB EF 30 A1 F4 DC 0C BF 5C 19 B8 62 97 15 0E 13 E5 C2 3A A5 9E 45 A3 30 6
4 CA 0D 6B 9A D1 F9 4B D2 71 E7 02 4D 0F A6 35 77 11 13 04 41 2B 91 60 6C 15 A7 0E 94 3B 36 81 93 EC 19 C3 36 D6 39 0D 8
9 A8 3B 64 15 F4 57 EE FA 02 36 5B C9 8D C5 AB 09 7E 99 20 AB CA D9 31 2B 62 80 39 CA 78 9A 6E A6 12 2C A2 96 90 AB 39 7
5 47 7D F4 DC B2 CC 3D 71 D4 67 F9 1C CF 4B 0D DC A7 FD 73 40 F5 31 DB F9 FC C4 7C D6 BB 74 DB F7 22 A5 B5 E8 75 A1 A8 A
6 FC 26 64 F1 88 FE 8D 18 F7 D7 C0 8A 59 FB 25 37 F4 B5 AB 29
[+] Config txt  -> ||0|60|40|100||||||||||||0|0|20.218.134.226|80|Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/90.0.4430.93 Safari/537.36|hinterGr33n!|OE37DKUM0ABD7CUF|/admin.php,/ jscript.css,/ wp
-config||6e714f616a878d59ac0fac1f30006ce9f43a25e26cc38326b3bbbeb55f54810d
```

sample 3 – 9ec67f1914603e729a3b6bafe3a96cdc660717ca7dfb457290f68fc56dd0a5e2

```
C:\Users\malware\Desktop\Code\sample3>python test.py "launchMyShellcodeN64.exe stage_sample3.bin 0"
[*] Program launched  : launchMyShellcodeN64.exe stage_sample3.bin 0
[*] Handle of program : 0x1B8
[-] No Key found
[+] Key Str     ->  $vseninq
[+] Key Hex     ->  24 76 73 65 6E 69 6E 71
[+] Config Size ->  264
[+] Config Hex  ->  A2 27 69 98 82 67 3B 2C 74 6E 13 5E DA F8 1D 2C 28 92 66 AD 43 D7 3A 60 97 3B 4A 68 F4 51 68 34 0D F
1 CE 8D 48 DD B9 AD 95 BF 9C 5C 54 9F 92 FB 9E 66 76 13 BD 77 05 35 1D 11 58 AC F5 49 94 EF 8A 07 BD 01 7E 40 54 70 77 7
B D8 0F 53 E1 7D 43 19 B3 70 0F 81 98 95 93 69 47 59 F8 FB 5D CF BA 8F 24 72 CC 5B BA 7A 55 73 1F 8D 69 87 2F 92 2A 04 D
2 E4 57 87 81 EF 26 65 54 8D 85 D1 4D FA CC 7E 0A CA F5 20 3D E7 73 AB 29 6C F7 E8 B1 A4 1A FD 3E 99 36 5F AC 5C F4 CE 7
8 49 CB 44 91 12 F6 4E 6E 28 54 26 3C 75 98 6D E5 3C A4 0C 63 95 87 49 F3 7D 85 0D AC 37 60 7D DF 1B 09 E0 40 9F A4 C0 C
C 60 1F 48 2C 77 7F 83 9F 38 A3 F8 C1 FB 1D FA 70 16 79 29 AF 31 98 EB 17 C7 A5 E6 C0 2E 0C 72 E6 25 17 9D 89 38 17 72 3
E F0 35 84 23 A9 CB C0 B8 25 98 61 C7 FC F0 18 35 70 2A 45 9E 33 D2 56 02 54 27 1E 3E 4E 5C
[+] Config txt  -> ||0|60|40|100||||||||||||0|1|10.39.95.19|443|Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/90.0.4430.93 Safari/537.36|password|MKBKR2FHIIEV7U0L|/content.php||7f1b365ff9ae85a539387
2621b118620599cfdd66b2a38d63b22ec3e6f07169b
```

sample 4 – bd32cbb6c08eff7fc6aa0bfe2fd81ec467f70d9b726015859da39744271bbcb0

