A Deep Dive into Brute Ratel C4 payloads – Part 2

cybergeeks.tech/a-deep-dive-into-brute-ratel-c4-payloads-part-2/

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. This second part of the analysis presents the remaining commands executed by the agent. The commands include: user impersonation, inject shellcode into processes, create and stop processes, extract information about the processes and services, create TCP listeners, block keyboard and mouse input events, extract Windows registry keys and values, and others. You can consult the first part of the analysis here.

Technical analysis

SHA256: d71dc7ba8523947e08c6eec43a726fe75aed248dfd3a7c4f6537224e9ed05f6f

We continue to describe the commands that can be used by the Brute Ratel agent.

0x0703 ID – Stop the current process

The malware stops the current process by calling the ExitProcess API:

Figure 1

0x6BAE/0x6F39 ID – User impersonation

The binary retrieves a pseudo handle for the current process using GetCurrentProcess:

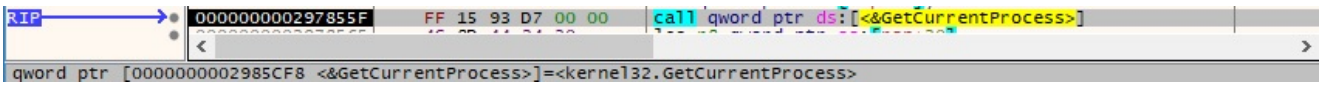

Figure 2

OpenProcessToken is utilized to open the access token associated with the process (0x28 = **TOKEN_ADJUST_PRIVILEGES** | **TOKEN_QUERY**):

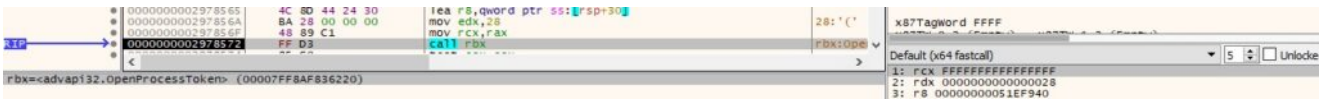

Figure 3

The process extracts the locally unique identifier (LUID) for the "SeDebugPrivilege" privilege (Figure 4).

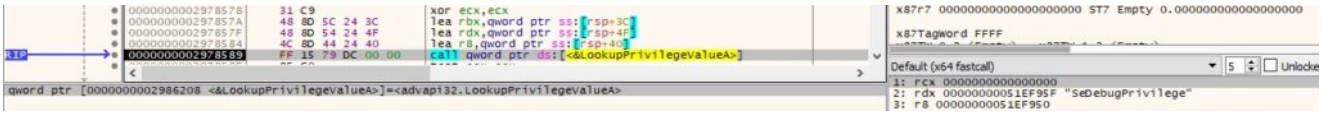

Figure 4

The executable enables the above privilege via a function call to AdjustTokenPrivileges:

Figure 5

The running processes are enumerated using the Process32FirstW and Process32NextW functions:

Figure 6

Figure 7

The agent is looking for the “LogonUI.exe”, “winlogon.exe”, and “lsass.exe” processes:

Figure 8

It opens the first process found using the OpenProcess method (0x400 = **PROCESS_QUERY_INFORMATION**):

Figure 9

ImpersonateLoggedOnUser is used to impersonate the security content of the user extracted from the process identified above:

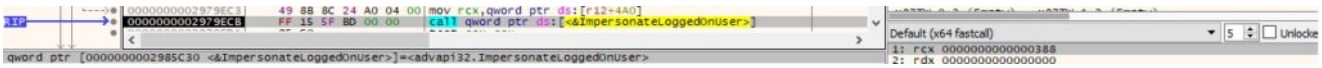

Figure 10

In order to confirm that the operation was successful, the malware calls the GetUserNameW API (see Figure 11).

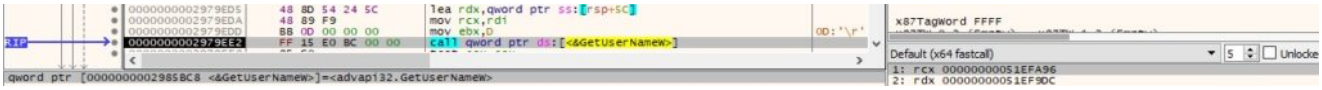

Figure 11

The message displayed in Figure 12 will be sent to the C2 server:

```
rcx=82299023EFEF0000
r12=000000000045B08A0 L"[+] Impersonated 'SYSTEM' via technique 2"
```
Figure 12

On another branch, the binary calls the DuplicateTokenEx method in order to duplicate the access token extracted from “winlogon.exe” or “lsass.exe”. Finally, a new process is created using CreateProcessWithTokenW.

0xA86A ID – Inject code into a remote process

The malicious executable converts the process ID passed as a parameter using atoi:


Figure 13

The shellcode to be executed is Base64-decoded by calling the CryptStringToBinaryA API (0x1 = **CRYPT_STRING_BASE64**):

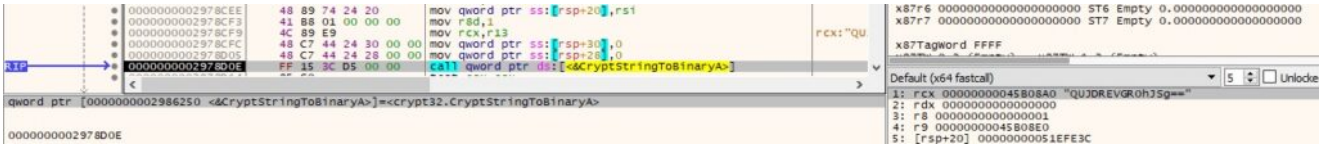
Figure 14

The badger opens the target process using OpenProcess (0x1F0FFF = **PROCESS_ALL_ACCESS**):


Figure 15

VirtualAllocEx is utilized to allocate a new memory area in the remote process (0x3000 = **MEM_COMMIT** | **MEM_RESERVE**, 0x4 = **PAGE_READWRITE**):


Figure 16

The malware writes the shellcode to the above area via a function call to WriteProcessMemory, as shown in Figure 17.


