Xiaoyi Zhou

Purdue University CIT581

Malware Forensics

Lab 15: Anti-Disassembly

Due November 23, 2014

Instructor: Samuel Liles

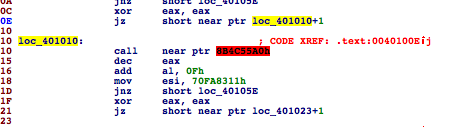
**Abstract**

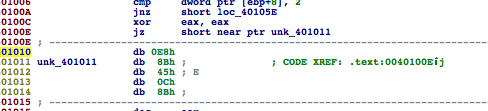
This lab is designed for understanding the anti-disassembly skill. I think the three labs are not difficult but sometimes there are tricky. They mainly use false conditional jump, pointer abuse and missed cross-reference as anti-disassembly skill. If we can figure out the trick part and fix it, the malware 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**

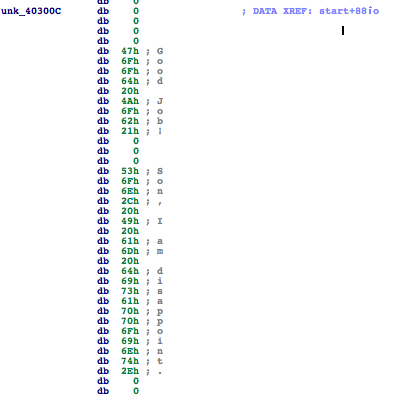
I think this malware contains less codes than previous labs. Therefore we can go through the main function easily. See the code reference is red and data the after call function is red. The malware might use anti-disassembly skill in red area. The call function has problem because the data after call 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.





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.

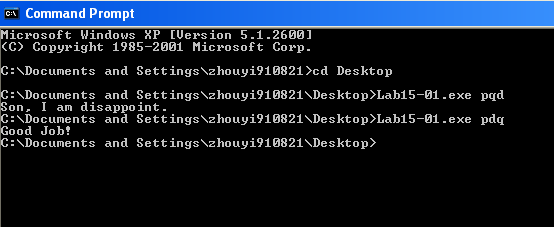
Now we can run this program by command line "Lab15-01.exe, XXX". I tried some random strings following Lab15-01.exe. The program will print, "Son, I am disappoint". I didn't know the it could print other string until I saw the following data.



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.

Mac:Users:xiaoyizhou:Desktop:Screen Shot 2014-11-23 at 7.47.02 PM.pngMac:Users:xiaoyizhou:Desktop:Screen Shot 2014-11-23 at 7.46.53 PM.png

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. 

**Issues or Problems**

Even thought I get 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 need 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 print 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.

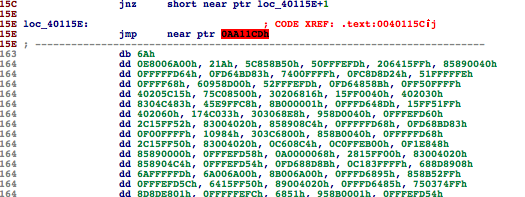
**4. What command-line argument will cause the program to print “Good Job!”**

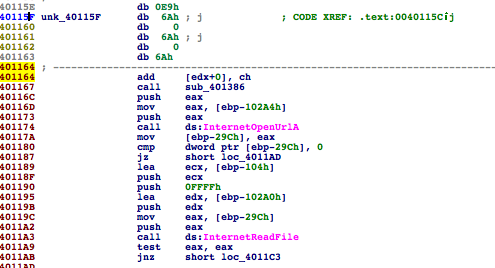
Lab15-01.exe pdq

**Lab15-02**

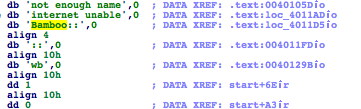
**Steps of Processes**

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.

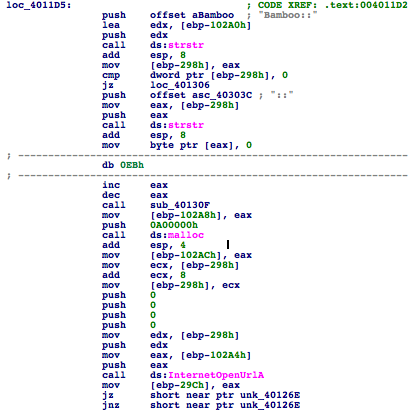




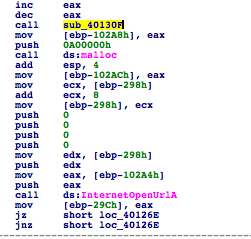
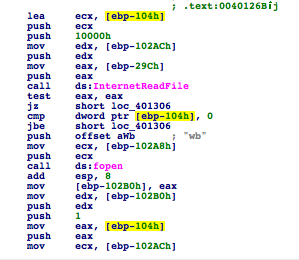
The first screenshot is before converting. The second one is after converting. We can see that JUM opcode is 0xE9 which is never executed. The malware intends to open an URL and read data from the specific URL initialized by sub\_401386. If we double click on sub\_401386, we will learn the purpose of 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 them 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.