```
C:\Users\malware\Desktop\Code\sample4>python test.py "launchMyShellcodeN64.exe stage_sample4.bin 0"
[*] Program launched  : launchMyShellcodeN64.exe stage_sample4.bin 0
[*] Handle of program : 0x1B8
[-] No Key found
[+] Key Str     ->  :<l$>f#|
[+] Key Hex     ->  3A 3C 6C 24 3E 66 23 7C
[+] Config Size ->  276
[+] Config Hex  ->  52 0B 85 4E F2 D5 46 AA BB BB 1A 68 D7 34 66 ED 4A 0B 48 70 32 77 AE 67 7E 4E BA EB F7 FF D0 D9 FC 3
7 65 21 BE 20 D7 99 64 47 DB 5E 8B F5 A5 3F A3 74 72 F3 F2 24 50 D4 06 15 E1 25 62 EC 9F 8A F7 B4 B3 AB C7 FB 0F 47 38 1
E 76 64 AB 47 C8 53 9C BD 9B AE 6C 17 1A BB 6E 86 82 0E A1 73 65 9C C8 4E 27 31 81 7A 8D 86 E0 AC A4 2E CC 20 A7 59 02 1
1 3E 0E D0 97 5A 02 9D 89 2B C1 2F E1 78 93 47 20 AB 62 B4 F3 03 B1 73 7C FB 2E 47 7E 28 93 8F 1E D9 FF 24 AC 9F 30 C7 B
3 0D 5D 91 83 50 4B 2E 46 F7 C3 7E C6 7D CF 71 49 6E 12 38 73 95 32 01 45 05 AF AA CE D8 C6 BC 64 04 68 2A 7C 19 20 C5 E
0 05 40 89 C3 0A 43 6F 9D 4B D4 A4 65 24 8E EC 99 B4 7D 8E 39 F7 08 DD 30 1B 61 40 4D DE 93 D8 4D E7 0A 61 09 A4 B6 60 7
1 5B A5 DA AF C7 7D 8E 24 5E 51 25 22 8D 1F 5B 6B F7 DF A6 20 6C 9A EE 11 D6 F3 97 15 0B 61 1E 12 69 CB 8A E8 C5 3D 80 0
F 06 EB
[+] Config txt  -> ||0|60|40|100|||||||||||0|1|179.43.144.250|1443|Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKi
t/537.36 (KHTML, like Gecko) Chrome/90.0.4430.93 Safari/537.36|JHJJ7EF2I61KRU9T|2LUR7P617USQHBSJ|/content.php||d0cf9d2be
1473579e729382f5c2e22c6ea2a429df325cdbe6d87d971ff6b5e0a
```

Encryption of data

Before sending data to the C2 server, the program uses RC4 to encrypt them. The first request is just a simple JSON containing data about the victim, like a profile. The first request before encryption looks like this:


```
{
 "cds": {
"auth":"[C2_PASSWORD]"
 },
 "mtdt": {
 "h_name":"[VICTIM_MACHINE_NAME]",
 "wver":"x64/100",
 "ip":"[VICTIM_MACHINE_IP], 0.0.0.0, 0.0.0.0",
 "arch":"x64",
 "bld":"17763",
 "p_name":"[PATH_OF_EXECUTABLE_IN_BASE64]",
 "uid":"[VICTIM_USERNAME]",
 "pid":"6992",
 "tid":"5448"
 }
}
```


Explanation of data fields:

- **C2_PASSWORD**: password for C2 authentication
- **VICTIME_MACHINE_NAME**: the name of the victim machine
- **VICTIME_MACHINE_IP**: the IP address of the victim machine
- **PATH_OF_EXECUTABLE_IN_BASE64**: the path to the base64-encoded executable
- **VICTIM_USERNAME**: the session username of the victim machine

RC4 encryption is operated by **SystemFunction033** from **cryptsp.dll**:

```
                                                        key
000001E7D1094C07 mov     [rsp+48h+var_20], rcx
000001E7D1094C0C lea     rdx, [rsp+48h+var_28]
000001E7D1094C11 lea     rcx, [rsp+48h+var_18]
000001E7D1094C16 mov     [rsp+48h+var_24], r8d
000001E7D1094C1B mov     [rsp+48h+var_28], r8d          key size
000001E7D1094C20 mov     [rsp+48h+var_10], rax          data to encrypt
000001E7D1094C25 mov     [rsp+48h+var_14], r9d
000001E7D1094C2A mov     [rsp+48h+var_18], r9d          size of data to encrypt
000001E7D1094C2F call    cs:qword_1E7D10D01D8           cryptsp_SystemFunction033
```

This function is an alias of **SystemFunction032** because both point to the same relative address:

Based on ReactOS documentation, this function operates an RC4 encryption routine:

```
NTSTATUS
WINAPI SystemFunction032(struct ustring *data, const struct ustring *key)
{
 RC4_CONTEXT a4i;
 rc4_init(&a4i, key->Buffer, key->Length);
 rc4_crypt(&a4i, data->Buffer, data->Length);
 return STATUS_SUCCESS;
}
```

To remind you, here the decrypted configuration for our case:

```
[+] Config txt ->
||2|1|0|100||||||eyJjb29raWUiOiI=|In0=|eyJibG9iIjoi|In0=|eyJIVFRQIjoiU1VDQ0VTUyJ9|1|1|
206.166.251.128|8081|Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/90.0.4430.93
Safari/537.36|password|MJSBLHLU6B8VG7JP|/test.asp|Y29udGVudC10eXBlOiBhcHBsaWNhdGlvbi9vY
3RldA==,cmVmZXJyZXI6IGdvb2dsZS5jb20=|d0cf9d2be1473579e729382f5c2e22c6453a93478a733b2f28
f86078cec0889b
```

In this table you will find the elements that interest us and that will allow us to understand in a little more detail how the data will be processed:

C2 IP	206.166.251[.]128
C2 PORT	8081
HEADER PARAMETER 1 (user-agent)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.93 Safari/537.36
HEADER PARAMETER 2 (content-type)	content-type: application/octet (Y29udGVudC10eXBlOiBhcHBsaWNhdGlvbi9vY3RldA==)
HEADER PARAMETER 3 (referrer)	referrer: google.com (cmVmZXJyZXI6IGdvb2dsZS5jb20=)
PASSWORD FOR AUTHENTICATION	password
PASSWORD TO ENCRYPT DATA TO SEND	MJSBLHLU6B8VG7JP
PAGE TO COMMUNICATE	/test.asp

In this context, we know that the first request will be send to **http[:]//206.166.251[.]128:8081/test.asp** with the header parameters provided in the table above. Before sending to the C2 server, the program will encrypt the data with the password
(MJSBLHLU6B8VG7JP) in the table and it will make this request with encrypted data:

```
{
    "cookie": "4c36726d0c276eab2057f7c387fd9d75c9d2d6475bb74fdf64731601047bf8df0e01a2be604191
               543187e5e60e1b2b3c9c7cc06299822f7113de4e292be62a4d5627484872ee2b9d2bb257d3bdd3
               9cda8f1c0ea9fd0c397aaf628247d53bbfe11dac570c41a50a81d7559257c96713ec3d3ada117a
               3736e0170b39c2e1b4b338623e554da091d982896a32befb99c7756356cb73cf7f705c0ef2f3c3
               71185ca560e8b881978d348e0b3962b2043ff84056336c1196a76745dbd7dbb314ba062c367a2a
               2dd433363b43263fbc21909cc1fe213e2e2f6add7281cf31869f32a11e4b81b3f05ac11f5a048f
               4f1525d63392a6514ee00605c250d9ad08460bca6c6957ac8731859cf88962228bff73a90f5434
               42f7716476560f843e167baaaa85ea48da3a3dd3fbb924473b07c30bef2202fc20fa544f644779
               5df5fe2f3660e3917d3332ef64f10c0148f64aab62574ced0db22c9bf2da33ceab2e60ec5435e6
               42ce1dfce98f23eb5595783e8ae155a87e288d5247f536a45ffded"
}
```


Conclusion

Our in-depth analysis of Brute Ratel allows us to highlight the complexity behind all techniques seen in this article.

We explored various obfuscation techniques that complicate analysis for experts and detection by security software. We also covered the importance of configuration, a key element of the program. This not only enables the retrieval of agent configuration data, including the IP address and port of the command and control (C2) server, but also the passwords used for authentication and encryption of data to be communicated to the server.

Finally, we've set up a configuration extractor that allows us to quickly retrieve the agent's key elements.

Year after year, malware grows in complexity, and we must continue our research to help the community to detect effectively. We hope that the findings and tools presented in our research will help you.

Detection

Yara

```
import "pe"
rule stage_Loader {
 meta:
 author = "Adams KONE"
 company = "Airbus PROTECT"
 sharing = "TLP:CLEAR"
 category = "MALWARE"
 description = "Loader’s stage"

