Assignment 4

Adam Livingston

University Of Arizona

CYBV 454 MALWARE THREATS & ANALYSIS

Professor Galde

31 Jan 2023

CYBV 454 Assignment 4 LIVINGSTON

LAB 6-2

- LAB06-02.exe : c0b54534e188e1392f28d17faff3d454 (Figure 1)

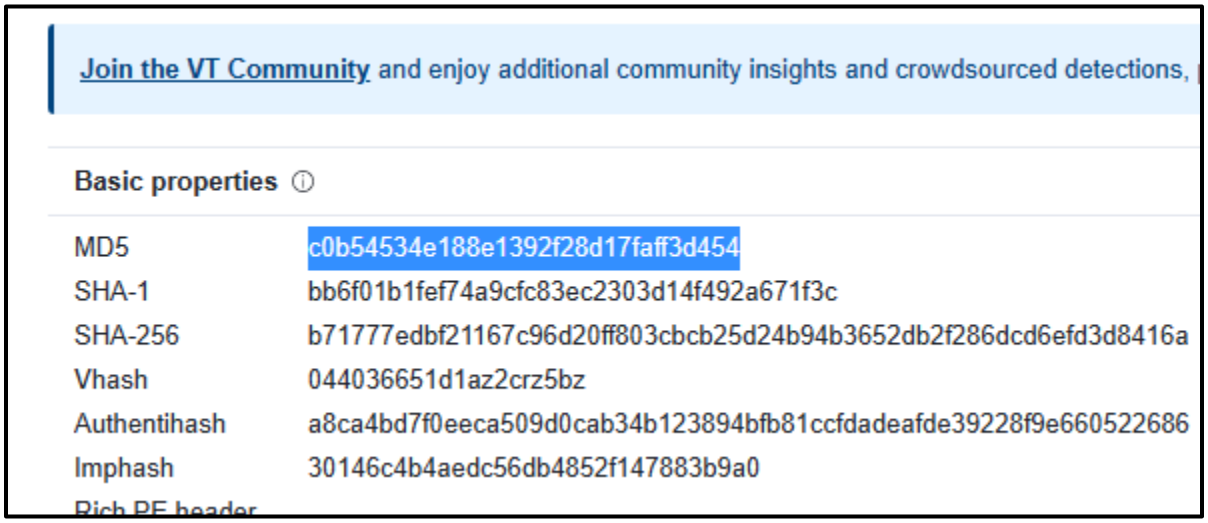

Figure 1: Virus Total MD5 Hash for file Lab06-02.exe.

Virus Total found 39 matching signatures for a generic trojan (Figure 2) and has a compilation timestamp of 2011-02-02 at 21:29:05 UTC (Figure 3). This file was compiled on my 21st birthday.

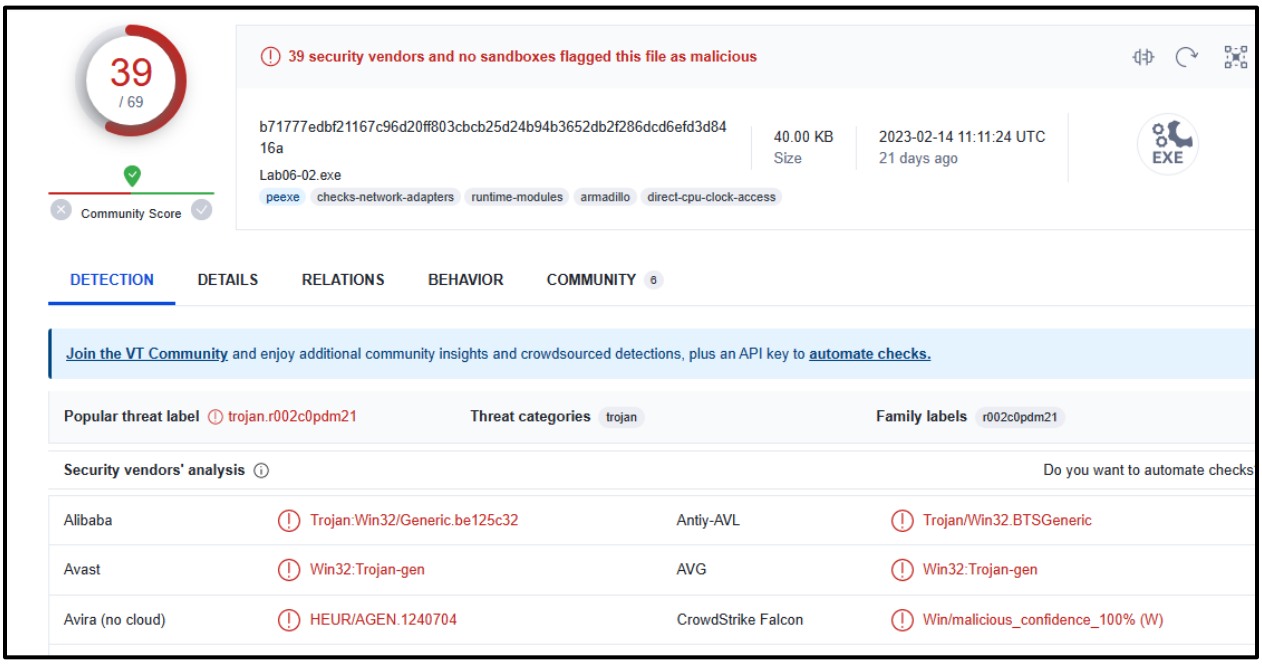

Figure 2: Virus Total Findings for file Lab06-02.exe.

Header	
Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2011-02-02 21:29:05 UTC
Entry Point	4528
Contained Sections	3

Figure 3: Virus Total compilation timestamp for file Lab06-02.exe.

The file only appears to import two dynamic linked libraries: kerenel32 and wininet. Kernel32.dll indicates that it has the capability to access and modify the core OS functions. Wininet.dll shows that it imports and implements functions related to networking protocols.

Imports

+ KERNEL32.dll
+ WININET.dll

Figure 4: Virus Total imports for file Lab06-02.exe.

The networking protocol functions that are supported within wininet.dll may be related to the detected network connections identified by Virus Total for Lab06-02.exe. There are multiple HTTP requests to practicalmalwareanalysis.com (Figure 5) and lots of IP traffic with potential Transport Layer Security (TLS) implemented for practicalmalwareanalysis.com (Figure 6).

HTTP Requests

+ http://practicalmalwareanalysis.com/?post_type=feedback&p=374
+ http://practicalmalwareanalysis.com/cc.htm
+ http://www.practicalmalwareanalysis.com
+ http://www.practicalmalwareanalysis.com/cc.htm
+ https://practicalmalwareanalysis.com
+ https://www.practicalmalwareanalysis.com

Figure 5: Virus Total HTTP requests for file Lab06-02.exe.

IP Traffic

104.65.174.220:80 (TCP)
104.80.88.97:80 (TCP)
104.98.118.138:80 (TCP)
104.99.72.226:80 (TCP)
13.107.4.50:80 (TCP)
15.197.142.173:80 (TCP)
192.0.78.24:443
192.0.78.24:443 (TCP)
192.0.78.24:80
192.0.78.24:80 (TCP)

TLS

+ www.practicalmalwareanalysis.com

Figure 6: Virus Total IP traffic for file Lab06-02.exe.

Virus Total also reports that the file has behavior of privilege escalation and defense evasion (Figure 7). It also shows behavior of downloading files using HTTP and using HTTPS for encrypted channels (Figure 8), most likely in reference for the TLS networking behavior identified in Figure 6. Based on these findings, this malware is most likely a generic trojan as

identified in Figure 2 and uses HTTP and HTTPS (ports 80 and 443) to download additional packages onto the infected machine when the file is ran by the user.

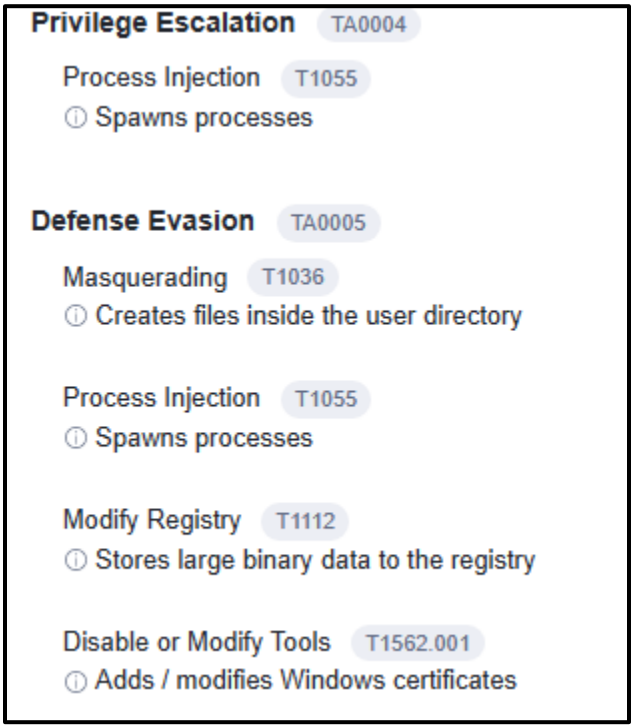

Figure 7: Virus Total behavior for file Lab06-02.exe.

Command and Control TA0011

Application Layer Protocol T1071
- Uses HTTPS
- Downloads files from webservers via HTTP
- Performs DNS lookups
- Tries to download or post to a non-existing http route (HTTP/1.1 404 Not Found / 503 Service Unavailable)

Non-Application Layer Protocol T1095
- Downloads files from webservers via HTTP
- Performs DNS lookups
- Tries to download or post to a non-existing http route (HTTP/1.1 404 Not Found / 503 Service Unavailable)

Ingress Tool Transfer T1105
- Downloads files from webservers via HTTP
- Some HTTP requests failed (404). It is likely the sample will exhibit less behavior
- Tries to download or post to a non-existing http route (HTTP/1.1 404 Not Found / 503 Service Unavailable)

Encrypted Channel T1573
- Uses HTTPS
- Uses HTTPS for network communication, use the SSL MITM Proxy cookbook for further analysis

Figure 8: Virus Total network-based behavior for file Lab06-02.exe.

LAB 6-2

LAB 6-2 Question 1

What operation does the first subroutine called by main perform?

BLUF: An if() statement that checks to see if there is an active internet connection.

When we place the file into IDA pro, we are taken to the _main function at 0x00401130 and can see the first subroutine called at 0x00401136 that takes us to 0x00401000 due to the naming convention of sub_401000 (Figure 9).

```
00401130
00401130
00401130                        ; Attributes: bp-based frame
00401130
00401130                        ; int __cdecl main(int argc,const char **argv,const char *envp)
00401130                        _main proc near
00401130
00401130                        var_8= byte ptr -8
00401130                        var_4= dword ptr -4
00401130                        argc= dword ptr  8
00401130                        argv= dword ptr  0Ch
00401130                        envp= dword ptr  10h
00401130
00401130 000 55                 push    ebp
00401131 004 8B EC              mov     ebp, esp
00401133 004 83 EC 08           sub     esp, 8          ; Integer Subtraction
00401136 00C E8 C5 FE FF FF     call    sub_401000      ; Call Procedure
0040113B 00C 89 45 FC           mov     [ebp+var_4], eax
0040113E 00C 83 7D FC 00        cmp     [ebp+var_4], 0  ; Compare Two Operands
00401142 00C 75 04              jnz     short loc_401148 ; Jump if Not Zero (ZF=0)
```

Figure 9: Lab06-02.exe _main function and first subroutine call.

After double-clicking on sub_401000, we are taken to 0x00401000 and see that we are taken to the very top portion of the IDA view disassembly code (Figure 10). We know this because IDA Pro automatically populates information in blue text at the very top of the disassembly. The graph view of this subroutine in its entirety is seen in Figure 11.

```
.text:00401000      ;
.text:00401000      ; +-------------------------------------------------------------------------+
.text:00401000      ; |      This file is generated by The Interactive Disassembler (IDA)        |
.text:00401000      ; |      Copyright (c) 2010 by Hex-Rays SA, <support@hex-rays.com>          |
.text:00401000      ; |                        Licensed to: Freeware version                    |
.text:00401000      ; +-------------------------------------------------------------------------+
.text:00401000      ;
.text:00401000      ; Input MD5   : C0B54534E188E1392F28D17FAFF3D454
.text:00401000
.text:00401000      ; File Name   : C:\Labs\BinaryCollection\Chapter_6L\Lab06-02.exe
.text:00401000      ; Format      : Portable executable for 80386 (PE)
.text:00401000      ; Imagebase   : 400000
.text:00401000      ; Section 1. (virtual address 00001000)
.text:00401000      ; Virtual size                  : 00004A78 (  19064.)
.text:00401000      ; Section size in file          : 00005000 (  20480.)
.text:00401000      ; Offset to raw data for section: 00001000
.text:00401000      ; Flags 60000020: Text Executable Readable
.text:00401000      ; Alignment     : default
.text:00401000      ; OS type          :  MS Windows
.text:00401000      ; Application type:  Executable 32bit
.text:00401000
```

Figure 10: 0x00401000 is the very top of IDA view disassembly.