Figure 17

The page’s protection is changed using the VirtualProtectEx API (0x20 = **PAGE_EXECUTE_READ**):

Figure 18

Finally, the binary creates a thread in the remote process that executes the shellcode:

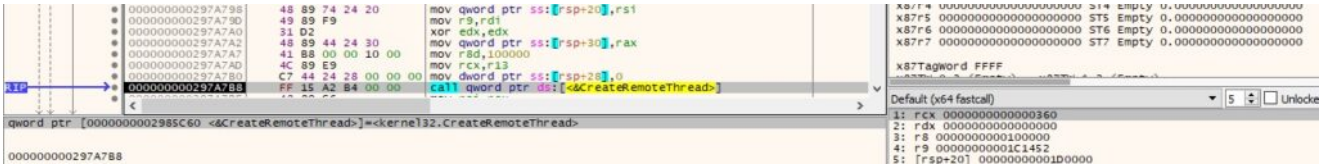

Figure 19

0xE9B0 ID – Create a process and read its output via a pipe

The agent creates an anonymous pipe using the CreatePipe method:

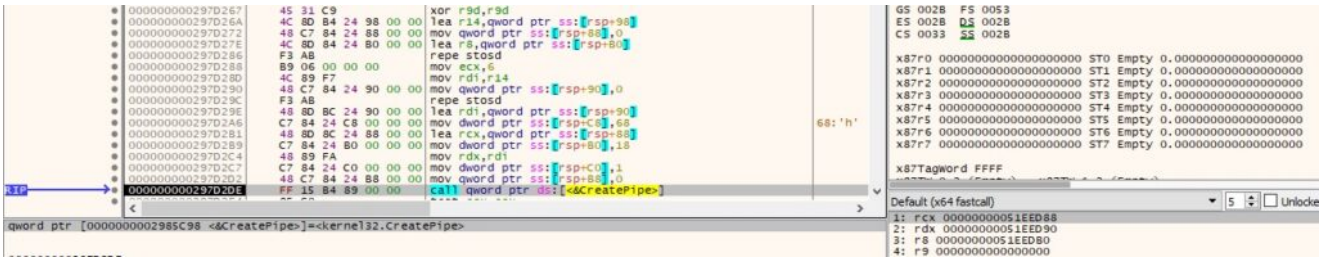

Figure 20

The pipe is set to be inherited via a call to SetHandleInformation (0x1 = **HANDLE_FLAG_INHERIT**):

Figure 21

The malicious executable creates a process specified by the C2 server using the CreateProcessA API, as shown in the figure below.

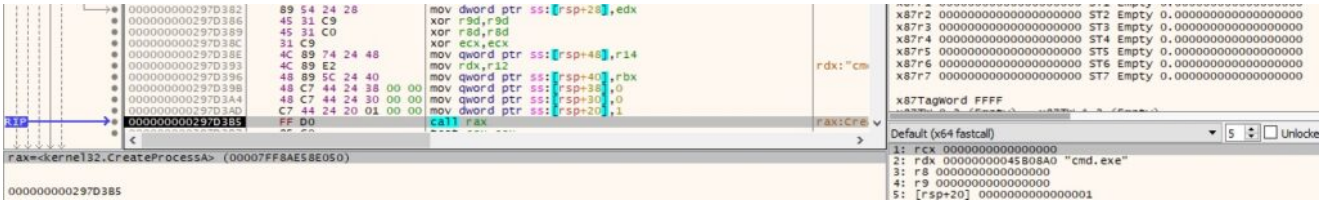

Figure 22

The process' output that resides in the anonymous pipe is copied into a buffer by calling PeekNamedPipe (Figure 23).

Figure 23

The output is read using ReadFile and then transmitted to the C2 server:

Figure 24

0x91B3 ID – Inject code into the current process

The CryptStringToBinaryA method is utilized to decode from Base64 the shellcode that will be executed:

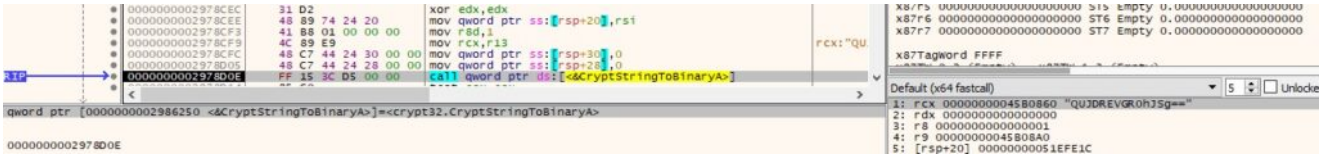

Figure 25

The agent creates a named pipe (0x80000003 = **FILE_FLAG_WRITE_THROUGH** | **PIPE_ACCESS_DUPLEX**):

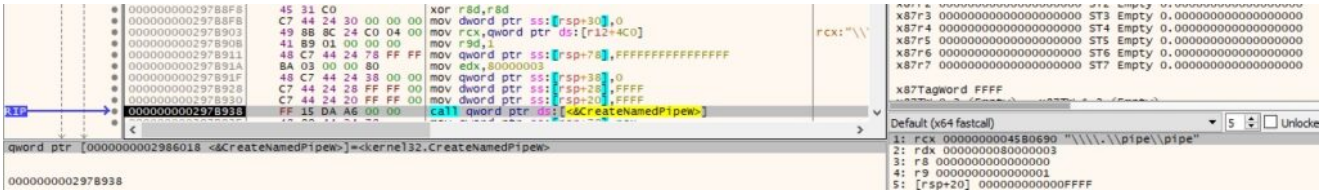

Figure 26

A new thread is created using the CreateThread function. In this thread, the malware connects to the pipe and reads data using the ConnectNamedPipe and ReadFile methods:

Figure 27

Figure 28

VirtualAllocEx is used to allocate a new memory area in the current process:

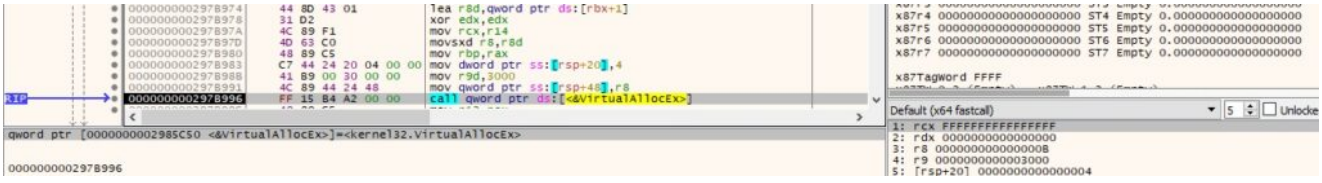

Figure 29

The shellcode is copied into the new area and its page is made executable, as highlighted below:

