# Labs

**Lab 15-1**

Analyze the sample found in the file *Lab15-01.exe*. This is a command-line program that takes an argument and prints “Good Job!” if the argument matches a secret code.

**Questions**

|  |  |
| --- | --- |
|  | 1. What anti-disassembly technique is used in this binary? |
|  | 2. What rogue opcode is the disassembly tricked into disassembling? |
|  | 3. How many times is this technique used? |
|  | 4. What command-line argument will cause the program to print “Good Job!”? |

**Brief analysis and Answers：**

I choose to use “Ghidra”to disassemble the program. After installing, it will automatically pop up the analysis tool, which shows 5 red points for bugs. Therefore, we could tell the broke code is 5 times. Each one, we could find something that look likes the following:

**0040100c 33 c0 XOR EAX,EAX**

**0040100e 74 01 JZ LAB\_00401010+1**

**LAB\_00401010+1 XREF[0,1]: 0040100e(j)**

**00401010 e8 8b 45 CALL SUB\_8b4c55a0**

**0c 8b**

**00401015 48 DEC EAX**

**00401016 04 0f ADD AL,0xf**

**00401018 be 11 83 MOV ESI,0x70fa8311**

**fa 70**

**0040101d 75 3f JNZ LAB\_0040105e**

**0040101f 33 c0 XOR EAX,EAX**

**00401021 74 01 JZ LAB\_00401024**

**00401023 e8 ?? E8h**

**LAB\_00401024 XREF[1]: 00401021(j)**

**00401024 8b 45 0c MOV EAX,dword ptr [EBP + 0xc]**

**00401027 8b 48 04 MOV ECX,dword ptr [EAX + 0x4]**

**0040102a 0f be 51 02 MOVSX EDX,byte ptr [ECX + 0x2]**

**0040102e 83 fa 71 CMP EDX,0x71**

**00401031 75 2b JNZ LAB\_0040105e**

**00401033 33 c0 XOR EAX,EAX**

**00401035 74 01 JZ LAB\_00401037+1**

**LAB\_00401037+1 XREF[0,1]: 00401035(j)**

**00401037 e8 8b 45 CALL SUB\_8b4c55c7**

**0c 8b**

**0040103c 48 DEC EAX**

**0040103d 04 0f ADD AL,0xf**

**0040103f be 51 01 MOV ESI,0xfa830151**

**83 fa**

**00401044 64 75 17 JNZ LAB\_0040105e**

**00401047 33 c0 XOR EAX,EAX**

**00401049 74 01 JZ LAB\_0040104b+1**

**LAB\_0040104b+1 XREF[0,1]: 00401049(j)**

**0040104b e8 68 10 CALL SUB\_407020b8**

**30 40**

We could tell from the above program that it has 5 times repeat for the false

conditional branches: xor eax,eax and followed by jz. This would cause an error because the disassembler will disassembling the opcode oxE8 while the jz instruction will run the ox8b or whatever after the oxE8. For disassembling code, the default is jz false and it will not jump, in the code it actually jumps for jz is true, thus the error happened.

By clearing the code, we have the following CMP, which means will jump if equal the number; Thus, 0x70,0x64,0x71 are what we need, which are “pdq”as their ASCII code.Note, MOVSX is the reading address for the register, therefore, 0x70,0x64,0x71 is the right sequence for the inputs

LAB\_00401011 XREF[1]: 0040100e(j)

00401011 8b 45 0c MOV EAX,dword ptr [EBP + 0xc]

00401014 8b 48 04 MOV ECX,dword ptr [EAX + 0x4]

00401017 0f be 11 MOVSX EDX,byte ptr [ECX]

0040101a 83 fa 70 CMP EDX,0x70

0040101d 75 3f JNZ LAB\_0040105e

0040101f 33 c0 XOR EAX,EAX

00401021 74 01 JZ LAB\_00401024

00401023 e8 ?? E8h

LAB\_00401024 XREF[1]: 00401021(j)

00401024 8b 45 0c MOV EAX,dword ptr [EBP + 0xc]

00401027 8b 48 04 MOV ECX,dword ptr [EAX + 0x4]

0040102a 0f be 51 02 MOVSX EDX,byte ptr [ECX + 0x2]

0040102e 83 fa 71 CMP EDX,0x71

00401031 75 2b JNZ LAB\_0040105e

00401033 33 c0 XOR EAX,EAX

00401035 74 01 JZ LAB\_00401038

00401037 e8 ?? E8h

LAB\_00401038 XREF[1]: 00401035(j)

00401038 8b 45 0c MOV EAX,dword ptr [EBP + 0xc]

0040103b 8b 48 04 MOV ECX,dword ptr [EAX + 0x4]

0040103e 0f be 51 01 MOVSX EDX,byte ptr [ECX + 0x1]

00401042 83 fa 64 CMP EDX,0x64

00401045 75 17 JNZ LAB\_0040105e

00401047 33 c0 XOR EAX,EAX

00401049 74 01 JZ LAB\_0040104b+1

**Q1.** This program uses false conditional branches: an xor eax, eax, followed by jz. For disassembling code, the default is jz false and it will not jump, in the code it actually jumps for jz is true, thus the error happened.

**Q2.** The default is take jz into false which will not jump, thus it will trick the disassembler into disassembling the opcode 0xE8, the first of a 5-byte call instruction, which immediately follows the jz instruction.

**Q3.** We could tell from the above program that it has 5 times repeat for the false conditional branches: xor eax,eax and followed by jz. Therefore, the false conditional branch technique is used five times in this program.