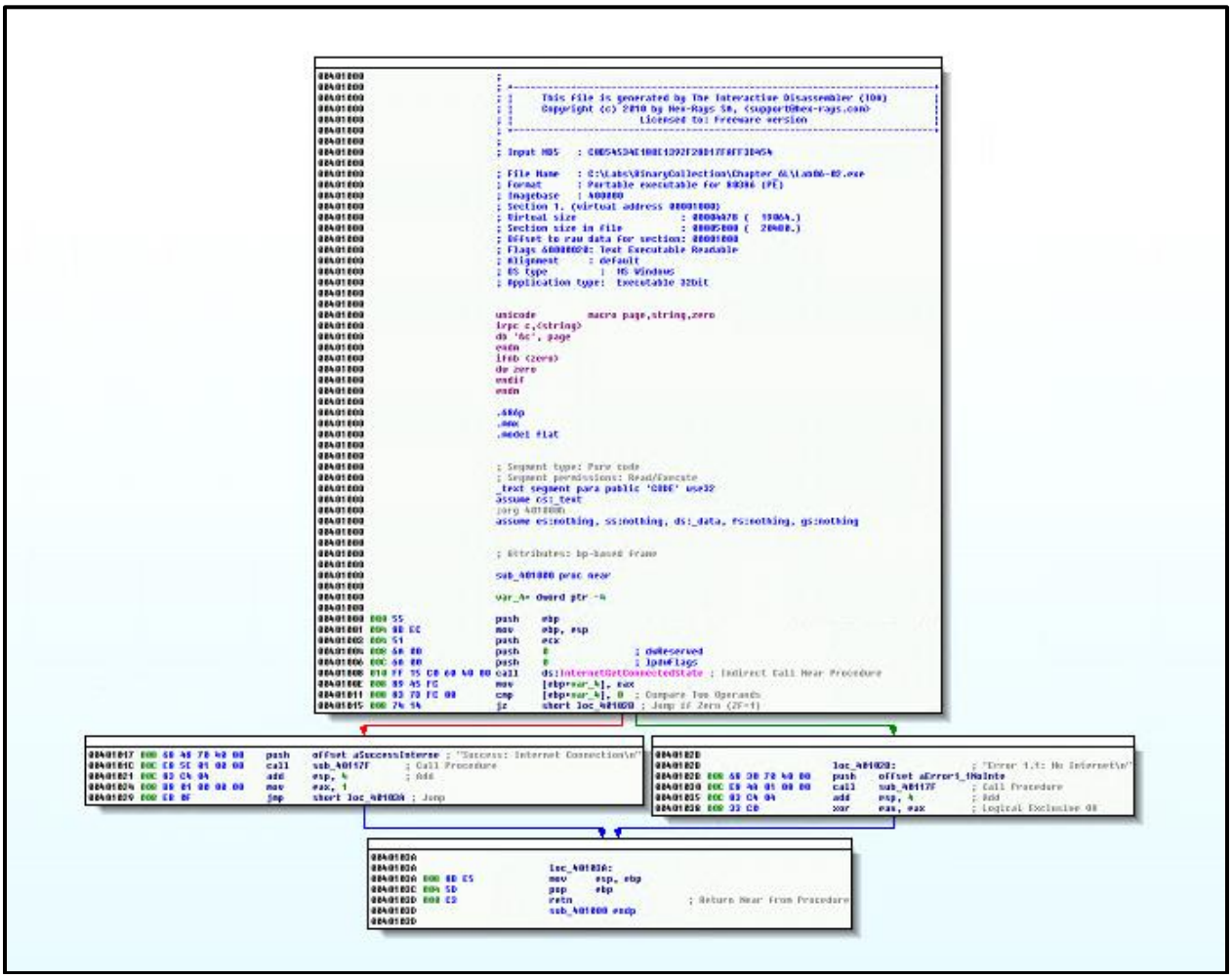

Figure 11: Graph view of sub_401000.

To identify the operation and code construct of this subroutine, we can reference the code in Figure 12. This subroutine begins by completing stack management operations and then it calls the external function “InternetGetConnectedState” at 0x00401008. Since the return values of these functions are stored in the EAX register, we see that the value is then moved into EBP+4 and then compared with 0.

```
00401000                        sub_401000 proc near
00401000
00401000                        var_4= dword ptr -4
00401000
00401000 000 55                 push    ebp
00401001 004 8B EC              mov     ebp, esp
00401003 004 51                 push    ecx
00401004 008 6A 00              push    0               ; dwReserved
00401006 00C 6A 00              push    0               ; lpdwFlags
00401008 010 FF 15 C0 60 40 00  call    ds:InternetGetConnectedState ; Indirect Call Near Procedure
0040100E 008 89 45 FC           mov     [ebp+var_4], eax
00401011 008 83 7D FC 00        cmp     [ebp+var_4], 0  ; Compare Two Operands
00401015 008 74 14              jz      short loc_40102B ; Jump if Zero (ZF=1)
```

```
ffset aSuccessInterne ; "Success: Internet Connection\n"
ub_40117F       ; Call Procedure
sp, 4           ; Add
```

```
0040102B
0040102B                        loc_40102B:
0040102B 008 68 30 70 40 00     push    offset aError
```

Figure 12: InternetGetConnectedState function.

When examining the documentation for InternetGetConnectedState (Figure 13), we see that it returns a Boolean value which will be represented by 0 for false and 1 for true. A true value represents that there is an active internet connection and false means there is no internet connection. As stated, this value will be stored in EAX upon return and then compared with 0.

Syntax

C++ Copy

```
BOOL InternetGetConnectedState(
  [out] LPDWORD lpdwFlags,
  [in]  DWORD   dwReserved
);
```

Figure 13: InternetGetConnectedState function syntax.

The true or false value being compared with 0 is followed by a “Jump if zero (jz)” instruction, meaning if the two values are equal then it will jump to the specified address at 0x004012B. We see in that address that an error note is pushed onto the stack and a comment stating “Error 1.1: No Internet\n” (Figure 14).

```
], 0 ; Compare Two Operands
 ; Jump if Zero (ZF=1)

0040102B
0040102B                        loc_40102B:             ; "Error 1.1: No Internet\n"
0040102B 008 68 30 70 40 00     push    offset aError1_1NoInte
00401030 00C E8 4A 01 00 00     call    sub_40117F      ; Call Procedure
00401035 00C 83 C4 04           add     esp, 4          ; Add
00401038 008 33 C0              xor     eax, eax        ; Logical Exclusive OR
```

Figure 14: InternetGetConnectedState returns 0 jump location.

However, if the return value of InternetGetConnectedState is 1, meaning that there is an active internet connection, then there is a message that gets displayed stating, “Success: Internet Connection\n” (Figure 15).

```
00401017 008 68 48 70 40 00     push    offset aSuccessInterne ; "Success: Internet Connection\n"
0040101C 00C E8 5E 01 00 00     call    sub_40117F      ; Call Procedure
00401021 00C 83 C4 04           add     esp, 4          ; Add
00401024 008 B8 01 00 00 00     mov     eax, 1
00401029 008 EB 0F              jmp     short loc_40103A ; Jump
```

Figure 15: InternetGetConnectedState returns 1 and executes this code.

This code construct that occurs in Figure 12 is indicative of an if() statement. A pseudocode representation would look like this:

```
bool isThereAConnection = InternetGetConnectedState(0, 0);

if (isThereAConnection == true)
{
   printf("Success: Internet Connection\n")
}
else
{
   printf("Error 1.1: No Internet\n");
}

return; // Back to main
```

LAB 6-2 Question 2

What is the subroutine located at 0x40117F?

BLUF: printf()

We noticed in Figures 14 and 15 a call to this address which is labeled as sub_40117F. In those figures, we also noticed that a string was pushed onto the stack and then this function was called, giving a big hint that this is a printf() function. To confirm our suspicions, we examine where this function was called by pulling up the xref window (Figure 16) and see what kind of values are pushed onto the stack prior to this subroutine being called. If there are strings or format strings pushed onto the stack prior to it being called, then we can reasonably infer that this subroutine is indeed a printf() function. The first two xrefs have addresses we recognize and analyzed in the previous question.

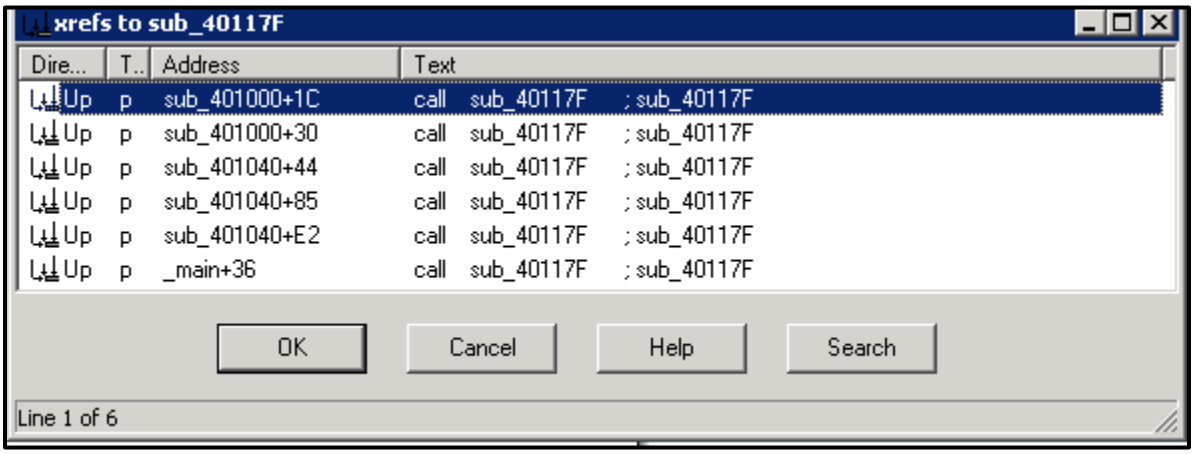

Figure 16: InternetGetConnectedState returns 1 and executes this code.

When we look at the other four xrefs of this subroutine being called in Figures 17- 19, every single time prior to it being called a string is pushed onto the stack. In Figure 20, ECX and a format string are pushed onto the stack, indicating that the ECX value will be placed in the format specifier of the string. This confirms that this subroutine is indeed a printf() function call.

```
0040107F 214 68 A8 70 40 00    push    offset aError2_1FailTo ; "Error 2.1: Fail to OpenUrl\n"
00401084 218 E8 F6 00 00 00    call    sub_40117F       ; Call Procedure
00401089 218 83 C4 04          add     esp, 4           ; Add
0040108C 214 8B 4D F4          mov     ecx, [ebp+hInternet]
0040108F 214 51                push    ecx              ; hInternet
00401090 218 FF 15 B8 60 40 00 call    ds:InternetCloseHandle ; Indirect Call Near Procedure
00401096 214 32 C0             xor     al, al           ; Logical Exclusive OR
00401098 214 E9 8F 00 00 00    jmp     loc_40112C       ; Jump
```

Figure 17: InternetGetConnectedState returns 1 and executes this code.

```
004010C0 214 68 88 70 40 00    push    offset aError2_2FailTo ; "Error 2.2: Fail to ReadFile\n"
004010C5 218 E8 B5 00 00 00    call    sub_40117F       ; Call Procedure
004010CA 218 83 C4 04          add     esp, 4           ; Add
004010CD 214 8B 55 F4          mov     edx, [ebp+hInternet]
004010D0 214 52                push    edx              ; hInternet
004010D1 218 FF 15 B8 60 40 00 call    ds:InternetCloseHandle ; Indirect Call Near Procedure
004010D7 214 8B 45 F0          mov     eax, [ebp+hFile]
004010DA 214 50                push    eax              ; hInternet
004010DB 218 FF 15 B8 60 40 00 call    ds:InternetCloseHandle ; Indirect Call Near Procedure
004010E1 214 32 C0             xor     al, al           ; Logical Exclusive OR
004010E3 214 EB 47             jmp     short loc_40112C ; Jump
```

Figure 18: InternetGetConnectedState returns 1 and executes this code.

```
0040111D
0040111D                       loc_40111D:              ; "Error 2.3: Fail to get command\n"
0040111D 214 68 68 70 40 00    push    offset aError2_3FailTo
00401122 218 E8 58 00 00 00    call    sub_40117F       ; Call Procedure
00401127 218 83 C4 04          add     esp, 4           ; Add
0040112A 214 32 C0             xor     al, al           ; Logical Exclusive OR
```

Figure 19: InternetGetConnectedState returns 1 and executes this code.

```
0040115C
0040115C                       loc_40115C:              ; Move with Sign-Extend
0040115C 00C 0F BE 4D F8       movsx   ecx, [ebp+var_8]
00401160 00C 51                push    ecx
00401161 010 68 10 71 40 00    push    offset aSuccessParsedC ; "Success: Parsed command is %c\n"
00401166 014 E8 14 00 00 00    call    sub_40117F       ; Call Procedure
0040116B 014 83 C4 08          add     esp, 8           ; Add
0040116E 00C 68 60 EA 00 00    push    0EA60h           ; dwMilliseconds
00401173 010 FF 15 00 60 40 00 call    ds:Sleep         ; Indirect Call Near Procedure
00401179 00C 33 C0             xor     eax, eax         ; Logical Exclusive OR
```

Figure 20: InternetGetConnectedState returns 1 and executes this code.

LAB 6-2 Question 3

What does the second subroutine called by main do?

BLUF: It parses an HTML comment from the beginning of the page at domain http://www.practicalmalwareanalysis.com/.

The second subroutine called by main is sub_401040 with the call located at 0x00401148. This function will be called due to the "jnz" instruction after comparing EAX to 0, where the value stored in EAX is 1 or 0 due to sub_401000 being called (analyzed in Question 1). The zero flag will be set to 1 (the comparison operation equals 0) if there is NOT an internet connection, meaning that if there IS an internet connection, then the code will execute sub_401040. This is another example of an if() statement and read as follows in pseudocode:

```
bool isThereAConnection = sub_401000();

// cmp eax, 0
if (isThereAConnection == true)  // zf = 0
{
   // jnz short loc_401148
   sub_401040();
}
```

Figure 21 shows this operating taking place.