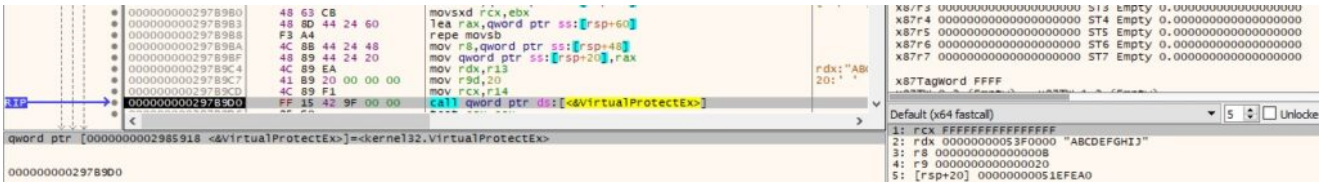

Figure 30

A new thread runs the shellcode copied earlier:

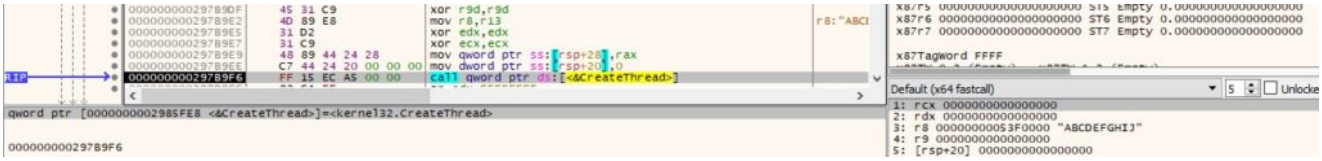

Figure 31

0x1719 ID – Enable SeDebugPrivilege

The malicious process calls the LookupPrivilegeValueA function with the “SeDebugPrivilege” parameter:

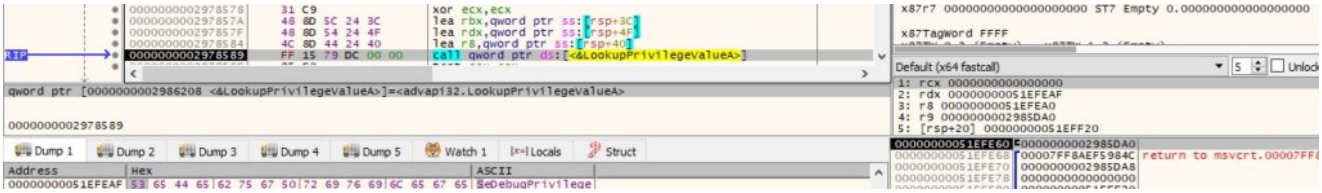

Figure 32

The PrivilegeCheck API is utilized to determine if the above privilege is enabled in the access token:

Figure 33

The message displayed in Figure 34 will be sent to the C2 server as a confirmation.

```
rdx=10
qword ptr [rsp+28]=[00000000051EFEF8 &L"[+] Enabled debug privilege"]
```

Figure 34

0x4FFE ID – Extract the status of the token’s privileges

The badger obtains the TOKEN_PRIVILEGES structure that contains the privileges of the token using GetTokenInformation (see Figure 35).

Figure 35

It retrieves the name of the privileges represented by a locally unique identifier (LUID) via a function call to LookupPrivilegeNameW:

Figure 36

The list of privileges and their status is written in the memory. The following statuses can be specified: “[+] %-50ls Enabled (Default)”, “[+] %-50ls Enabled (Adjusted)”, “[+] %-50lsDisabled\n”, “[+] Elevated”, or “[+] Restricted”.

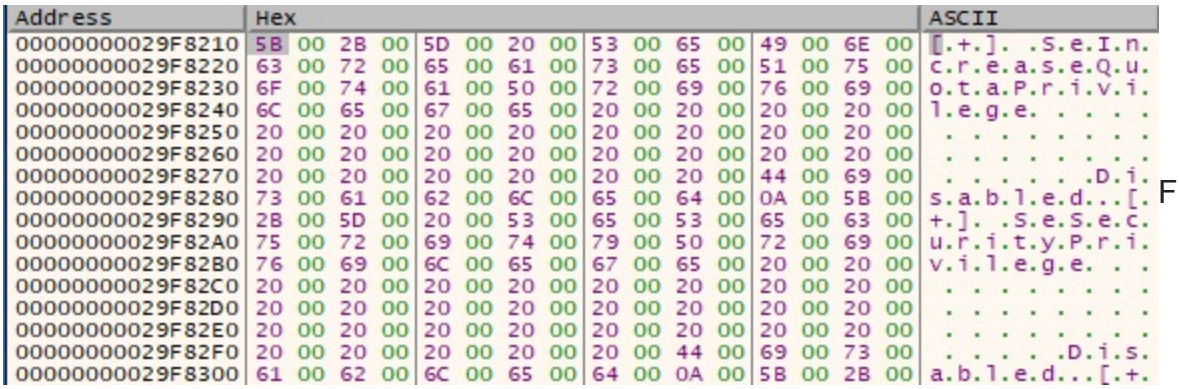

Figure 37

0x9DE0 ID – Extract Username, PPID, PID, and Executable path for every running process

The binary obtains a snapshot of all processes in the system using CreateToolhelp32Snapshot. It enumerates them using the Process32FirstW and Process32NextW methods:

Figure 38

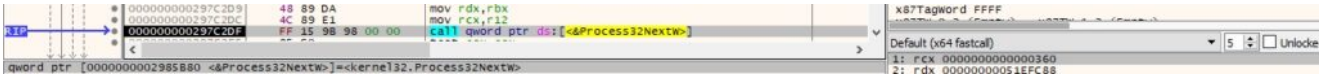

Figure 39

The agent tries to open the local process object using OpenProcess (0x410 = **PROCESS_QUERY_INFORMATION** | **PROCESS_VM_READ**):

Figure 40

For each of the access token extracted from the processes, the executable calls the GetTokenInformation function and retrieves the user account of the token (Figure 41).