**Q4.** “pdq” will cause the program to print “Good Job!”.By clearing the code, we have the three CMP lines, which means jump if two numbers are equal; Thus, 0x70,0x64,0x71 are what we need, which are “pdq”as their ASCII code.Note, MOVSX is the reading address for the register, therefore, 0x70,0x64,0x71 is the right sequence for the inputs and “pdq”is the right command. The following is the result:

C:\Users\User\Desktop\Practical Malware Analysis Labs\BinaryCollection\Chapter\_15L>Lab15-01.exe "pdq"

Good Job!

## Lab 16-1

Analyze the malware found in *Lab16-01.exe* using a debugger. This is the same malware as *Lab09-01.exe*, with added anti-debugging techniques.

### Questions

|  |  |
| --- | --- |
|  | 1. Which anti-debugging techniques does this malware employ? |
|  | 2. What happens when each anti-debugging technique succeeds? |
|  | 3. How can you get around these anti-debugging techniques? |
|  | 4. How do you manually change the structures checked during runtime? |
|  | 5. Which OllyDbg plug-in will protect you from the anti-debugging techniques used by this malware? |

**Brief analysis and Answers：**

Tools: <https://github.com/romanzaikin/OllyDbg-v1.10-With-Best-Plugins-And-Immunity-Debugger-theme->

Using both “Ghidra”(look for code) and OllyDbg(change debug flag at run time) for this part.

Checking Ghidra code, we find three flags at address, FS:[0x30] from 0x2,0x18,0x68 etc.From the book, page 354-355, we can know the corresponding name of the flags are BeingDebugged, ProcessHeap, and NTGlobalFlag. Let’s take a look at the call function after each JZ instruction, since no line is out, this function will terminate this program probably.

00401117 64 a1 30 MOV EAX,FS:[0x30]

00 00 00

0040111d 8a 58 02 MOV BL,byte ptr [EAX + 0x2]

00401120 88 5d f4 MOV byte ptr [EBP + local\_10],BL

00401123 0f be 45 f4 MOVSX EAX,byte ptr [EBP + local\_10]

00401127 85 c0 TEST EAX,EAX

00401129 74 05 JZ LAB\_00401130

0040112b e8 d0 fe CALL FUN\_00401000 undefined FUN\_00401000(void)

ff ff

LAB\_00401130 XREF[1]: 00401129(j)

00401130 64 a1 30 MOV EAX,FS:[0x30]

00 00 00

00401136 8b 40 18 MOV EAX,dword ptr [EAX + 0x18]

00401139 3e 8b 40 10 MOV EAX,dword ptr DS:[EAX + 0x10]

0040113d 89 45 f0 MOV dword ptr [EBP + local\_14],EAX

00401140 83 7d f0 00 CMP dword ptr [EBP + local\_14],0x0

00401144 74 05 JZ LAB\_0040114b

00401146 e8 b5 fe CALL FUN\_00401000 undefined FUN\_00401000(void)

ff ff

LAB\_0040114b XREF[1]: 00401144(j)

0040114b 64 a1 30 MOV EAX,FS:[0x30]

00 00 00

00401151 3e 8b 40 68 MOV EAX,dword ptr DS:[EAX + 0x68]

00401155 83 e8 70 SUB EAX,0x70

00401158 89 45 ec MOV dword ptr [EBP + local\_18],EAX

0040115b 83 7d ec 00 CMP dword ptr [EBP + local\_18],0x0

0040115f 75 05 JNZ LAB\_00401166

00401161 e8 9a fe CALL FUN\_00401000 undefined FUN\_00401000(void)

ff ff

1. Checking Ghidra code, we find three flags at address, FS:[0x30] from 0x2,0x18,0x68 etc.From the book, page 354-355, we can know the corresponding name of the flags are BeingDebugged, ProcessHeap, and NTGlobalFlag.Therefore, the malware checks the status for BeingDebugged, ProcessHeap, and NTGlobalFlag.

2. The call function after each JZ instruction, since no line is out, this function will terminate this program probably. Therefore,if any of the malware’s anti-debugging techniques succeed, it will terminate and remove itself from disk.

3. One way is manually change the jump flags in OllyDbg during runtime; Another way is to modify the structures the malware checks in memory either manually or by using an OllyDbg plug-in like PhantOm or the Immunity Debugger (ImmDbg) PyCommand hidedebug.

4. In the book pages from 657 to 659, we could use a step-by-step way to dump and modify the structures in OllyDbg. In its command line, type:

dump fs:[30] + 2

This command helps BeingDebugged flag into the dump window.

Then, we have to right click the flag and Binary -> Filled with 00’s

This command will set the flag to be 0; With this change,this flag will perform several times and the start function will not be called by malware now.

Finally,we also need to use dump to clear the command after that

The same step will be used for other flags.

5. We can use the OllyDbg plug-in PhantOm and ImmDbg PyCommand hidedebug, which will stop this malware’s checks.

## Lab 18-1

Your goal for the labs in this chapter is simply to unpack the code for further analysis. For each lab, you should try to unpack the code so that other static analysis techniques can be used. While you may be able to find an automated unpacker that will work with some of these labs, automated unpackers won’t help you learn the skills you need when you encounter custom packers. Also, once you master unpacking, you may be able to manually unpack a file in less time than it takes to find, download, and use an automated unpacker.

Each lab is a packed version of a lab from a previous chapter. Your task in each case is to unpack the lab and identify the chapter in which it appeared. The files are *Lab18-01.exe* through *Lab18-05.exe*.

**Answers：**

**“http://www.practicalmalwareanalysis.com/%s/%c.png”**