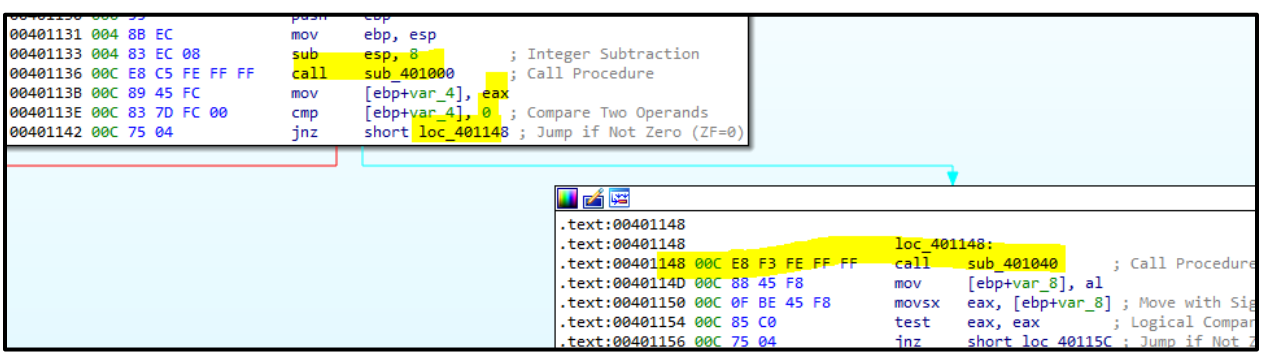

Figure 21: _main jnz instruction.

Examining the beginning portion of subroutine 401040 located at 0x00401040, we first see stack management operations followed by pushing the decimal vale of 0 and the string “Internet Explorer 7.5/pma” onto the stack (Figure 22). The subroutine then calls the external function “InternetOpenA” (the documentation for this function can be found here). The documentation states that the first parameter is lpszAgent, a “pointer to a null-terminated string that specifies the name of the application or entity calling the WinINet functions.” Therefore, the program will use Internet Explorer version 7.5 as the user agent for other WinINet functions and stores it onto the stack.

```
.text:00401040
.text:00401040 000 55                push    ebp
.text:00401041 004 8B EC             mov     ebp, esp
.text:00401043 004 81 EC 10 02 00 00 sub     esp, 210h       ; Integer Subtraction
.text:00401049 214 6A 00             push    0               ; dwFlags
.text:0040104B 218 6A 00             push    0               ; lpszProxyBypass
.text:0040104D 21C 6A 00             push    0               ; lpszProxy
.text:0040104F 220 6A 00             push    0               ; dwAccessType
.text:00401051 224 68 F4 70 40 00    push    offset szAgent  ; "Internet Explorer 7.5/pma"
.text:00401056 228 FF 15 C4 60 40 00 call    ds:InternetOpenA ; Indirect Call Near Procedure
.text:0040105C 214 89 45 F4          mov     [ebp+hInternet], eax
.text:0040105F 214 6A 00             push    0               ; dwContext
.text:00401061 218 6A 00             push    0               ; dwFlags
.text:00401063 21C 6A 00             push    0               ; dwHeadersLength
.text:00401065 220 6A 00             push    0               ; lpszHeaders
.text:00401067 224 68 C4 70 40 00    push    offset szUrl    ; "http://www.practicalmalwareanalysis.com"...
.text:0040106C 228 8B 45 F4          mov     eax, [ebp+hInternet]
.text:0040106F 228 50                push    eax             ; hInternet
.text:00401070 22C FF 15 B4 60 40 00 call    ds:InternetOpenUrlA ; Indirect Call Near Procedure
.text:00401076 214 89 45 F0          mov     [ebp+hFile], eax
.text:00401079 214 83 7D F0 00       cmp     [ebp+hFile], 0  ; Compare Two Operands
.text:0040107D 214 75 1E             jnz     short loc_40109D ; Jump if Not Zero (ZF=0)
```

Figure 22: Opening code for sub_401040.

Following the call to InternetOpenA, we see more values being pushed onto the stack including the string “http://www.practicalmalwareanalysis.com/cc.htm” stored within the variable szUrl, whose full string can be seen in Figure 23. We also see the return value from InternetOpenA which was stored into the stack at [ebp+hInternet] and is moved into EAX, then pushed onto the stack prior to the call to InternetOpenUrlA.

```
.data:004070C4      szUrl           db 'http://www.practicalmalwareanalysis.com/cc.htm',0
.data:004070C4                                                  ; DATA XREF: sub_401040+27↑o
.data:004070F3                      align 4
.data:004070F4      ; CHAR szAgent[]
```

Figure 23: Full string for variable szUrl

The documentation for the InternetOpenUrlA shows 'hInternet' as the first parameter and lpszUrl as the second, meaning the variable string within szUrl equals the lpszUrl parameter (syntax can be seen in Figure 24). The InternetOpenUrlA function "opens a resource specified by a complete FTP or HTTP URL." The documentation shows the parameters defined as follows:

- hInternet: The handle to the current Internet session. The handle must have been returned by a previous call to InternetOpen.
- lpszUrl: A pointer to a null-terminated string variable that specifies the URL to begin reading. Only URLs beginning with ftp:, http:, or https: are supported.
- lpszHeaders: A pointer to a null-terminated string that specifies the headers to be sent to the HTTP server. For more information, see the description of the lpszHeaders parameter in the HttpSendRequest function.
- dwHeadersLength: The size of the additional headers, in TCHARs. If this parameter is -1L and lpszHeaders is not NULL, lpszHeaders is assumed to be zero-terminated (ASCIIZ) and the length is calculated.
- dwFlags: Multiple values can be set to perform different actions.
- dwContext: A pointer to a variable that specifies the application-defined value that is passed, along with the returned handle, to any callback functions.

```
C++                                                    Copy

HINTERNET InternetOpenUrlA(
  [in] HINTERNET hInternet,
  [in] LPCSTR    lpszUrl,
  [in] LPCSTR    lpszHeaders,
  [in] DWORD     dwHeadersLength,
  [in] DWORD     dwFlags,
  [in] DWORD_PTR dwContext
);
```

Figure 24: Syntax for InternetOpenUrlA.

Therefore, we can know that based on the push values for this function, we can imagine it being called like this:

```
hInternet = InternetOpenA(szAgent, 0, 0, 0, 0);
hFile = InternetOpenUrlA(hInternet, szUrl, 0, 0, 0, 0);
```

The zeroes in InternetOpenUrlA represent that no headers are sent to the HTTP server, the length of the headers is 0, no flags are set, and no application-defined values are passed. We then see in Figure 22 that the return value of InternetOpenUrlA, now in EAX, is moved onto the stack and compared to 0. If the return value of InternetOpenUrlA is NULL, then the zero flag will be set and we see in Figure 17 the string printed to the screen is it fails. If the URL is successfully opened, we see the next commands in Figure 25. This figure shows a call to InternetReadFile (documentation here) which will read data from a handle, which is the hFile variable returned from the InternetOpenUrlA function call. InternetReadFile is going to read 0x200 bytes (decimal 512), store it into the buffer pushed at 0x004010AC, and store the number of bytes it read into EDX. The function returns a true or false value and we see the comparison occur at 0x0047010BA. If it did not read any information, it prints the error message seen in Figure 18.

```
.text:0040109D
.text:0040109D                          loc_40109D:
.text:0040109D 214 8D 55 F8             lea     edx, [ebp+dwNumberOfBytesRead] ; Load Effective Address
.text:004010A0 214 52                   push    edx             ; lpdwNumberOfBytesRead
.text:004010A1 218 68 00 02 00 00       push    200h            ; dwNumberOfBytesToRead
.text:004010A6 21C 8D 85 F0 FD FF FF lea     eax, [ebp+Buffer] ; Load Effective Address
.text:004010AC 21C 50                   push    eax             ; lpBuffer
.text:004010AD 220 8B 4D F0             mov     ecx, [ebp+hFile]
.text:004010B0 220 51                   push    ecx             ; hFile
.text:004010B1 224 FF 15 BC 60 40 00 call    ds:InternetReadFile ; Indirect Call Near Procedure
.text:004010B7 214 89 45 FC             mov     [ebp+var_4], eax
.text:004010BA 214 83 7D FC 00          cmp     [ebp+var_4], 0  ; Compare Two Operands
.text:004010BE 214 75 25                jnz     short loc_4010E5 ; Jump if Not Zero (ZF=0)
```

Figure 25: Syntax for InternetOpenUrlA.

To summarize what has taken place until this point, InternetReadFile will read the web page opened by InternetOpenUrlA with the user agent returned from InternetOpenA. The information read from the web page, not exceeding 512 bytes, will be stored in the buffer. If reading the web page is unsuccessful, the function terminates.

Upon successful reading of the web page, we then see the following code in Figure 26 being executed (graph representation in Figure 27. Note in Figure 27, we see a portion of Figure 19 in the lower-right-hand corner).

```
.text:004010E5
.text:004010E5
.text:004010E5      loc_4010E5:                             ; CODE XREF: sub_401040+7E↑j
.text:004010E5 214                  movsx   ecx, [ebp+Buffer] ; Move with Sign-Extend
.text:004010EC 214                  cmp     ecx, 3Ch ; '<'  ; Compare Two Operands
.text:004010EF 214                  jnz     short loc_40111D ; Jump if Not Zero (ZF=0)
.text:004010F1 214                  movsx   edx, [ebp+var_20F] ; Move with Sign-Extend
.text:004010F8 214                  cmp     edx, 21h ; '!'  ; Compare Two Operands
.text:004010FB 214                  jnz     short loc_40111D ; Jump if Not Zero (ZF=0)
.text:004010FD 214                  movsx   eax, [ebp+var_20E] ; Move with Sign-Extend
.text:00401104 214                  cmp     eax, 2Dh ; '-'  ; Compare Two Operands
.text:00401107 214                  jnz     short loc_40111D ; Jump if Not Zero (ZF=0)
.text:00401109 214                  movsx   ecx, [ebp+var_20D] ; Move with Sign-Extend
.text:00401110 214                  cmp     ecx, 2Dh ; '-'  ; Compare Two Operands
.text:00401113 214                  jnz     short loc_40111D ; Jump if Not Zero (ZF=0)
.text:00401115 214                  mov     al, [ebp+var_20C]
.text:0040111B 214                  jmp     short loc_40112C ; Jump
```

Figure 26: Next lines of code after successful reading of web page.

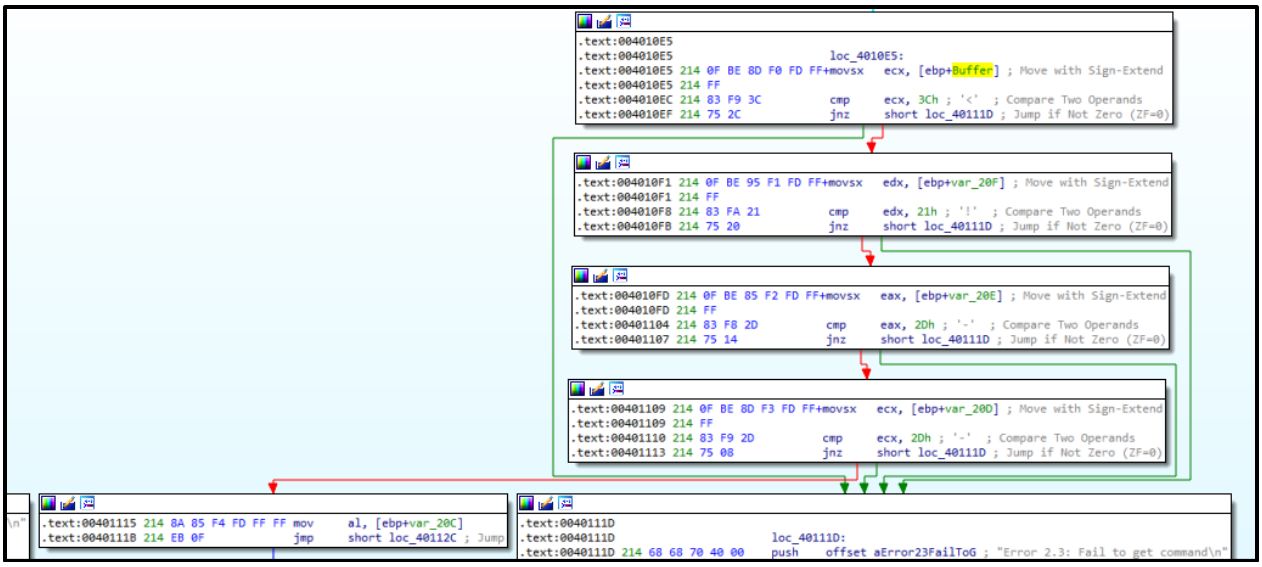

Figure 27: Graph view of Figure 26.

What this code is doing is comparing the buffer at each index one byte at-a-time (var_20F represents EBP+Buffer+1, var_20E is EBP+Buffer+2, etc.). The comparison is done to the characters. <, !, -, and -, combining into the string "<!—". This string represents the start of a comment in HTML language and can only be viewed when looking at the page source HTML code. This can be accomplished within the Google Chrome web browser by pressing F12. An example of this is given in Figure 28, where we can see the string, "<!—end of .mainContainer -->".