Figure 41

The malware extracts the name of the account for the security identifier (SID) and the first domain on which the SID is found:

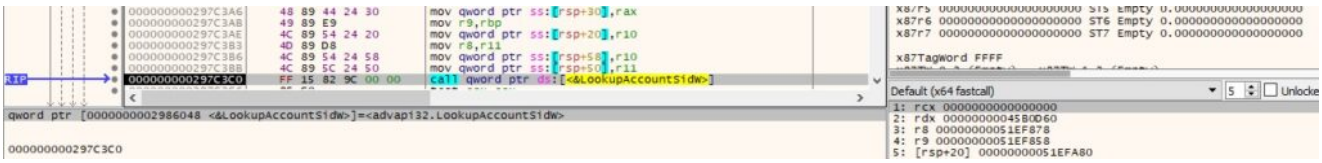

Figure 42

0xEBC0 ID – Kill processes

The target process is opened via a function call to OpenProcess (0x1 = **PROCESS_TERMINATE**):

Figure 43

The process is killed using the TerminateProcess API:

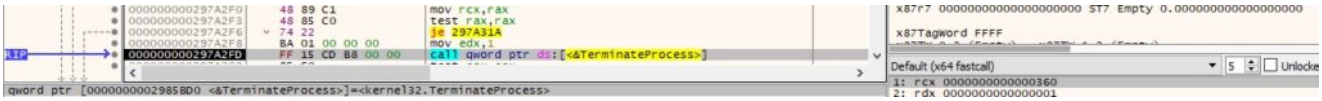

Figure 44

0xF584 ID – Create a new process using the Domain, Username, and Password received from the C2 server

The binary spawns a new process using the CreateProcessWithLogonW method. The parameters are modified according to the command's arguments:

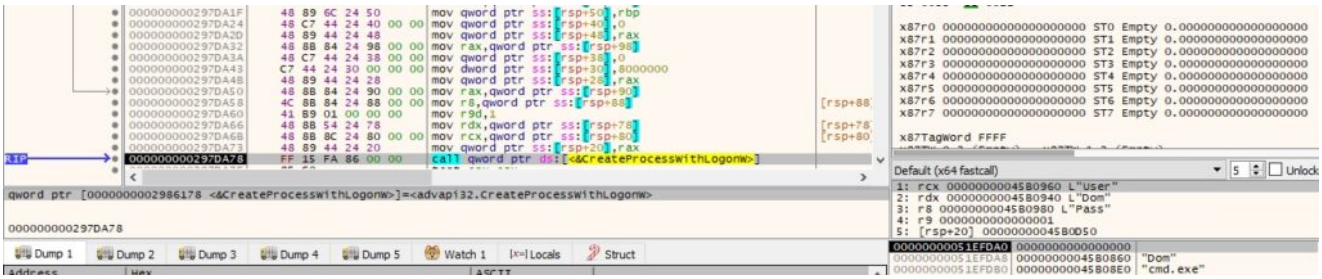

Figure 45

0xBED0 ID – Execute the "open", "runas", or "print" command

The first parameter is compared with the above commands, as shown in Figure 46.

```
.text:000000000297E872 mov     dword ptr [rsp+2F8h+Str2], 'nepo'
.text:000000000297E87A movsxd  rcx, ecx        ; Count
.text:000000000297E87D mov     dword ptr [rsp+2F8h+var_2B4], 'anur'
.text:000000000297E885 mov     byte ptr [rsp+2F8h+var_2B0], 's'
.text:000000000297E88A mov     dword ptr [rsp+2F8h+var_2AE], 'nirp'
.text:000000000297E892 mov     byte ptr [rsp+2F8h+var_2AA], 't'
```

Figure 46

We could use the runas command to spawn a cmd.exe process:

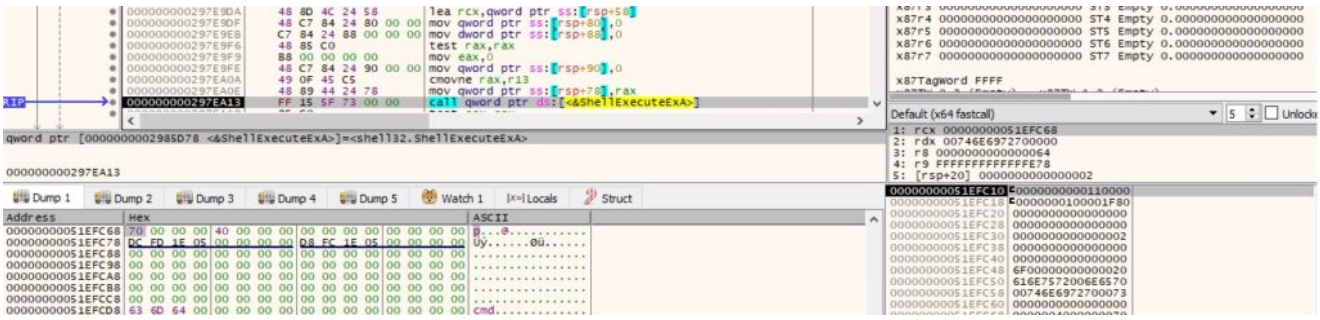

Figure 47

GetProcessId is utilized to obtain the PID of the newly created process:

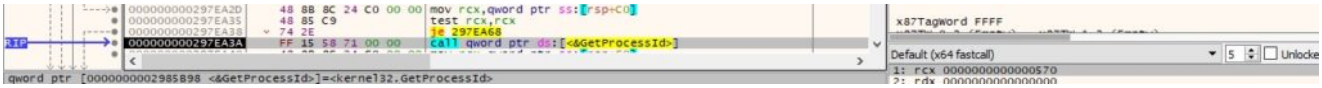

Figure 48

0xE2EA ID – Copy bytes into memory

The second parameter is Base64-decoded by calling the CryptStringToBinaryA API:

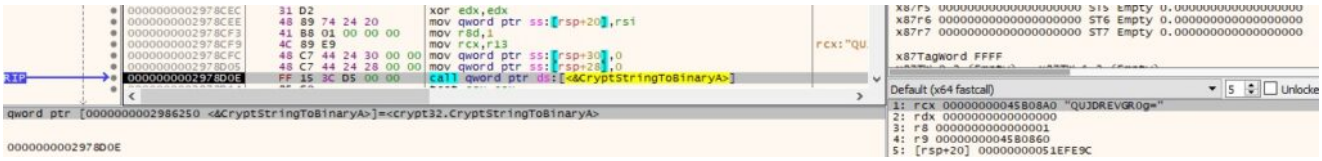

Figure 49

The address containing the resulting bytes is stored in a table that contains functions pointers (see Figure 50).