This is the correct code after converting.



Anyway, just be patient when converting between data and code. The value in edx is the return value of InternetReafFile. The malware calls strstr to find the string 'Bamboo::' at the return data. The return value of strstr 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 strstr 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 censor 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.



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.

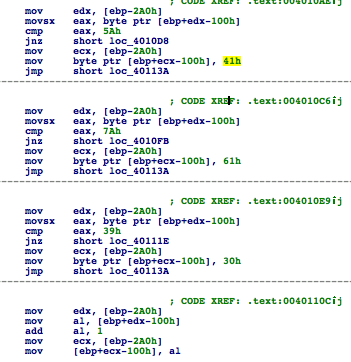
**Reviewed** **Questions**

**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.



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?**

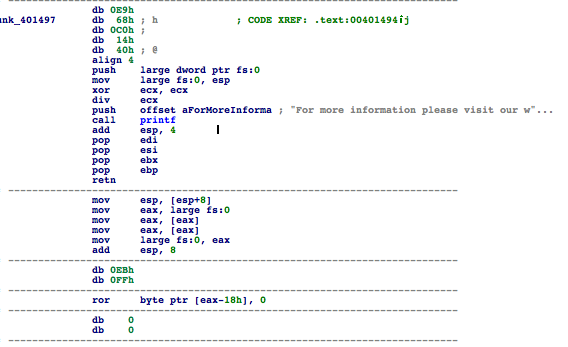
The information extracted from the page is a URL. The malware will open the URL and download the data to Account Summary.xls.exe. Then launch the modified Account Summary.xls.exe.

**Lab15-03**

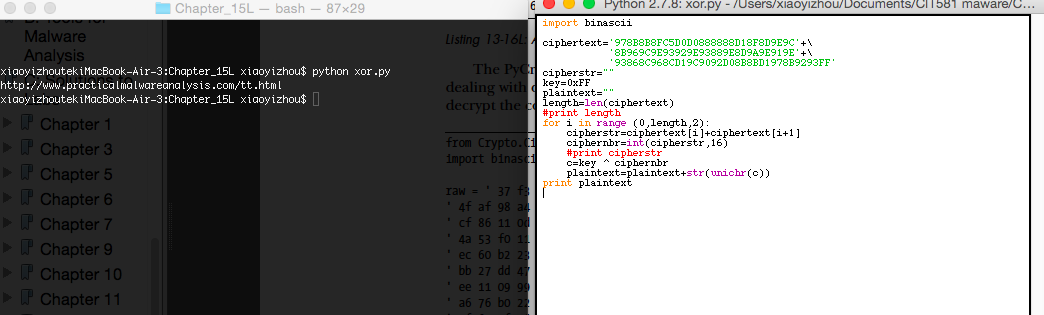
**Steps of Processes**

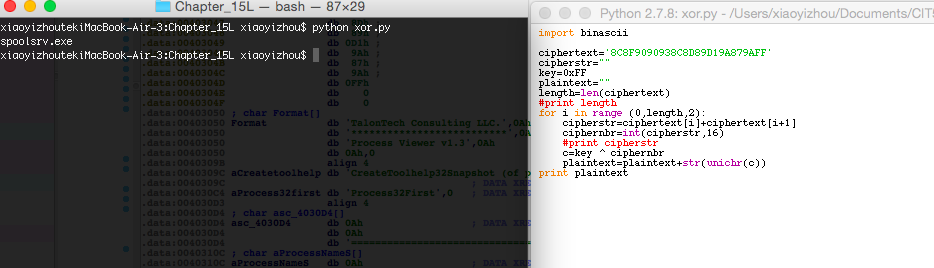
At the beginning of main function, we see that [ebp+4] is red, which means there is an anti-disassembly technique. I guess it might be pointer abuse introduced at our book. To verify my assumption, we can switch the window to stack of main and see what is [ebp+4]. Offset positive 4 means return. Therefore 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 0x004018C. We can directly navigate to 0x004018C 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.

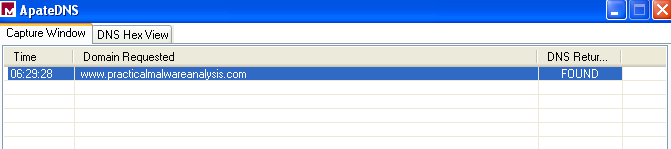


Now we directly navigate to 0x4014D7. There is an anti-disassembly trick. If we convert the code to data, we will see a few data but I didn't know what the data represented. However I don't think those data make matter. The most important function is located at next few lines. We see a function call 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.





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 spoolsrv.exe as parameter. The purpose is to run the application that contains spoolsrv.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.



**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 en 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