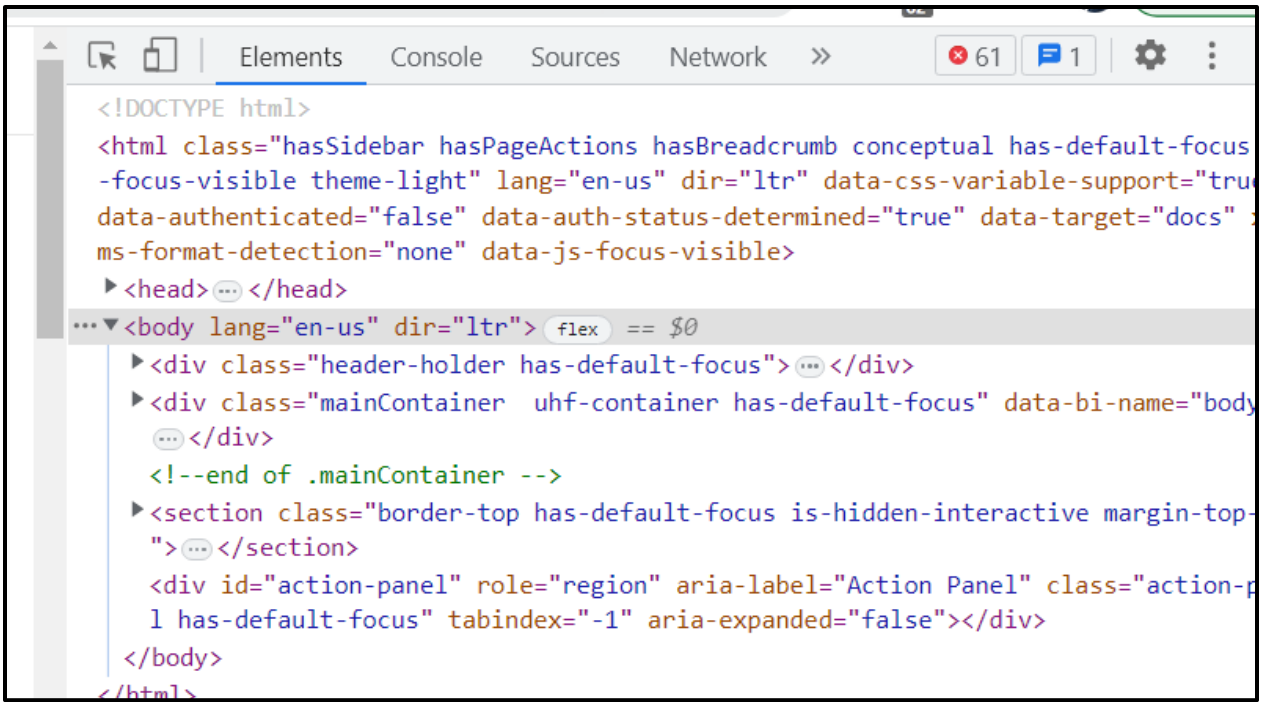

Figure 28: Example of viewing source code.

If any one of these characters does not meet the comparison criteria, it will jump to 0x0040111D and execute the code previously seen in Figure 19. Referring back to Buffer, since it is being compared to HTML strings in order to find a comment, it is reasonable to assume that it is the web page downloaded by InternetReadFile. This section of code attempting to find “<!--" means its looking for a comment string that is at the beginning of the HTML source of the web page. If it finds a comment at the top of the page, it will move the value stored at abp+var_20C into the AL register (Figure 29) and then return to the next line after this subroutine was called.

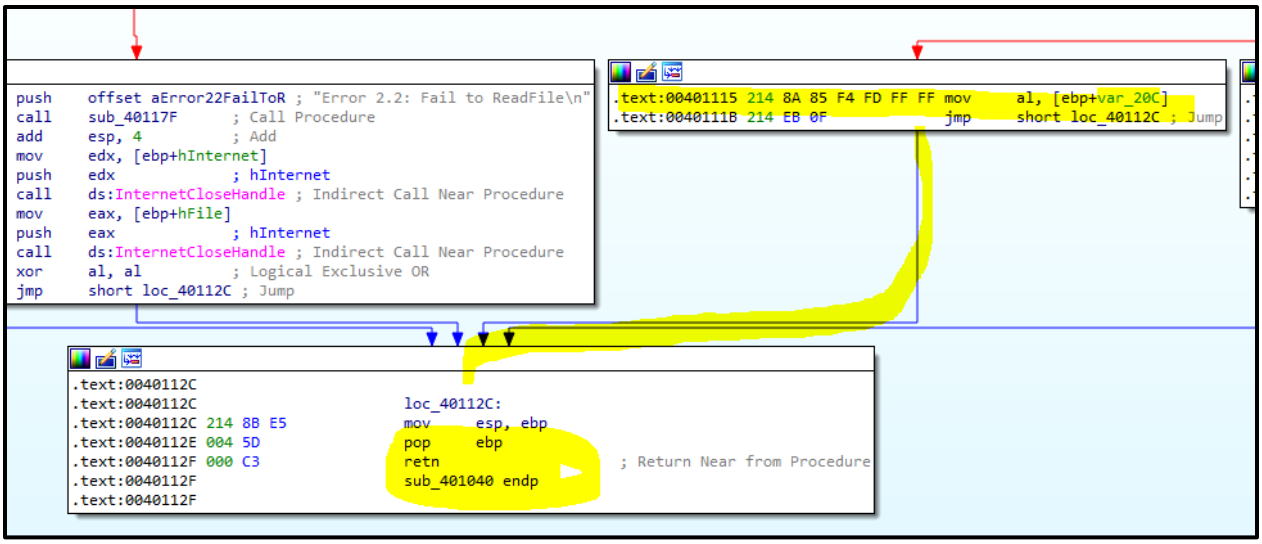

Figure 29: Final block of code if the subroutine is successful.

LAB 6-2 Question 4

What type of code construct is used in this subroutine?

BLUF: A series of if() statements.

This code construct can be considered a series of nested if() statements residing within a for() loop. A pseudocode example of these would look like:

```
char buffer = [];   // Where the characters are stored
char htmlComment = ["<", "!", "-", "-"];
int bytesToRead = 512;
int bytesRead;
bool startOfComment = false;

void aError23FailToG();
int InternetReadFile(hFile, buffer, bytesToRead, bytesRead);

for (int i = 0; i < 4; i++)
{
    if (buffer[i] != htmlComment[i])
    {
        startOfComment = false;
        aError23FailToG();
        break;
    }

    else
    {
        startOfComment = true;
        continue;
    }
}
```

The function “aError23FailToG” is the section of code located at 0x0040111D. This is where the ‘jnz’ commands take place and print the error message in Figure 19. If each of these statements is successful, then the Boolean value of startOfComment will be true and the code at 0x00401115 will begin (Figure 29). However, there is no incrementing of a counter for the ‘int i’ variable

within the code example above. So, a for() loop is not the proper code construct for this subroutine.

This code could is simply a series of if() statements that look like the following:

```
char buffer = [];   // Where the characters are stored
int bytesToRead = 512;
int bytesRead;
bool startOfComment = true;

void aError23FailToG();
int InternetReadFile(hFile, buffer, bytesToRead, bytesRead);

if (buffer[0] != "<")
{
    startOfComment = false;
    aError23FailToG();
}

if(buffer[1] != "!")
{
    startOfComment = false;
    aError23FailToG();
}

if((buffer[2] || buffer[3]) != "-" )
{
    startOfComment = false;
    aError23FailToG();
}
```

These examples are not precise in how the code is executed, but provide a simplified representation outside of the disassembly code. The variable type that the buffer is stored into (an "out" value from the InternetReadFile function) is an array of characters. The code simply indexes through the char array and compares it to the ASCII character. However, the assembly code representation of these ASCII characters are in hexadecimal form. Each of the function

calls to aError23FailToG() within the code example above are equivalent to the ‘jnz’ instructions analyzed in Question 3.

LAB 6-2 Question 5

Are there any network-based indicators for this program?

BLUF: Yes. http://www.practicalmalwareanalysis.com/cc.htm.

The network-based indicators were discussed extensively in Question 3. Following the call to InternetOpenA at 0x00401056, we see values being pushed onto the stack including the string "http://www.practicalmalwareanalysis.com/cc.htm" stored within the variable szUrl, whose full string can be seen in Figure 23. We also see the return value from InternetOpenA which was stored into the stack at [ebp+hInternet] and is moved into EAX, then pushed onto the stack prior to the call to InternetOpenUrlA.

The documentation for the InternetOpenUrlA shows 'hInternet' as the first parameter and lpszUrl as the second, meaning the variable string within szUrl equals the lpszUrl parameter (syntax can be seen in Figure 24). The InternetOpenUrlA function "opens a resource specified by a complete FTP or HTTP URL."

We also know that when examining the beginning portion of subroutine 401040 located at 0x00401040, we see push instructions for decimal vale of 0 and the string "Internet Explorer 7.5/pma" onto the stack prior to calling the external function InternetOpenA (Figure 22).The documentation for this function can be found here. The documentation states that the first parameter is lpszAgent, a "pointer to a null-terminated string that specifies the name of the application or entity calling the WinINet functions." Therefore, the program will use Internet Explorer version 7.5 as the user agent for other WinINet functions and stores it onto the stack.

With this information, we can therefore be certain that one network-based indicator is the querying of the domain “http://www.practicalmalwareanalysis.com/cc.htm”. Although the use of the program Internet Explorer 7.5/pma would be considered a host-based indicator, the combination of Internet Explorer 7.5/pma with the domain point to malware being executed on a machine.

LAB 6-2 Question 6

What is the purpose of this malware?

To summarize the facts gathered in the previous questions regarding the malware's behavior:

- It first checks for an active internet connection within sub_401000. This subroutine uses the external function InternetGetConnectedState" at 0x00401008 which checks for an active internet connection (Figure 12, Question 1).
- If an active internet connection exists, sub_401040 is called. The subroutine then calls the external function "InternetOpenA" with the user agent "Internet Explorer 7.5/pma" (Figure 22, Question 3)
- If InternetOpenA is successful, InternetOpenUrlA is called to open the domain http://www.practicalmalwareanalysis.com/cc.htm. (Figure 22, Question 3).
- If InternetOpenUrlA function is successful, the function InternetReadFile is then called on the return value to read 512 bytes (Figure 25, Question 3).
- The return value of InternetReadFile is the HTML data from the domain opened in InternetOpenUrlA. The malware then checks to see if the domain has a comment string at the beginning of the HTML data (Figures 26-28, Question 3).
- If an HTML comment string is detected at the top of the HTML data on the web page, the value stored in the stack base pointer + var_20C will be moved into the AL register and returned to _main.

Taking these facts into consideration, this malware most likely hides additional commands within the HTML comment string that it gained from calling InternetReadFile on the domain.

Since HTML comments are hidden when viewed through a web browser, this code will make it appear to the user that they are visiting a legitimate website while the malware executes further code. Specifically, we see in Figure 20 that upon successful reading of the code, the malware goes to sleep for 0xEA60 milliseconds (1 minute).

The domain is therefore the C2 node with hidden instructions for the malware embedded inside of the HTML comment located at the top of the page. It is unclear as to what this code might be but that is the point: the malware creator has offshored more commands to a website that the malware accesses and the user is none the wiser as the domain is a legitimate web page. This is a method and evidence of obfuscation.

LAB 6-3

- LAB06-03.exe : 3f8e2b945deba235fa4888682bd0d640 (Figure 30)

Figure 30: Virus Total MD5 Hash for file Lab06-03.exe.

Virus Total found 43 of 70 matching security vendor signatures for a generic trojan (Figure 31) and has a compilation timestamp of 2011-02-03 at 15:14:16 UTC (Figure 32).

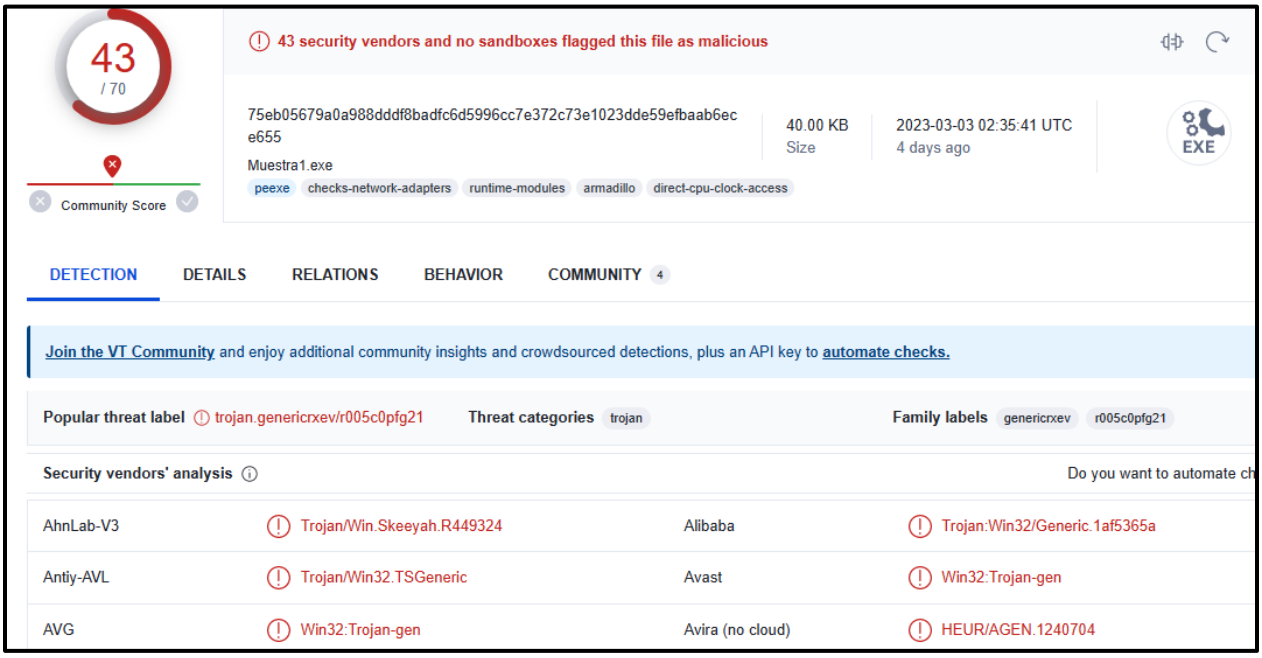

Figure 31: Virus Total Signatures for file Lab06-03.exe.

Header	
Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2011-02-03 15:14:16 UTC
Entry Point	4770
Contained Sections	3

Figure 32: Virus Total Compilation Timestamp for file Lab06-03.exe.

The file appears to import three dynamic linked libraries: advapi32, kerenel32, and wininet. Kernel32.dll indicates that it has the capability to access and modify the core OS functions. Wininet.dll shows that it imports and implements functions related to networking protocols. Advapi32 allows the program to manipulate the service manager and registry (Figure 33).