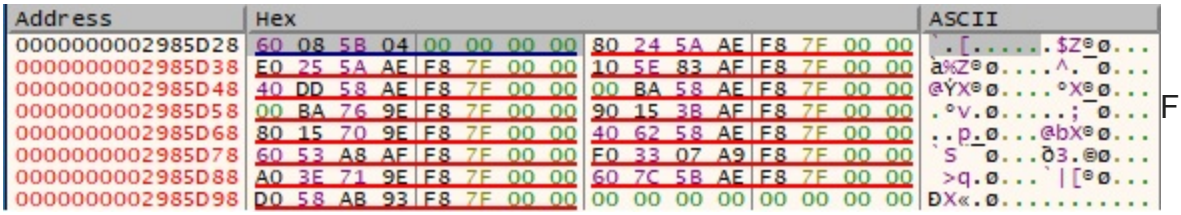

Figure 50

Depending on the number of bytes, the malware will send the “[+] Imported %d bytes” message to the C2 server:

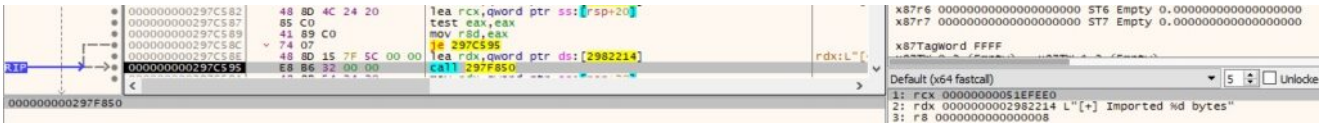

Figure 51

0x6154 ID – Free the pointer storing the address of the imported bytes

The agent calls the free function with the pointer displayed in the above command. The message shown below is transmitted to the C2 server.

```
.text:000000000297C492 mov     rcx, cs:qword_2985D28 ; Memory
.text:000000000297C499 call    free
.text:000000000297C49E mov     rcx, cs:qword_2986000
.text:000000000297C4A5 mov     cs:qword_2985D28, 0
.text:000000000297C4B0 call    cs:qword_2985D30
.text:000000000297C4B6 lea     rcx, [rsp+38h+Memory]
.text:000000000297C4BB lea     rdx, aImportCleared ; "[+] Import cleared"
.text:000000000297C4C2 call    sub_297F850
```

Figure 52

0x699A ID – Create a TCP listener

The process creates a new thread that is responsible for the listener creation:

Figure 53

It calls the getaddrinfo method with the port number and the first parameter being NULL, which returns all registered addresses on the local machine:

Figure 54

The badger creates a TCP socket (0x2 = **AF_INET**, 0x1 = **SOCK_STREAM**, 0x6 = **IPPROTO_TCP**):

Figure 55

The bind function is used to associate the local address with the socket, as highlighted below:

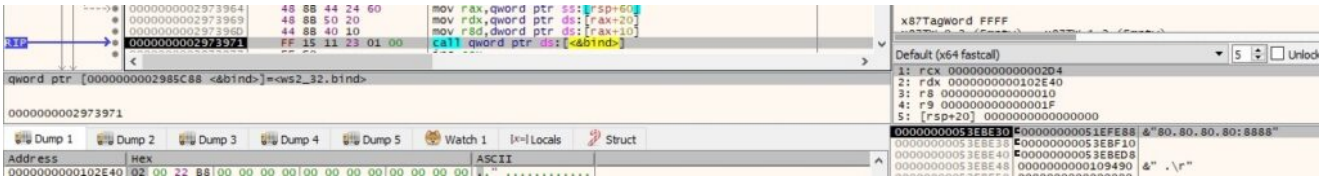

Figure 56

The malware starts listening on the port specified in the command's arguments (in our case, 8888):

Figure 57

Finally, the accept method is utilized to allow incoming connection attempts (Figure 58).

Figure 58

The IP address from the connection is converted into an ASCII string in dotted-decimal format:

Figure 59

A new thread that handles the receive operation is created:

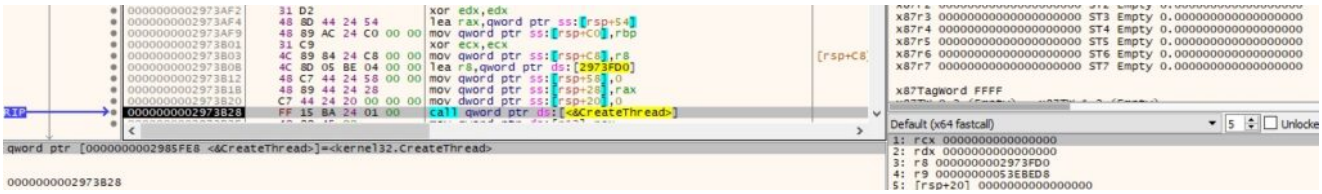

Figure 60

Figure 61

0xB458 ID – Extract information about Windows services

The binary opens the service control manager on the local machine using OpenSCManagerA (0x4 = **SERVICE_QUERY_STATUS**):

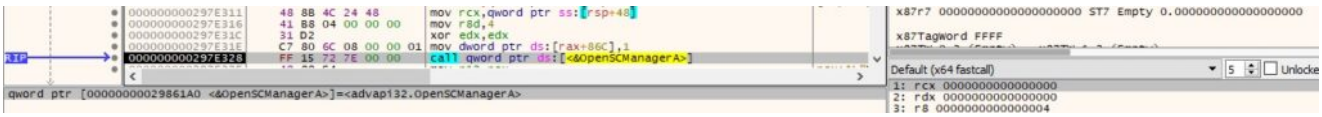

Figure 62

EnumServicesStatusW is used to enumerate all services in the database (0x30 = **SERVICE_WIN32**, 0x3 = **SERVICE_STATE_ALL**):

Figure 63

For every service, the malware calls the OpenServiceW API (0x1 = **SERVICE_QUERY_CONFIG**):

Figure 64

The agent extracts the configuration parameters of the service using QueryServiceConfigW. The following fields are relevant: display name, service name, service state, service path, service user, and service type.

Figure 65

0xE3CB ID – Retrieve information about Domain Controllers and policies

The malicious executable obtains the name of a domain controller via a function call to DsGetDcNameW, as displayed in Figure 66.