 strings:
 //Obfuscation technique
 //API hashing function
 $HashingFunction = {
 31 D2 0F BE 01 84 C0 74 14 01 D0 48 FF C1 69 C0
 01 04 00 00 89 C2 C1 EA 06 31 C2 EB E5 8D 04 D2
 89 C2 C1 EA 0B 31 D0 69 C0 01 80 00 00 C3
 }
 //Obfuscation technique
 //Function to resolve syscall ID
 $getIdForSyscall = {
 80 79 FF CC 74 58 45 85 C0 75 04 48 83 E9 20 44
 8A 09 41 80 F9 E9 74 0A 44 8A 41 03 41 80 F8 E9
 75 07 FF C2 45 31 C0 EB D7 31 C0 41 80 F9 4C 75
 2F 80 79 01 8B 75 29 80 79 02 D1 75 21 41 80 F8
 B8 75 1B 80 79 06 00 75 17 0F B6 41 05 C1 E0 08
 41 89 C0 0F B6 41 04 44 09 C0 01 D0 EB 02 31 C0
 C3
 }
 //Obfuscation technique
 //Function to forge a pointer to syscall, ret instructions.
 $getAddrToJumpToSyscallAndRetInstruction = {
 48 89 C8 48 8D 51 14 80 38 0F 75 0C 80 78 01 05
 75 06 80 78 02 C3 74 0A 48 FF C0 48 39 C2 75 E7
 31 C0 C3
 }
 //Obfuscation technique
 //Hash of functions
 $ZwProtectVirtualMemory = { E0 0E BB 82 }
 $ZwAllocateVirtualMemory = { BF 06 3A E3 }
 $NtWaitForSingleObject = { 26 6E C2 E2 }
 $NtCreateThreadEx = { AA 5D F1 E5 }
 condition:
 (uint16(0) == 0x5a4d and uint16(uint16(0x3c)) == 0x4550) and all of them
}

rule Stage_BRC4_Part1 {
 meta:
 author = "Adams KONE"
 company = "Airbus PROTECT"
 sharing = "TLP:CLEAR"
 category = "MALWARE"
 description = "First part of the shellcode’s stage"

 strings:
 //Obfuscation technique
 //API hashing function
 $HashingFunction = {
 0F BE 01 84 C0 74 39 31 D2 0F 1F 80 00 00 00 00
 01 D0 48 83 C1 01 89 C2 C1 E2 0A 01 D0 89 C2 C1
 EA 06 31 C2 0F BE 01 84 C0 75 E5 8D 14 D2 89 D0
 C1 E8 0B 31 D0 89 C2 C1 E2 0F 01 D0 C3 0F 1F 00
 31 C0 C3
 }
 //Obfuscation technique
 //Operation to check if the program is running in a debugger
 $AntiDebugViaPEBParsing = {
 65 48 8B 14 25 60 00 00
 00 0F B6 82 BC 00 00 00
 83 E0 70 3C 70
 }
 condition:
 all of them
}
rule Stage_BRC4_Part2 {
 meta:
 author = "Adams KONE, Airbus PROTECT"
 company = "Airbus PROTECT"
 sharing = "TLP:CLEAR"
```

```
 category = "MALWARE"
 description = "Stage of second part of shellcode"

 strings:
 //Obfuscation technique
 //API hashing function
 $HashingFunction = {
 31 D2 0F BE 01 84 C0 74 14 01 D0 48 FF C1 69 C0
 01 04 00 00 89 C2 C1 EA 06 31 C2 EB E5 8D 04 D2
 89 C2 C1 EA 0B 31 D0 69 C0 01 80 00 00 C3
 }

//Obfuscation technique
 //Function to forge a pointer to syscall, ret instructions.
 $getAddrToJumpToSyscallAndRetInstruction = {
 48 89 C8 48 8D 51 14 80 38 0F 75 0C 80 78 01 05
 75 06 80 78 02 C3 74 0A 48 FF C0 48 39 C2 75 E7
 31 C0 C3
 }
 //Obfuscation technique
 //Hash of functions
 $InternetOpenW = { 2E 8F 43 C1 }
 $InternetConnectW = { E8 60 1F 7F }
 $InternetCloseHandle = { 43 30 5C 03 }
 $HttpOpenRequestW = { A9 2D 8A 74 }
 $InternetSetOptionW = { 25 04 40 7A }
 $HttpAddRequestHeadersW = { 35 25 AF A5 }
 $HttpSendRequestW = { 80 7B 17 E8 }
 $HttpSendRequestA = { 3A 79 F2 E6 }
 $HttpQueryInfoA = { 27 AF D2 5D }
 $InternetReadFile = { 46 CE FE BC }
 $InternetQueryDataAvailable = { AE D7 26 30 }

 condition:
 all of them
}
```

IOCs

- 4400750cbc597b7e0cec813dcaf66d00e83955a034591a5a6ba40547a045721b
- bd32cbb6c08eff7fc6aa0bfe2fd81ec467f70d9b726015859da39744271bbcb0
- 780b2b715aa33e8910479a671469ad27cc88a7ed513b83e43cf7a6a16f613013
- 206.166.251[.]128
- 179.43.144[.]250
- 213.215.163[.]51