Imports

- ADVAPI32.dll
- KERNEL32.dll
- WININET.dll

Figure 33: Virus Total Imports for file Lab06-03.exe.

Much like Lab06-02.exe, this file has networking protocol functions that are supported within wininet.dll and may be related to detected network connections identified by Virus Total for Lab06-03.exe. There are multiple HTTP requests to practicalmalwareanalysis.com and we see the same domain identified in Lab06-03.exe (Figure 34). There is lots of IP traffic with potential Transport Layer Security (TLS) implemented for practicalmalwareanalysis.com (Figure 35).

Figure 34: Virus Total domain calls for file Lab06-03.exe.

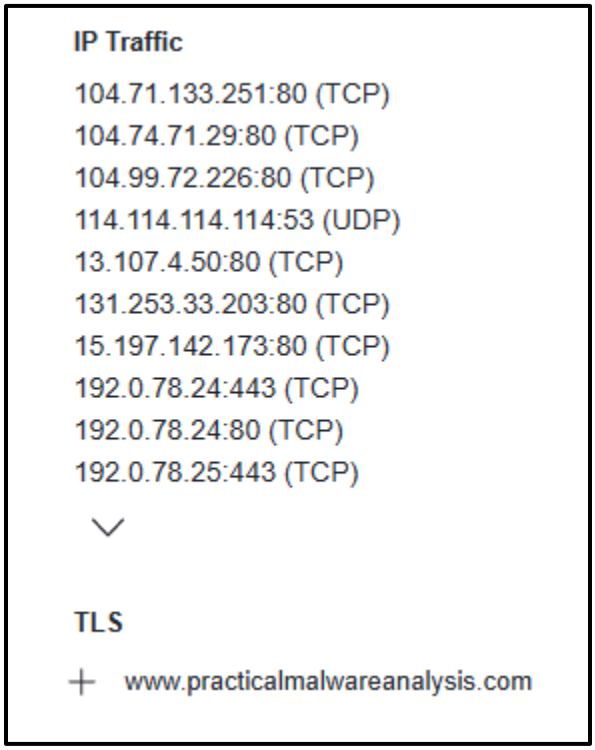

Figure 35: Virus Total IP Traffic for file Lab06-03.exe.

Virus Total also reports that the file has behavior of persistence, privilege escalation, and defense evasion (Figure 36). We also see in Figure 37 the C2 behaviors of downloading files using HTTP and using HTTPS for encrypted channels, most likely in reference for the TLS networking behavior identified in Figure 35. Based on these findings, this malware is most likely a generic

trojan as identified in Figure 2 and uses HTTP and HTTPS (ports 80 and 443) to download additional packages onto the infected machine when the file is run by the user.

Persistence TA0003

Registry Run Keys / Startup Folder T1547.001
ⓘ Persist via Run registry key

Privilege Escalation TA0004

Process Injection T1055
ⓘ Spawns processes

Privilege Escalation TA0004

Registry Run Keys / Startup Folder T1547.001
ⓘ Persist via Run registry key

Defense Evasion TA0005

Masquerading T1036
ⓘ Creates files inside the user directory

Process Injection T1055
ⓘ Spawns processes

Figure 36: Virus Total Behavior for file Lab06-03.exe.

Command and Control TA0011

Application Layer Protocol T1071
ⓘ Uses HTTPS
ⓘ Performs DNS lookups
ⓘ Downloads files from webservers via HTTP
ⓘ Tries to download or post to a non-existing http route (HTTP/1.1 404 Not Found / 503 Service Unavailable)

Non-Application Layer Protocol T1095
ⓘ Performs DNS lookups
ⓘ Downloads files from webservers via HTTP
ⓘ Tries to download or post to a non-existing http route (HTTP/1.1 404 Not Found / 503 Service Unavailable)

Ingress Tool Transfer T1105
ⓘ Downloads files from webservers via HTTP
ⓘ Some HTTP requests failed (404). It is likely the sample will exhibit less behavior
ⓘ Tries to download or post to a non-existing http route (HTTP/1.1 404 Not Found / 503 Service Unavailable)

Encrypted Channel T1573
ⓘ Uses HTTPS
ⓘ Uses HTTPS for network communication, use the SSL MITM Proxy cookbook for further analysis

Figure 37: Virus Total C2 for file Lab06-03.exe.

LAB 6-3

LAB 6-3 Question 1

Compare the calls in main to Lab 6-2's main method. What is the new function called from main?

BLUF: sub_401130 exists in Lab06-03.exe but not in Lab06-02.exe.

In Figure 38, we see the _main function of Lab06-03.exe begin at 0x00401210. We can compare this to Figure 9 which is Lab06-02.exe's _main function and relabeled below as Figure 39 (Figure 9 and Figure 39 are the same). In both _main examples in the topmost block of code in graph view, there are the exact same lines of code. We see a call to sub_401000 which, as analyzed in Lab06-02 Question 1, was determined to be a function that determines the infected machine's internet connectivity by using the imported function InternetGetConnectedState.

```
00401210
00401210
00401210                        ; Attributes: bp-based frame
00401210
00401210                        ; int __cdecl main(int argc,const char **argv,const char *envp)
00401210                        _main proc near
00401210
00401210                        var_8= byte ptr -8
00401210                        var_4= dword ptr -4
00401210                        argc= dword ptr  8
00401210                        argv= dword ptr  0Ch
00401210                        envp= dword ptr  10h
00401210
00401210 000 55                 push    ebp
00401211 004 8B EC              mov     ebp, esp
00401213 004 83 EC 08           sub     esp, 8          ; Integer Subtraction
00401216 00C E8 E5 FD FF FF     call    sub_401000      ; Call Procedure
0040121B 00C 89 45 FC           mov     [ebp+var_4], eax
0040121E 00C 83 7D FC 00        cmp     [ebp+var_4], 0  ; Compare Two Operands
00401222 00C 75 04              jnz     short loc_401228 ; Jump if Not Zero (ZF=0)
```

Figure 38: Beginning code of Lab06-03.exe _main function.

```
00401130
00401130
00401130                        ; Attributes: bp-based frame
00401130
00401130                        ; int __cdecl main(int argc,const char **argv,const char *envp)
00401130                        _main proc near
00401130
00401130                        var_8= byte ptr -8
00401130                        var_4= dword ptr -4
00401130                        argc= dword ptr  8
00401130                        argv= dword ptr  0Ch
00401130                        envp= dword ptr  10h
00401130
00401130 000 55                 push    ebp
00401131 004 8B EC              mov     ebp, esp
00401133 004 83 EC 08           sub     esp, 8          ; Integer Subtraction
00401136 00C E8 C5 FE FF FF     call    sub_401000      ; Call Procedure
0040113B 00C 89 45 FC           mov     [ebp+var_4], eax
0040113E 00C 83 7D FC 00        cmp     [ebp+var_4], 0  ; Compare Two Operands
00401142 00C 75 04              jnz     short loc_401148 ; Jump if Not Zero (ZF=0)
```

Figure 39: Lab06-02.exe _main function and first subroutine call.

At a glance, the graphical view of Lab06-03.exe's _main function (Figure 40) looks strikingly-similar to Lab06-02.exe (Figure 41). However, we notice that the block of code that extends furthest to the right (circled in both) in Lab06-03.exe looks larger than Lab06-02.exe. Note that the two blocks that are two lines each perform identical operations in both .exe files save for the memory location that they jump to.

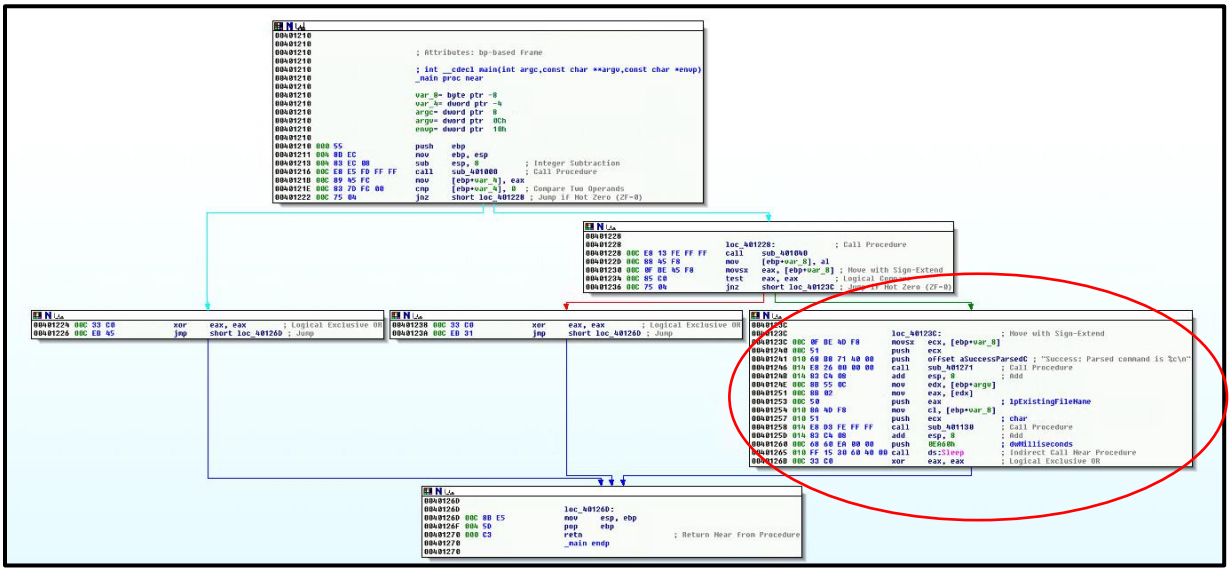

Figure 40: Lab06-03.exe _main function graph view.

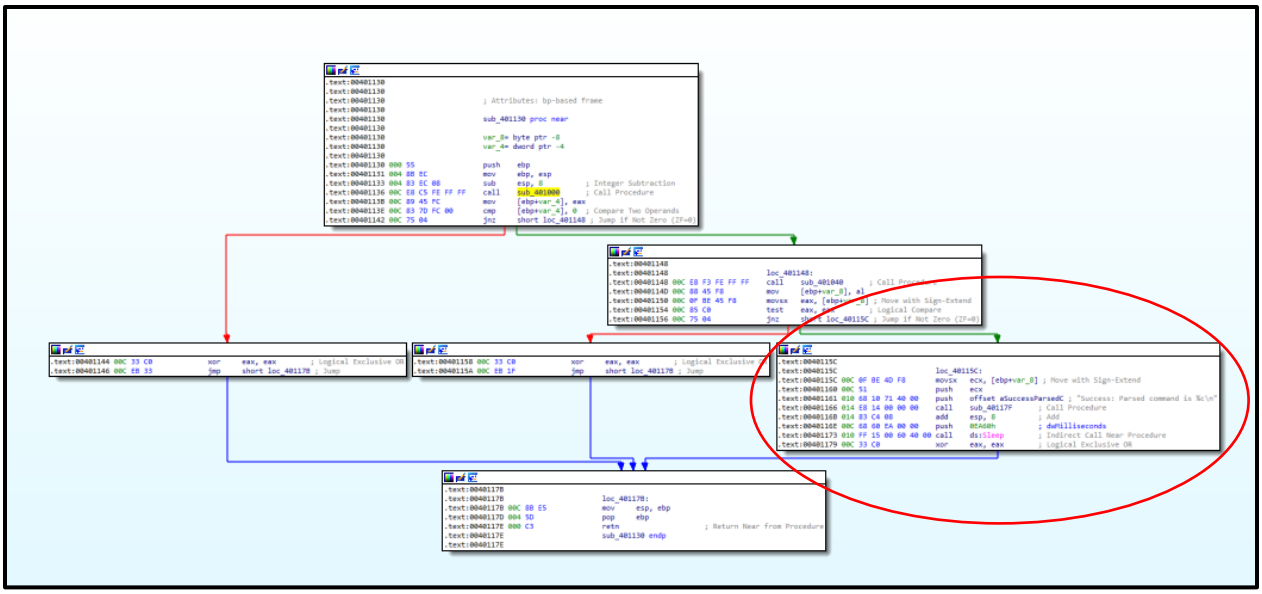

Figure 41: Lab06-02.exe _main function graph view.

The next block of code down from the top block of code in Lab06-03.exe (Figure 42) shows the same call to sub_401040 like Lab06-02.exe (Figure 43) and the same instructions. Subroutine 401040 in both labs is identical and its functionality can be reviewed in detail in section Lab06-02 Question 3 of this report. Click here to view it. To summarize: the subroutine downloads a web page and parses an HTML comment.

```
00401228
00401228                        loc_401228:             ; Call Procedure
00401228 00C E8 13 FE FF FF     call    sub_401040
0040122D 00C 88 45 F8           mov     [ebp+var_8], al
00401230 00C 0F BE 45 F8        movsx   eax, [ebp+var_8] ; Move with Sign-Extend
00401234 00C 85 C0              test    eax, eax        ; Logical Compare
00401236 00C 75 04              jnz     short loc_40123C ; Jump if Not Zero (ZF=0)
```

Figure 42: Lab06-03.exe _main next block of code.

```
.text:00401148
.text:00401148                  loc_401148:
.text:00401148 00C E8 F3 FE FF FF   call    sub_401040      ; Call Procedure
.text:0040114D 00C 88 45 F8         mov     [ebp+var_8], al
.text:00401150 00C 0F BE 45 F8      movsx   eax, [ebp+var_8] ; Move with Sign-Extend
.text:00401154 00C 85 C0            test    eax, eax        ; Logical Compare
.text:00401156 00C 75 04            jnz     short loc_40115C ; Jump if Not Zero (ZF=0)
```

Figure 43: Lab06-03.exe _main next block of code.