Figure 66

The DsGetDcOpenW API is utilized to open a new domain controller enumeration operation (0x2 = **DS_NOTIFY_AFTER_SITE_RECORDS**):

Figure 67

The badger extracts the global password parameters and lockout information by calling the NetUserModalsGet function. The information is organized using the following structure:

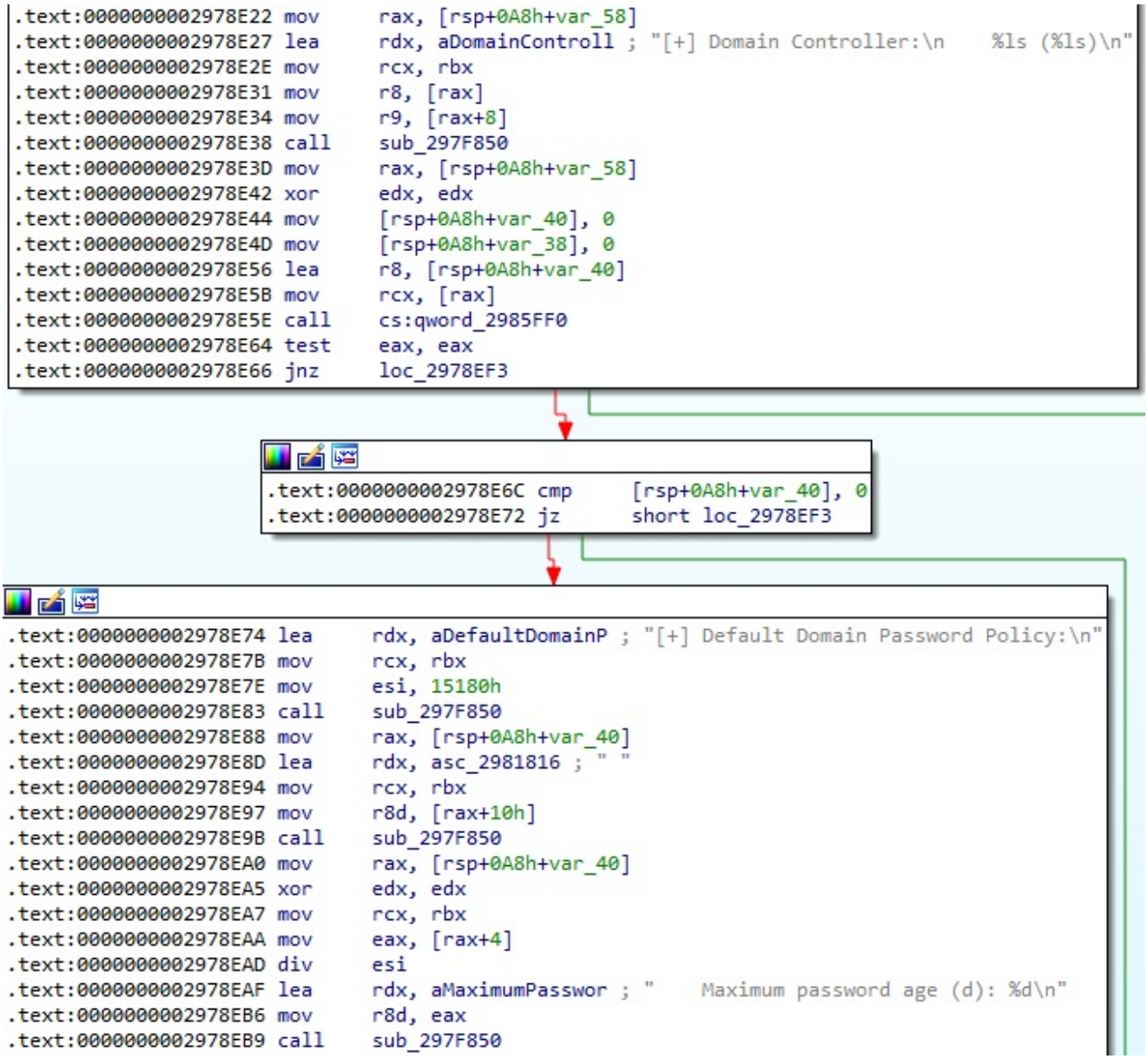

Figure 68

0x0105 ID – Extract data from the clipboard

The process opens the clipboard by calling the OpenClipboard method:

Figure 69

The data is obtained from the clipboard in the Unicode format (0xD = **CF_UNICODETEXT**):

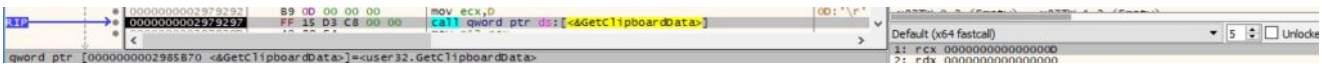

Figure 70

0x0B06 ID – Convert the time of the last input event in minutes

The binary obtains the number of milliseconds elapsed since the system was started using GetTickCount:

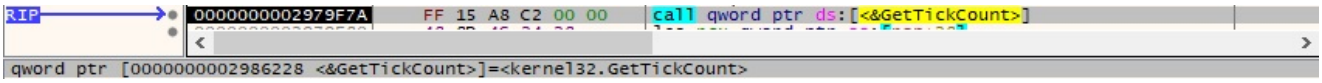

Figure 71

GetLastInputInfo is used to retrieve the time of the last input event:

Figure 72

0xB63A ID – Block keyboard and mouse input events

The BlockInput method is used to perform the operation, as displayed in the figure below.

Figure 73

0x0391 ID – Lock the workstation's display

LockWorkStation is utilized to lock the display (see Figure 74).

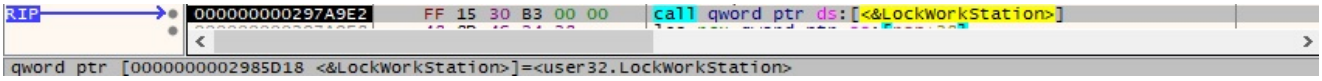

Figure 74

0xF999 ID – Impersonate the context of a logged-on user

The badger attempts to log a user on to the local machine via a call to LogonUserA (0x2 = **LOGON32_LOGON_INTERACTIVE**):

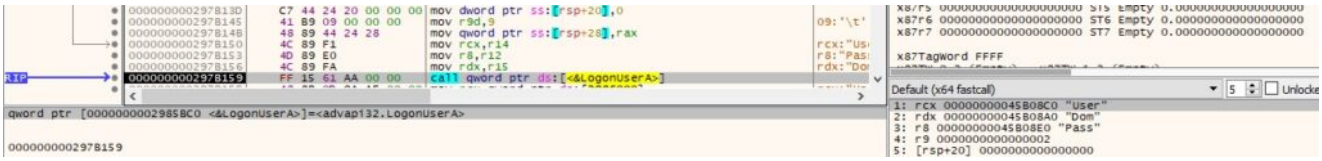

Figure 75

The binary impersonates the context of the above user using the ImpersonateLoggedOnUser function:

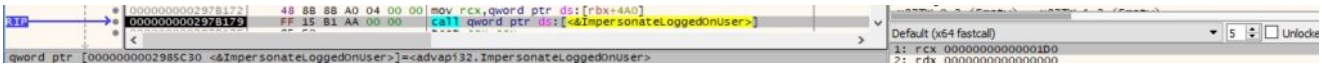

Figure 76

0xA959 ID – Retrieve information about users

The first parameter is compared with "user" and "users". In the first case, the malware calls the NetUserGetInfo API to obtain information about the user account:

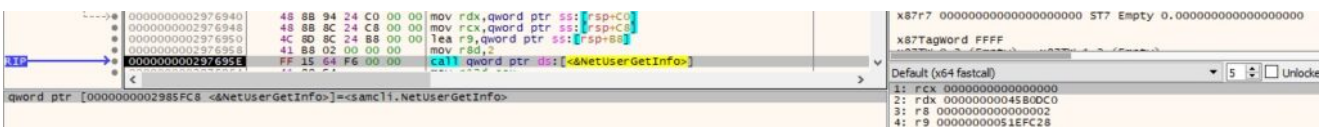

Figure 77

The information is organized in the following manner:

```
.text:0000000002976A9A mov     [rsp+328h+var_290], eax
.text:0000000002976AA1 lea     rax, aNumberOfLogons ; "Number of logons"
.text:0000000002976AA8 mov     [rsp+328h+var_298], rax
.text:0000000002976AB0 mov     eax, [rcx+80h]
.text:0000000002976AB6 mov     [rsp+328h+var_2B0], rsi
.text:0000000002976ABB mov     [rsp+328h+var_2A0], eax
.text:0000000002976AC2 lea     rax, aB          ; "B"
.text:0000000002976AC9 mov     [rsp+328h+var_2A8], rax
.text:0000000002976AD1 lea     rax, aLastLogon ; "Last logon"
.text:0000000002976AD8 mov     [rsp+328h+var_2B8], rax
.text:0000000002976ADD mov     eax, [rcx+10h]
.text:0000000002976AE0 mov     [rsp+328h+var_2D0], rbx
.text:0000000002976AE5 div     r8d
.text:0000000002976AE8 lea     r8, aUserName   ; "User name"
.text:0000000002976AEF lea     rdx, asc_29824EE ; "%"
.text:0000000002976AF6 mov     [rsp+328h+var_2C0], eax
.text:0000000002976AFA lea     rax, aPasswordLastSe ; "Password last set"
.text:0000000002976B01 mov     [rsp+328h+var_2C8], rax
.text:0000000002976B06 lea     rax, aAccountExpires ; "Account expires"
.text:0000000002976B0D mov     [rsp+328h+var_2D8], rax
.text:0000000002976B12 mov     eax, [rcx+90h]
.text:0000000002976B18 mov     [rsp+328h+var_2E0], eax
.text:0000000002976B1C lea     rax, aCountryRegionC ; "Country/region code"
.text:0000000002976B23 mov     [rsp+328h+var_2E8], rax
.text:0000000002976B28 mov     rax, [rcx+20h]
.text:0000000002976B2C mov     [rsp+328h+var_2F0], rax
.text:0000000002976B31 lea     rax, aComment   ; "Comment"
.text:0000000002976B38 mov     [rsp+328h+var_2F8], rax
.text:0000000002976B3D mov     rax, [rcx+40h]
.text:0000000002976B41 mov     [rsp+328h+var_300], rax
.text:0000000002976B46 lea     rax, aFullName  ; "Full name"
.text:0000000002976B4D mov     [rsp+328h+var_308], rax
```

Figure 78

In the second case, the agent retrieves information about all user accounts on the local computer (0x2 = **FILTER_NORMAL_ACCOUNT**):

Figure 79

0x6C36 ID – Extract registry keys and values

The first argument can be “hklm”, “hkcu”, “root”, “config”, and “users”. These are Windows registry hives.

The registry key passed as the second argument is opened using the RegOpenKeyExA method (0x20019 = **KEY_READ**):

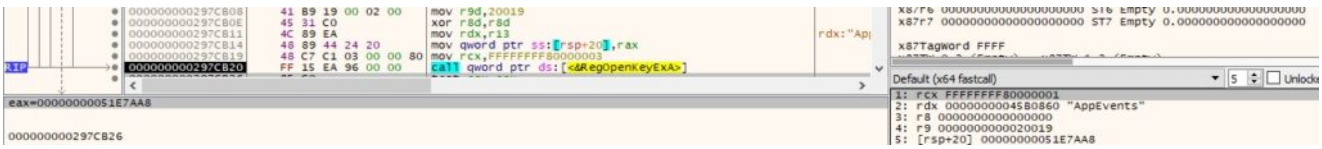

Figure 80

The malicious process retrieves information about the registry key by calling the RegQueryInfoKeyW function:

Figure 81

It enumerates the subkeys of the key using RegEnumKeyExW (Figure 82).

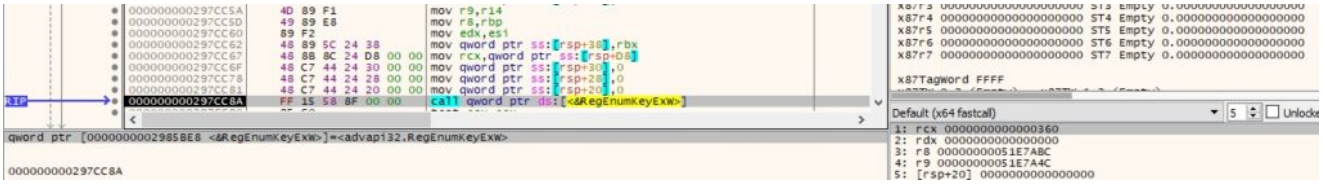

Figure 82

For each of the subkeys, the malware calls the RegEnumValueW API in order to enumerate the registry values:

Figure 83

Finally, the type and data for all registry values identified are extracted:

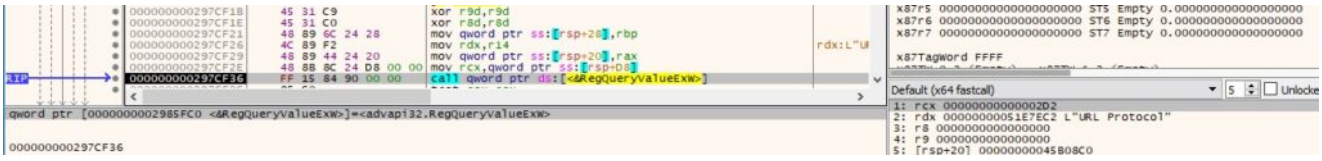

Figure 84

0x9C41 ID – Take a screenshot and send it to the C2 server

The GdiplusStartup function initializes Windows GDI+ (see Figure 85).