Next, we compare and contrast the block of code that occurs after Figures 42 and 43 where they perform the 'jnz' instruction at 0x00401236 ad 0x00401156, respectively. These were the blocks of code circled in red in figures 40 and 41. We now see two different calls to subroutines. In Figure 45, we see sub_40117F which we determined to be a printf() statement. In Figure 44, we see a call to sub_401271 and then a call to sub_401130 for Lab06-03.exe. Since the 'push' instructions are identical to sub_40117F, we can deduce that this is also a printf() subroutine. This is confirmed in Figures 46 and 47 by examining the identical code.

```
0040123C
0040123C                        loc_40123C:             ; Move with Sign-Extend
0040123C 00C 0F BE 4D F8        movsx   ecx, [ebp+var_8]
00401240 00C 51                 push    ecx
00401241 010 68 B8 71 40 00     push    offset aSuccessParsedC ; "Success: Parsed command is %c\n"
00401246 014 E8 26 00 00 00     call    sub_401271      ; Call Procedure
0040124B 014 83 C4 08           add     esp, 8          ; Add
0040124E 00C 8B 55 0C           mov     edx, [ebp+argv]
00401251 00C 8B 02              mov     eax, [edx]
00401253 00C 50                 push    eax             ; lpExistingFileName
00401254 010 8A 4D F8           mov     cl, [ebp+var_8]
00401257 010 51                 push    ecx             ; char
00401258 014 E8 D3 FE FF FF     call    sub_401130      ; Call Procedure
0040125D 014 83 C4 08           add     esp, 8          ; Add
00401260 00C 68 60 EA 00 00     push    0EA60h          ; dwMilliseconds
00401265 010 FF 15 30 60 40 00  call    ds:Sleep        ; Indirect Call Near Procedure
0040126B 00C 33 C0              xor     eax, eax        ; Logical Exclusive OR
```

Figure 44: Lab06-03.exe _main next block of code after figure 41.

```
.text:0040115C
.text:0040115C                  loc_40115C:
.text:0040115C 00C 0F BE 4D F8  movsx   ecx, [ebp+var_8] ; Move with Sign-Extend
.text:00401160 00C 51           push    ecx
.text:00401161 010 68 10 71 40 00  push  offset aSuccessParsedC ; "Success: Parsed command is %c\n"
.text:00401166 014 E8 14 00 00 00  call  sub_40117F      ; Call Procedure
.text:0040116B 014 83 C4 08     add     esp, 8          ; Add
.text:0040116E 00C 68 60 EA 00 00  push  0EA60h          ; dwMilliseconds
.text:00401173 010 FF 15 00 60 40 00 call ds:Sleep       ; Indirect Call Near Procedure
.text:00401179 00C 33 C0        xor     eax, eax        ; Logical Exclusive OR
```

Figure 45: Lab06-02.exe _main next block of code after figure 42.

```
00401271
00401271
00401271
00401271                        sub_401271 proc near
00401271
00401271                        arg_0= dword ptr  0Ch
00401271                        arg_4= dword ptr  10h
00401271
00401271 000 53                 push    ebx
00401272 004 56                 push    esi
00401273 008 BE 08 72 40 00     mov     esi, offset unk_407208
00401278 008 57                 push    edi
00401279 00C 56                 push    esi
0040127A 010 E8 4B 01 00 00     call    __stbuf         ; Call Procedure
0040127F 010 8B F8              mov     edi, eax
00401281 010 8D 44 24 18        lea     eax, [esp+8+arg_4] ; Load Effective Address
00401285 010 50                 push    eax             ; int
00401286 014 FF 74 24 18        push    [esp+0Ch+arg_0] ; int
0040128A 018 56                 push    esi             ; FILE *
0040128B 01C E8 04 02 00 00     call    sub_401494      ; Call Procedure
00401290 01C 56                 push    esi
00401291 020 57                 push    edi
00401292 024 8B D8              mov     ebx, eax
00401294 024 E8 BE 01 00 00     call    __ftbuf         ; Call Procedure
00401299 024 83 C4 18           add     esp, 18h        ; Add
0040129C 00C 8B C3              mov     eax, ebx
0040129E 00C 5F                 pop     edi
0040129F 008 5E                 pop     esi
004012A0 004 5B                 pop     ebx
004012A1 000 C3                 retn                    ; Return Near from Procedure
004012A1                        sub_401271 endp
004012A1
```

Figure 46: Lab06-03.exe sub_401271 code.

```
.text:0040117F
.text:0040117F
.text:0040117F
.text:0040117F                      sub_40117F proc near
.text:0040117F
.text:0040117F                      arg_0= dword ptr  4
.text:0040117F                      arg_4= byte ptr  8
.text:0040117F
.text:0040117F 000 53               push    ebx
.text:00401180 004 56               push    esi
.text:00401181 008 BE 60 71 40 00   mov     esi, offset unk_407160
.text:00401186 008 57               push    edi
.text:00401187 00C 56               push    esi
.text:00401188 010 E8 4B 01 00 00   call    sub_4012D8      ; Call Procedure
.text:0040118D 010 8B F8            mov     edi, eax
.text:0040118F 010 8D 44 24 18      lea     eax, [esp+10h+arg_4] ; Load Effective Address
.text:00401193 010 50               push    eax
.text:00401194 014 FF 74 24 18      push    [esp+14h+arg_0]
.text:00401198 018 56               push    esi
.text:00401199 01C E8 04 02 00 00   call    sub_4013A2      ; Call Procedure
.text:0040119E 01C 56               push    esi
.text:0040119F 020 57               push    edi
.text:004011A0 024 8B D8            mov     ebx, eax
.text:004011A2 024 E8 BE 01 00 00   call    sub_401365      ; Call Procedure
.text:004011A7 024 83 C4 18         add     esp, 18h        ; Add
.text:004011AA 00C 8B C3            mov     eax, ebx
.text:004011AC 00C 5F               pop     edi
.text:004011AD 008 5E               pop     esi
.text:004011AE 004 5B               pop     ebx
.text:004011AF 000 C3               retn                    ; Return Near from Procedure
.text:004011AF                      sub_40117F endp
.text:004011AF
```

Figure 47: Lab06-02.exe sub_40117F code.

So now we have identified the critical difference between these two _main functions: the existence of sub_401130 in Lab06-03.exe and not in Lab06-02.exe. As the name suggests, the location of this subroutine is at 0x00401130.

LAB 6-3 Question 2

What parameters does this new function take?

To examine the parameters of this function, we can analyze the hints from the push instructions prior to it being called as well as the subroutine itself. In Figure 44, we see that the value stored in the address of EAX was the value stored in EDX. IDA Pro has labeled this value as 'lpExistingFileName'. We then see that a value stored 8 bytes from the stack base pointer EBP was moved into the 8-bit register CL and then ECX was pushed onto the stack and IDA has labeled it as a 'char'. We can expect to see these two values to be within the function parameters in sub_401130.

Once we examine the subroutine itself, we confirm that we see these two labeled values as parameters in addition to a variable named "LPCSTR". This could possibly be an output that this function stores a return value in. We also see at 0x00401136 that [ebp+arg_0] is placed into EAX with a 'movsx' instruction. IDA automatically uses arg_0 as a label to list the last parameter before the call. This is the command character retrieved from the webpage HTML comment from sub_401040. It is then moved into ECX at 0x0040113D and subtracts it with hex value 61 (decimal 97 and lowercase 'a' in ASCII). If the character was 'a', then ECX will equal 0. If it was b, ECX will equal 1, and so on.

This character is then moved back into var_8 at 0x00401143 and compared with decimal 4 immediately after. If the comparison returns a value greater than 4, it will jump to 0x004011F2 (the 'ja' instruction).

```
00401130
00401130
00401130                        ; Attributes: bp-based frame
00401130
00401130                        ; int __cdecl sub_401130(char,LPCSTR lpExistingFileName)
00401130                        sub_401130 proc near
00401130
00401130                        var_8= dword ptr -8
00401130                        hKey= dword ptr -4
00401130                        arg_0= byte ptr  8
00401130                        lpExistingFileName= dword ptr  0Ch
00401130
00401130 000 55                 push    ebp
00401131 004 8B EC              mov     ebp, esp
00401133 004 83 EC 08           sub     esp, 8          ; Integer Subtraction
00401136 00C 0F BE 45 08        movsx   eax, [ebp+arg_0] ; Move with Sign-Extend
0040113A 00C 89 45 F8           mov     [ebp+var_8], eax
0040113D 00C 8B 4D F8           mov     ecx, [ebp+var_8]
00401140 00C 83 E9 61           sub     ecx, 61h        ; Integer Subtraction
00401143 00C 89 4D F8           mov     [ebp+var_8], ecx
00401146 00C 83 7D F8 04        cmp     [ebp+var_8], 4  ; Compare Two Operands
0040114A 00C 0F 87 91 00 00 00 ja       loc_4011E1      ; Jump if Above (CF=0 & ZF=0)
```

Figure 48: sub_401130 first block of code.

If the 'ja' instruction is performed, then we see that sub_401271 (the printf() subroutine) is called after an error string is pushed onto the stack (Figure 49). This tells us that any characters other than a, b, c, d, or e will result in an error.

```
004011E1
004011E1                        loc_4011E1:             ; "Error 3.2: Not a valid command provided"...
004011E1 00C 68 10 71 40 00     push    offset aError3_2NotAVa
004011E6 010 E8 86 00 00 00     call    sub_401271      ; Call Procedure
004011EB 010 83 C4 04           add     esp, 4          ; Add
```

Figure 49: sub_401130 error message.

However, if the comparison results in an affirmation that the character is an a, b, c, d, or e, it will move that character into EDX and jump to 0x004011F2 with EDX multiplied by 4. These instructions can be seen in Figure 50 below.

```
.text:00401130 000 55                          push    ebp
.text:00401131 004 8B EC                       mov     ebp, esp
.text:00401133 004 83 EC 08                    sub     esp, 8          ; Integer Subtraction
.text:00401136 00C 0F BE 45 08                 movsx   eax, [ebp+arg_0] ; Move with Sign-Extend
.text:0040113A 00C 89 45 F8                    mov     [ebp+var_8], eax
.text:0040113D 00C 8B 4D F8                    mov     ecx, [ebp+var_8]
.text:00401140 00C 83 E9 61                    sub     ecx, 61h        ; Integer Subtraction
.text:00401143 00C 89 4D F8                    mov     [ebp+var_8], ecx
.text:00401146 00C 83 7D F8 04                 cmp     [ebp+var_8], 4  ; Compare Two Operands
.text:0040114A 00C 0F 87 91 00 00 00           ja      loc_4011E1      ; Jump if Above (CF=0 & ZF=0)
.text:00401150 00C 8B 55 F8                    mov     edx, [ebp+var_8]
.text:00401153 00C FF 24 95 F2 11 40+          jmp     ds:off_4011F2[edx*4] ; Indirect Near Jump
.text:0040115A
```

Figure 50: sub_401130 with jump to 0x004011F2.

At the address of 0x004011F2, we see five different variables that each have memory addresses separated by 4 bytes. This tells us that each of these offsets correspond with a, b, c, d, and e, respectively (Figure 51).

```
.text:004011F2 5A 11 40 00          off_4011F2      dd offset loc_40115A    ; DATA XREF: sub_401130+23↑r
.text:004011F6 6C 11 40 00                          dd offset loc_40116C
.text:004011FA 7F 11 40 00                          dd offset loc_40117F
.text:004011FE 8C 11 40 00                          dd offset loc_40118C
.text:00401202 D4 11 40 00                          dd offset loc_4011D4
.text:00401206 CC CC CC CC CC CC+                   align 10h
.text:00401210
```

Figure 51: 0x004011F2 offsets.

By double-clicking on these variables, we can see that within their locations resides different sets of instructions. An ‘a’ selection will execute code at Figure 52, a ‘b’ selection will execute code at Figure 52, and so on until Figure 55. Each one of these sections ends with a jump instruction to 0x004011EE which clears the stack and returns to the next line of code after this function was called (Figure 56). Explanation of the what each of these figures do will be addressed in Question 4.

```
.text:0040115A
.text:0040115A                      loc_40115A:                             ; DATA XREF: .text:off_4011F2↓o
.text:0040115A 00C 6A 00                        push    0                   ; lpSecurityAttributes
.text:0040115C 010 68 B0 71 40 00               push    offset PathName     ; "C:\\Temp"
.text:00401161 014 FF 15 0C 60 40 00            call    ds:CreateDirectoryA ; Indirect Call Near Procedure
.text:00401167 00C E9 82 00 00 00               jmp     loc_4011EE          ; Jump
.text:0040116C                      ; ---------------------------------------------------------------------------
```

Figure 52: ‘a’ creates a directory.

```
text:0040116C                          ; ---------------------------------------------------------------------------
text:0040116C
text:0040116C                          loc_40116C:                                     ; CODE XREF: sub_401130+23↑j
text:0040116C                                                                          ; DATA XREF: .text:004011F6↓o
text:0040116C 00C 6A 01                                  push    1                     ; bFailIfExists
text:0040116E 010 68 A0 71 40 00                         push    offset Data           ; "C:\\Temp\\cc.exe"
text:00401173 014 8B 45 0C                               mov     eax, [ebp+lpExistingFileName]
text:00401176 014 50                                     push    eax                   ; lpExistingFileName
text:00401177 018 FF 15 14 60 40 00                      call    ds:CopyFileA          ; Indirect Call Near Procedure
text:0040117D 00C EB 6F                                  jmp     short loc_4011EE ; Jump
```

Figure 53: 'b' copies a file.

```
text:0040117F                          ; ---------------------------------------------------------------------------
text:0040117F
text:0040117F                          loc_40117F:                                     ; CODE XREF: sub_401130+23↑j
text:0040117F                                                                          ; DATA XREF: .text:004011FA↓o
text:0040117F 00C 68 A0 71 40 00                         push    offset Data           ; "C:\\Temp\\cc.exe"
text:00401184 010 FF 15 28 60 40 00                      call    ds:DeleteFileA        ; Indirect Call Near Procedure
text:0040118A 00C EB 62                                  jmp     short loc_4011EE ; Jump
text:0040118C                          ; ---------------------------------------------------------------------------
```

Figure 54: 'c' deletes a file.

```
.text:0040118C                         ; ---------------------------------------------------------------------------
.text:0040118C
.text:0040118C                         loc_40118C:                                     ; CODE XREF: sub_401130+23↑j
.text:0040118C                                                                         ; DATA XREF: .text:004011FE↓o
.text:0040118C 00C 8D 4D FC                              lea     ecx, [ebp+hKey]       ; Load Effective Address
.text:0040118F 00C 51                                    push    ecx                   ; phkResult
.text:00401190 010 68 3F 00 0F 00                        push    0F003Fh               ; samDesired
.text:00401195 014 6A 00                                 push    0                     ; ulOptions
.text:00401197 018 68 70 71 40 00                        push    offset SubKey         ; "Software\\Microsoft\\Windows\\CurrentVersi"...
.text:0040119C 01C 68 02 00 00 80                        push    80000002h             ; hKey
.text:004011A1 020 FF 15 04 60 40 00                     call    ds:RegOpenKeyExA ; Indirect Call Near Procedure
.text:004011A7 00C 6A 0F                                 push    0Fh                   ; cbData
.text:004011A9 010 68 A0 71 40 00                        push    offset Data           ; "C:\\Temp\\cc.exe"
.text:004011AE 014 6A 01                                 push    1                     ; dwType
.text:004011B0 018 6A 00                                 push    0                     ; Reserved
.text:004011B2 01C 68 68 71 40 00                        push    offset ValueName ; "Malware"
.text:004011B7 020 8B 55 FC                              mov     edx, [ebp+hKey]
.text:004011BA 020 52                                    push    edx                   ; hKey
.text:004011BB 024 FF 15 00 60 40 00                     call    ds:RegSetValueExA ; Indirect Call Near Procedure
.text:004011C1 00C 85 C0                                 test    eax, eax              ; Logical Compare
.text:004011C3 00C 74 0D                                 jz      short loc_4011D2 ; Jump if Zero (ZF=1)
.text:004011C5 00C 68 3C 71 40 00                        push    offset aError3_1CouldN ; "Error 3.1: Could not set Registry value"...
.text:004011CA 010 E8 A2 00 00 00                        call    sub_401271            ; Call Procedure
.text:004011CF 010 83 C4 04                              add     esp, 4                ; Add
```

Figure 55: 'd' sets a registry key value.

```
.text:004011D4                         ; ---------------------------------------------------------------------------
.text:004011D4
.text:004011D4                         loc_4011D4:                                     ; CODE XREF: sub_401130+23↑j
.text:004011D4                                                                         ; DATA XREF: .text:00401202↓o
.text:004011D4 00C 68 A0 86 01 00                        push    186A0h                ; dwMilliseconds
.text:004011D9 010 FF 15 30 60 40 00                     call    ds:Sleep              ; Indirect Call Near Procedure
.text:004011DF 00C EB 0D                                 jmp     short loc_4011EE ; Jump
.text:004011E1                         ; ---------------------------------------------------------------------------
```

Figure 56: 'e' makes the program sleep.

```
004011EE
004011EE                               loc_4011EE:
004011EE 00C 8B E5                     mov     esp, ebp
004011F0 004 5D                        pop     ebp
004011F1 000 C3                        retn                            ; Return Near from Procedure
004011F1                               sub_401130 endp
004011F1
```

Figure 57: Endpoint of sub_401130.

LAB 6-3 Question 3

What major code construct does this function contain?

BLUF: Switch statement with a jump table.

Based on Figures 52-56, we can interpret that these are menu options to perform different operations. A common approach to a menu is by using a "switch" statement. An example of this is shown in Figure 58 and taken from a program I created for CSCV 352 at University of Arizona.

```
while ((choice = menu()) != 'q')
{
    switch (choice)
    {

    case 'f':
        FindEmp(pListHead);
        break;

    case 'e':
        EditEmployee(pListHead);
        break;

    case 'r':
        pListHead = RemoveEmployee(pListHead, &count);
        break;

    case 'l':
        PrintList(pListHead, &count);
        break;

    default:
        puts("Switching error");
    }
}
```

Figure 58: Example of a switch statement.

However, the code construct in sub_ 401130 doesn't just utilize a switch statement, it also uses a Jump Table. We know this because in Figure 50, EDX (the character value) is multiplied by 4 in order to reach the desired location to perform the specific operation. This Jump Table is more efficient than using multiple if() statements because it simply performs one operation on one

value to reach the desired location instead of five. Evidence of the Jump Table can be seen in the graph view of sub_401130 in Figure 59 and seeing the similarities of it on Page 126 of Sikorski. Each case is broken down into separate code chunks and all terminate at the box at the very bottom of Figure 59, clear evidence for using switch statements and a Jump Table. Figure 29 also has labels for the chunks of code corresponding to the switch options of a, b, c, d, e, and default.

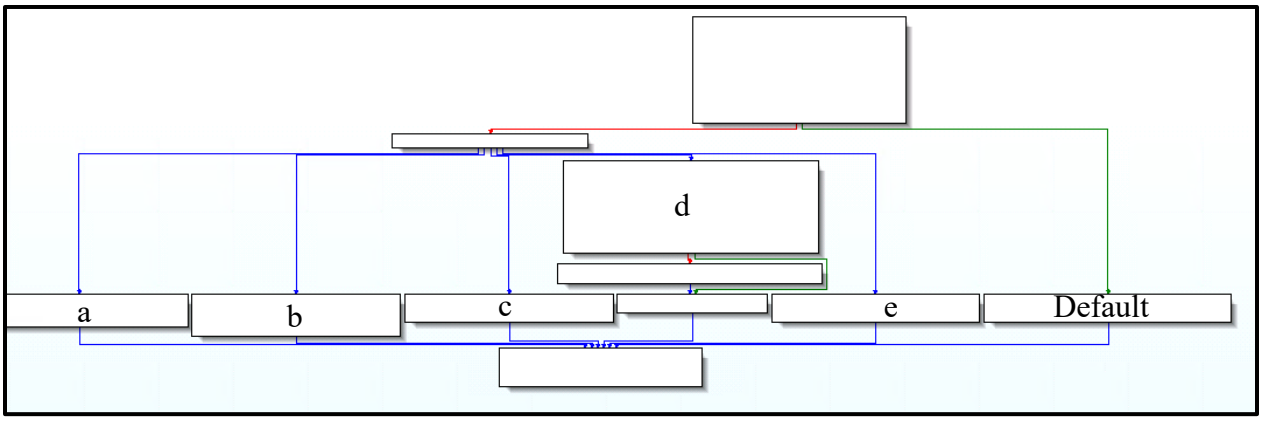

Figure 59: Graph view of sub_401130.

LAB 6-3 Question 4

What can this function do?

Figures 52 through 57 will be used to analyze this function and can be located here.

Figure 52: What occurs corresponding to an 'a' selection. This block executes the external function of "CreateDirectory" after pushing a value of 0 for the parameter lpSecurityAttributes and the path name of C:\\Temp (if it doesn't already exist). Since the value of lpSecurityAttributes is 0, it will have default security. If the directory already exists, it will return the return code, "ERROR_ALREADY_EXISTS". Documentation for this function can be found here and is part of winbase.h.

Figure 53: What occurs corresponding to an 'b' selection. This block executes the external function of "CopyFile". It takes three parameters. The lpExistingFileName parameter is stored in EAX and is pushed onto the stack. This is the source file and corresponding directory that will be copied into the C:\\Temp directory. The lpNewFileName string is the new file name and is titled cc.exe and will be stored in the C:\\Temp directory. We know that cc.exe is malware based on a quick Google search and we can anticipate it to be a type of adware (see here for details). The bFailIfExists parameter is set to 1, meaning that if the file pushed already exists, this function will fail. Documentation for this function can be found here and is part of winbase.h.

Figure 54: What occurs corresponding to an 'c' selection. This block executes the external function of "DeleteFile". It takes one parameter and deletes the file located at the path passed

into it. In this case, it will delete the file cc.exe in the C:\\Temp directory. Documentation for this function can be found here and is part of winbase.h.

Figure 55: What occurs corresponding to an ‘d’ selection. This performs two external functions: RegOpenKeyExA and RegSetValueExA (documentation can be seen by clicking the respective function). We see that the variable SubKey is passed into this function and is defined in the .data section as Software\Microsoft\Windows\CurrentVersion\Run (Figure 60). This Registry Key directory contains commands that will run every time a user will log on. The external function RegSetValueExA is called to set the value in Software\Microsoft\Windows\CurrentVersion\Run\Malware to C:\\Temp\\cc.exe. This is evidence of cc.exe, identified as malware and labeled as such in the \Run registry key, establishing persistence.

```
data:00407170                    ; char SubKey[]
data:00407170 53 6F 66 74 77 61+ SubKey          db 'Software\Microsoft\Windows\CurrentVersion\Run',0
data:00407170 72 65 5C 4D 69 63+                                          ; DATA XREF: sub_401130+67↑o
data:0040719E 00 00                              align 10h
data:004071A0                    ; BYTE Data
```

Figure 60: SubKey definition.

Figure 56: What occurs corresponding to an ‘e’ selection. This simply makes the program sleep for hex 186A0 milliseconds (decimal 100,000), or 100 seconds.

Default: Since switches contain an option of ‘default’ which executes its code if one of the values to switch is not valid, we see a call to sub_401271 (previously identified as a printf() function) which prints an error message stating that a valid command was not provided (Figure 61).

```
004011E1
004011E1                        loc_4011E1:             ; "Error 3.2: Not a valid command provided"...
004011E1 00C 68 10 71 40 00     push    offset aError3_2NotAVa
004011E6 010 E8 86 00 00 00     call    sub_401271      ; Call Procedure
004011EB 010 83 C4 04           add     esp, 4          ; Add
```

Figure 61: Default option for the switch.

LAB 6-3 Question 5

Are there any host-based indicators for this malware?

BLUF: C:\Temp\cc.exe, Software\Microsoft\Windows\CurrentVersion\Run\Malware reg key.

To evaluate the host-based indicators for this malware, we need to do a more comprehensive static and dynamic analysis. We have previously identified a few host-based indicators, such as the existence of cc.exe in the C:\Temp directory (Figures 53 and 54). We also know that the registry key value in the HKLM hive Software\Microsoft\Windows\CurrentVersion\Run\ Malware (Figure 60) will be set to that file. Even the existence of \Malware in CurrentVersion\Run is a host-based indicator for this malware.

To corroborate what we saw in the IDA Pro analysis in Lab03-06 Questions 1-4, I placed the PE file within BinText (Figure 62). We notice cc.exe as well as Malware and the Registry Key being passed. We also notice the format string with %c being referenced for a successful parsing of a command. We also notice the C:\Temp file with cc.exe being referenced. Once the malware is run, these will be good places to find host-based indicators.

A 000000007030	000000407030	0	Error 1.1: No Internet
A 000000007048	000000407048	0	Success: Internet Connection
A 000000007068	000000407068	0	Error 2.3: Fail to get command
A 000000007088	000000407088	0	Error 2.2: Fail to ReadFile
A 0000000070A8	0000004070A8	0	Error 2.1: Fail to OpenUrl
A 0000000070C4	0000004070C4	0	http://www.practicalmalwareanalysis.com/cc.htm
A 0000000070F4	0000004070F4	0	Internet Explorer 7.5/pma
A 000000007110	000000407110	0	Error 3.2: Not a valid command provided
A 00000000713C	00000040713C	0	Error 3.1: Could not set Registry value
A 000000007168	000000407168	0	Malware
A 000000007170	000000407170	0	Software\Microsoft\Windows\CurrentVersion\Run
A 0000000071A0	0000004071A0	0	C:\Temp\cc.exe
A 0000000071B0	0000004071B0	0	C:\Temp
A 0000000071B8	0000004071B8	0	Success: Parsed command is %c
U 000000006150	000000406150	0	(null)

Figure 62: Strings found in IDA Pro found in BinText.

Upon executing this malware within a Windows 10 sandbox, a command prompt window briefly popped up and was too quick for it to be captured via screenshot. However, I did capture traffic sent to www.practicalmalwareanalysis.com using the User-Agent string identified in Figure 22 (Figure 63).

```
10/23 05:09:29 PM [        DNS Server] Received A request for domain 'www.practicalmalware
ysis.com'.
10/23 05:09:29 PM [          Diverter] Lab06-03.exe (1696) requested TCP 192.0.2.123:80
10/23 05:09:29 PM [    HTTPListener80]   GET /cc.htm HTTP/1.1
10/23 05:09:29 PM [    HTTPListener80]   User-Agent: Internet Explorer 7.5/pma
10/23 05:09:29 PM [    HTTPListener80]   Host: www.practicalmalwareanalysis.com
10/23 05:09:29 PM [    HTTPListener80]
10/23 05:09:59 PM [          Diverter] msedge.exe (2932) requested UDP 239.255.255.250:190
```

Figure 63: Requests in Fakenet corroborate IDA Analysis.

Because the malware has functionality to quit if there is not an internet connection, the C:\Temp folder was not found and the registry key was not located either. There was also traffic caught by Lab06-03.exe within process monitor (Figure 64).