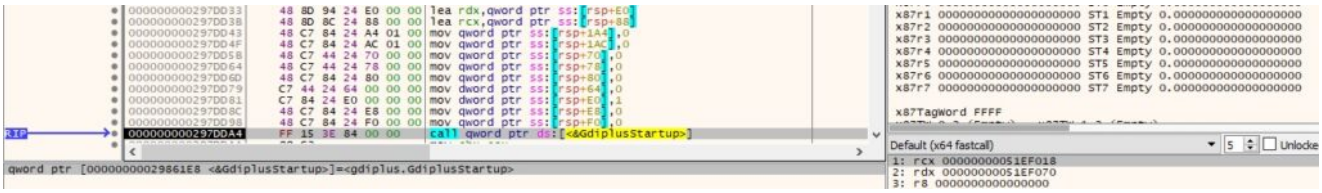

Figure 85

The agent retrieves a handle to the desktop window via a call to GetDesktopWindow:

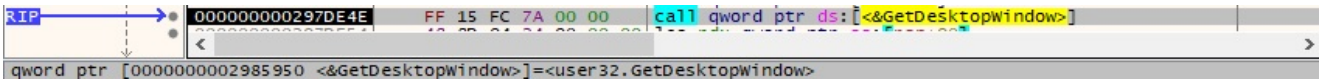

Figure 86

It obtains the number of adjacent color bits for each pixel for the device context (DC) for the above window (0xC = **BITSPIXEL**):

Figure 87

The BitBlt method is used to capture the image:

Figure 88

The malware creates a Bitmap object based on a handle to a Windows GDI bitmap and a handle to a GDI palette:

Figure 89

The process calls the CLSIDFromString function with the “1d5be4b5-fa4a-452d-9cdd-5db35105e7eb” CLSID – Quality field:

Figure 90

GdipSaveImageToStream is utilized to save the screenshot to a stream (see Figure 91). The name of the image is derived from the current date and time.

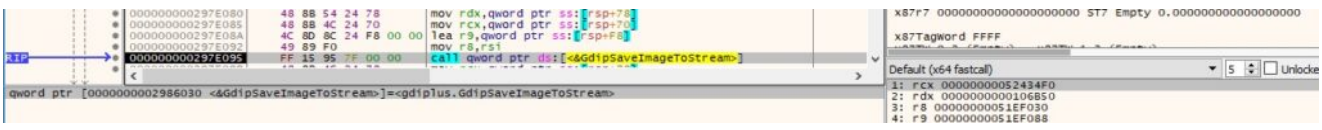

Figure 91

0x3C4D ID – Read content from pipe and send it to the C2 server. Write server's response to the pipe

The agent opens an existing pipe using the CreateFileA API (0xC0000000 = **GENERIC_READ** | **GENERIC_WRITE**, 0x3 = **OPEN_EXISTING**):

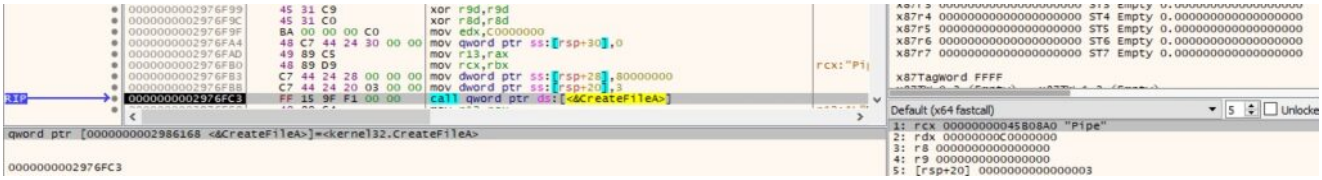

Figure 92

The malware modifies the read and the blocking mode via a function call to SetNamedPipeHandleState (0x0 = **PIPE_READMODE_BYTE** | **PIPE_WAIT**):

Figure 93

The pipe's content is read using the ReadFile method:

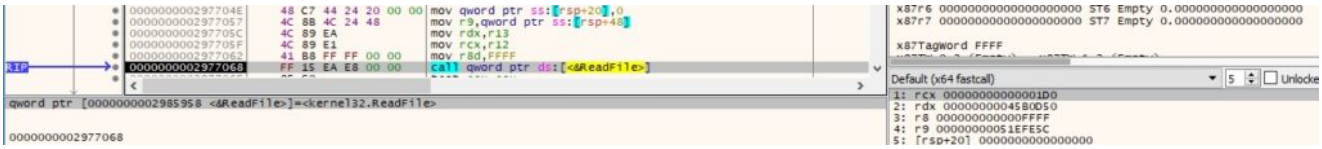

Figure 94

The content is exfiltrated to the C2 server, and the server's response is written back to the pipe.

0x2129 ID – Write two numbers into memory

The command takes two parameters and writes them in the following format:

Figure 95

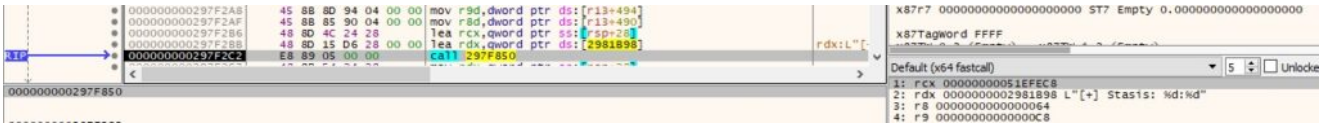

Figure 96

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/

MDSec: https://www.mdsec.co.uk/2022/08/part-3-how-i-met-your-beacon-brute-ratel/