Time ...	Process Name	PID	Operation	Path	Result	Detail
5:09:2...	svchost.exe	2168	UDP Send	DESKTOP-P05952A:57816 -> DESKTOP-P05952A:domain	SUCCESS	Length: 50, seqnu...
5:09:2...	fakenet.exe	6716	UDP Receive	DESKTOP-P05952A:domain -> DESKTOP-P05952A:57816	SUCCESS	Length: 50, seqnu...
5:09:2...	fakenet.exe	6716	UDP Send	DESKTOP-P05952A:domain -> DESKTOP-P05952A:57816	SUCCESS	Length: 66, seqnu...
5:09:2...	svchost.exe	2168	UDP Receive	DESKTOP-P05952A:57816 -> DESKTOP-P05952A:domain	SUCCESS	Length: 66, seqnu...
5:09:2...	Lab06-03.exe	1696	TCP Connect	DESKTOP-P05952A:1163 -> 192.0.2.123:http	SUCCESS	Length: 0, mss: 14...
5:09:2...	fakenet.exe	6716	TCP Accept	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 0, mss: 14...
5:09:2...	fakenet.exe	6716	TCP Receive	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 103, seqn...
5:09:2...	Lab06-03.exe	1696	TCP Send	DESKTOP-P05952A:1163 -> 192.0.2.123:http	SUCCESS	Length: 103, starti...
5:09:2...	Lab06-03.exe	1696	TCP Receive	DESKTOP-P05952A:1163 -> 192.0.2.123:http	SUCCESS	Length: 17, seqnu...
5:09:2...	Lab06-03.exe	1696	TCP Receive	DESKTOP-P05952A:1163 -> 192.0.2.123:http	SUCCESS	Length: 1460, seq...
5:09:2...	Lab06-03.exe	1696	TCP Receive	DESKTOP-P05952A:1163 -> 192.0.2.123:http	SUCCESS	Length: 93, seqnu...
5:09:2...	fakenet.exe	6716	TCP Send	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 17, startim...
5:09:2...	fakenet.exe	6716	TCP Send	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 21, startim...
5:09:2...	fakenet.exe	6716	TCP Send	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 37, startim...
5:09:2...	fakenet.exe	6716	TCP Send	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 25, startim...
5:09:2...	fakenet.exe	6716	TCP Send	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 22, startim...
5:09:2...	fakenet.exe	6716	TCP Send	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 2, startime:...
5:09:2...	fakenet.exe	6716	TCP Send	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 1446, starti...
5:09:2...	fakenet.exe	6716	TCP Disconnect	DESKTOP-P05952A:http -> DESKTOP-P05952A:1163	SUCCESS	Length: 0, seqnum:...
5:09:2...	Lab06-03.exe	1696	TCP Disconnect	DESKTOP-P05952A:1163 -> 192.0.2.123:http	SUCCESS	Length: 0, seqnum:...

Figure 64: Procmon network traffic.

When the malware was run in a Windows XP environment, the command prompt box showed for an extended period of time and I was able to capture a screenshot showing the string, “Success: Internet Connection” (Figure 65) corroborating the string found in Figure 15.

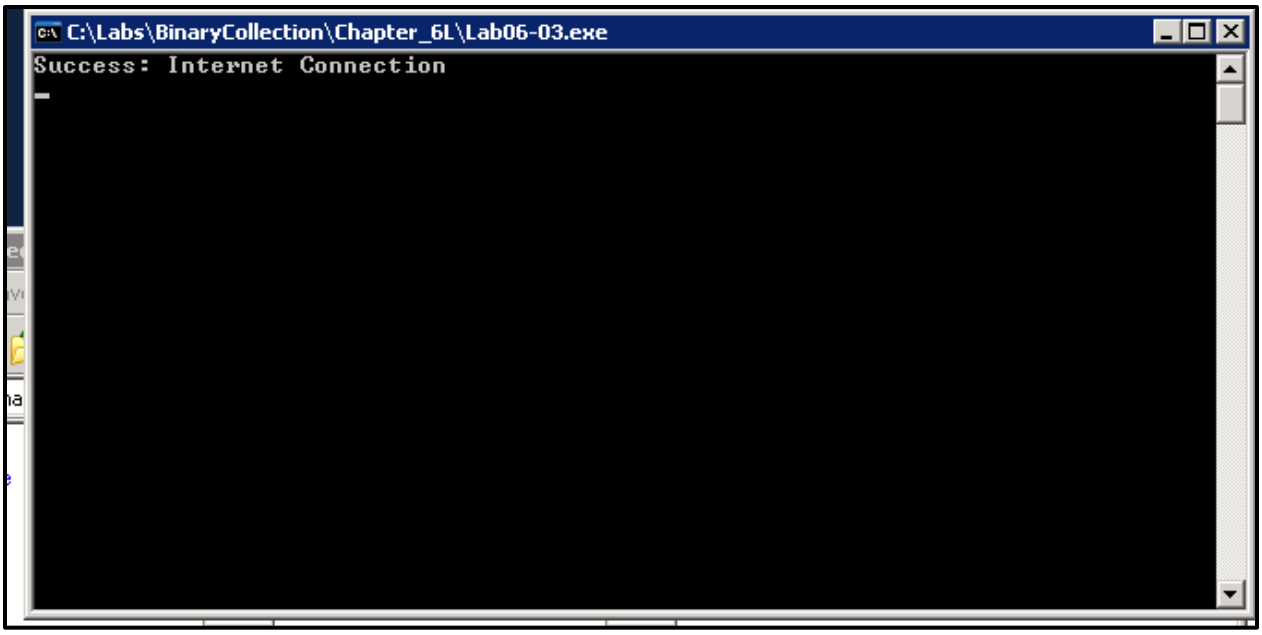

Figure 65: Popup window after running Lab06-03.exe.

However, no files or registry changes identified within the IDA Pro analysis could be found. A hypothesis as to why the anticipated behavior did not happen is that the domain the malware calls out to is no longer existent due to the timestamp showing the malware as being over 12 years old. This was confirmed when navigating to http://www.practicalmalwareanalysis.com/ (Figure 66). The domain navigates to a standard page reserved by GoDaddy.com, a web domain broker company and shows that the page has been purchased but has not yet been made into a website yet.

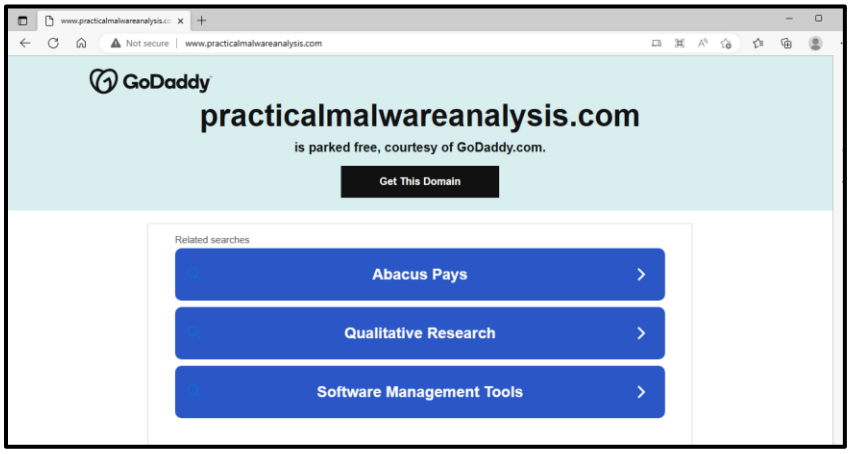

Figure 66: Domain of http://www.practicalmalwareanalysis.com/.

To ensure that this is not a mistake and that the purchased domain, although not built, is valid, we can refer to the book to see if there are any references as to accessing the domain. Evidence that the domain should be accessible is located on page xxix and states that the malware samples can be downloaded from it (Figure 67).

packing techniques.

This book will teach you how and when to use various malware analysis techniques. Understanding when to use a particular technique can be as important as knowing the technique, because using the wrong technique in the wrong situation can be a frustrating waste of time. We don't cover every tool, because tools change all the time and it's the core skills that are important. Also, we use realistic malware samples throughout the book (which you can download from *http://www.practicalmalwareanalysis.com/* or *http://www.nostarch.com/malware.htm*) to expose you to the types of things that you'll see when analyzing real-world malware.

Figure 67: The domain should be accessible according to the book.

To find additional evidence that the domain is dead and does not contain commands that the malware intends to execute, we navigate to the domain in Figure 23 that is called by the function InternetOpenUrl. The same GoDaddy page appears. We press F12 to bring up the HTML code and see that there are no comments at the top of the page where we expect the code the malware intents to execute to be (Figure 68).

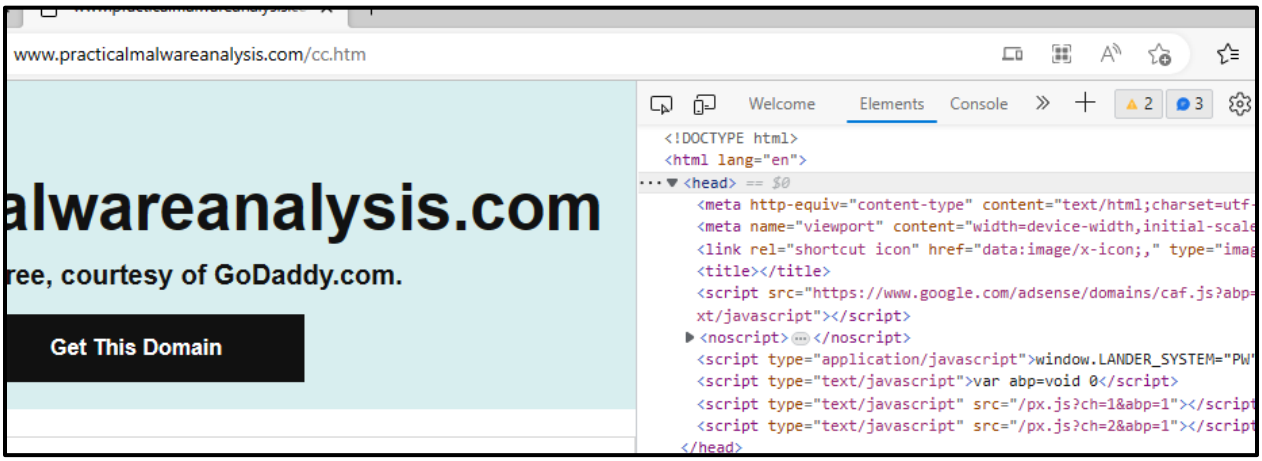

Figure 68: No HTML comments for the malware to execute.

Based on this, we can determine that this malware is dead and cannot execute. There is no valid domain to call out to in order for it to retrieve the extra code. The age of the malware and its attempts to use obsolete software suggest that if this was found in the wild, the creator of the malware has moved on to modify it with more up-to-date code and potentially uses a different domain name where they hid the extra code.

However, if this malware was discovered around the time of its creation where the domain is active, then we could expect to see the host-based indicators of C:\Temp\cc.exe existing as well as the existence of CurrentVersion\Run\Malware registry key.

LAB 6-3 Question 6

What is the purpose of this malware?

As detailed extensively above, this malware first checks for an internet connection and quits if the infected machine does not have one. It then intends to use Internet Explorer 7.5/pma to open and read the HTML code from the domain http://www.practicalmalwareanalysis.com//cc.htm. This page contains an HTML comment at the top of the page identified by the string <!--. If the string is not found, the malware quits. If the string is found, then it will execute a series of commands to download from the domain a file titled cc.exe and place it into the infected machine's C:\Temp folder. If the folder does not exist, then the malware creates it.

The malware then will modify the registry key in the HKLM hive of Software\Microsoft\Windows\CurrentVersion\Run\. It will create the key of \Malware, making the full path Software\Microsoft\Windows\CurrentVersion\Run\Malware. The \Malware value will be set to C:\Temp\cc.exe in order to establish persistence within the infected machine because the \Run registry is used to establish programs that run upon starting up the machine.

The malware provides its owner a series of command options within a Switch Jump Table structure to create files, delete files, and create directories. It also allows the malware owner to copy files from their machine and place them on the infected machine. There aren't any indications that the owner of the malware can do this from a shell as no function calls to a shell found within the IDA Pro analysis (Figure 69). This suggests that in order for the malware owner to perform the commands built in to the malware, they simply modify the HTML comment in the domain it calls out to. Although the malware specifically requests to copy a file named cc.exe

from the domain, any PE file can be renamed to cc.exe to perform any number of tasks to further cause damage to the infected machine.

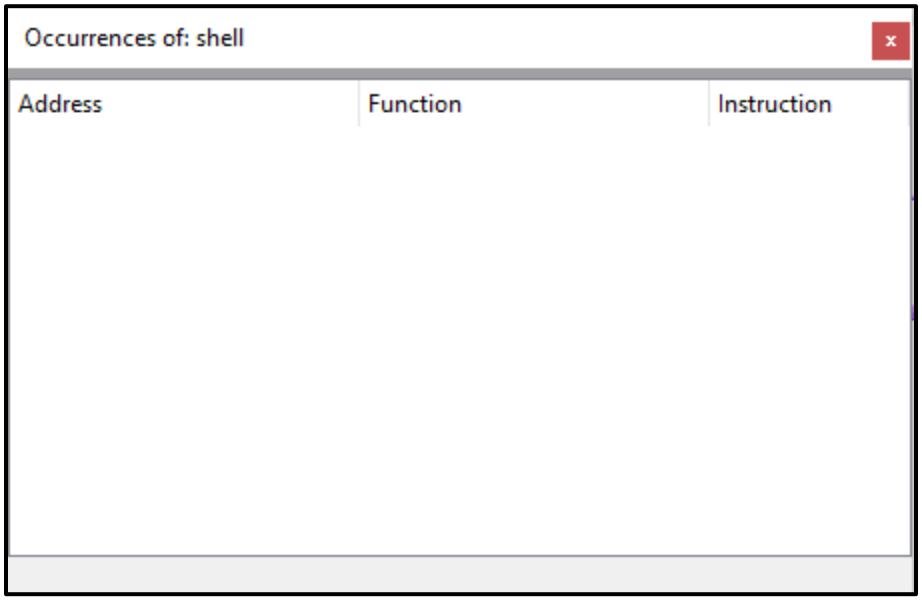

Figure 69: No references to "Shell" within Lab06-03.exe disassembly.