Xiaoyi Zhou

Purdue University CIT581

Malware Forensics

Lab 12: Cover Malware Launching

Due October 29, 2014

Instructor: Samuel Liles

**Abstract**

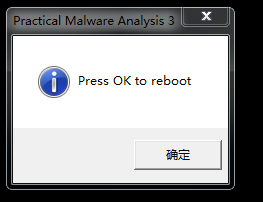
Lab12 is the most difficult lab to me so far. It is time consuming. Lab12-01 performs a DLL injection that infects the explorer.exe. Lab12-02 performs a process replacement that launches another program. But I didn't figure out which PE file is being injected into process. The file is encrypted and my decryption program can only decrypt partial strings. Lab12-03 is a keylogger, which is the easiest program at chapter12. Lab12-04 performs a privilege escalation and disables the Window File Protection in order to download more malwares. For lab12-02 and lab12-04, even though I tried to figure out the answers, I still feel like I don't understand those malwares.

**Lab 12-01**

**Steps of Process**

Based on previous experience, the DLL file is more likely to carry malicious code. So I open DLL file at IDA Pro first before checking the executable file. Navigating to DLLMain code area, there are two function calls sub\_10001030 and CreateThread. IDA Pro labeled sub\_10001030 as StartAddress. According to the features of CreateThread, StartAddress is one of the parameters, which indicates StartAddress is a pointer to a function stored at sub\_10001030. Then CreateThread could create an environment that helps to execute the function. I assume it is a malicious function. where three parameters are taken, lpThreadID, dwCreationFlags and lpParameter. Except the lpThreadID, rest of parameters has value 0. That is important in future analyze. Double click on sub\_10001030. The first function call is sprintf that takes ecx and a string as parameters. The purpose of sprintf is to format string. ecx is the value stored at var\_18. In this case, var\_18 is corresponding to CreationFlags. So it is 0. The formatted string "Practical Malware Analysis 0" is passed to next function call StartAddress. Technically StartAddress is not a function call. It is a pointer to a function. And I can notice there is another CreateThread after few lines of StartAddress. Therefore, the CreateThread here is to resolve the function stored at StartAddress. So double click on StartAddress. The function is not complicated here. It pops up a window box said that "Press OK to reboot". After the above operation is executed, the malware will sleep for 0xEA60 milliseconds which is one minute at decimal. Then ecx is increased by one. Remember that ecx is initialized as 0 at very beginning of sub\_10001030. Here ecx is increased and passed to sprintf again when the loop starts. The malicious function contains a loop from loc\_1000103D to 0x10001086. During this loop, the message box will keep popping and the number in "Practical Malware Analysis X" keeps increasing by one. The loop cannot be broken by condition.

That is pretty much the DLL file. Now I can assume that the executable file might attach the malicious DLL file to another process. If it pops up a window, then the infected processes could be explorer.exe at least. So I launch the malware and see what is the popping message box. Each time I click close or press ok, the counter at the string is increased by one. If I want to stop the message box, I could reboot the system or terminate the explorer process.

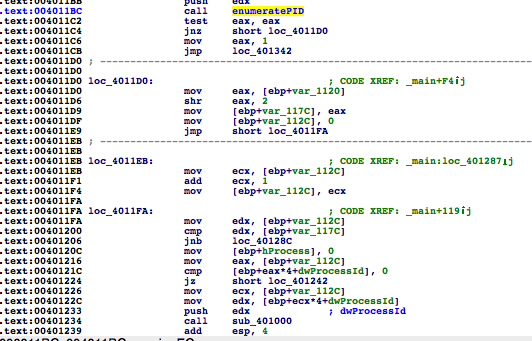


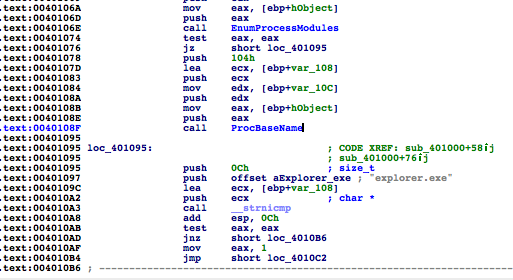
Load the executable file at IDA Pro. At parameter area, I notice a few parameters like lpStartAddress, lpBaseAddress, hModule, dwProcessID, hProcess, and Buffer. Those parameters are related to load library, get process address, process injection or process replacement. Therefore, the goal of analysis is to find which process is being injected by the malicious code. There are a lot of interceptions with psapi.dll. Based on the introduction at psapi.dll at MSDN, it is located at system directory and used to obtain the current status of a process. The program keeps calling LoadLibrary and GetProcessAddress to preform some functions to psapi.dll. And the addresses of the functions are stored into different pointer registers. The first function enumerates process modules of psapi.dll, and the function address is stored at dword\_408714. The second function gets the module base name. The third function enumerates process. After interception with psapi.dll, the program appends current directory path to Lab12-01.dll. Therefore Lab12-01.dll has to be placed at same place with Lab12-01.exe. The path of dll is stored at buffer. So rename buffer like "Lab12\_01\_DLL".



The next important function calls are located at dword\_408710 and Loc\_4011FA. dword\_408710 enumerates process and takes parameter dwProcessID. There isn't any specific instruction for enumerate process. But it takes process ID as parameter. So dword\_408710 is used to count all the process intercepting with psapi.dll and pass it to later function call. So I rename dword\_408710 like enumeratePID.

The function pass dwProcessID to sub\_401000 where OpenProcess is called with parameter PID. If OpenProcess is success, then it will return the specific process handle otherwise it is 0. So if the object with its PID didn't open successfully, the program will directly jump to loc\_401095. Otherwise the program will try to get some information related to the current opening process.



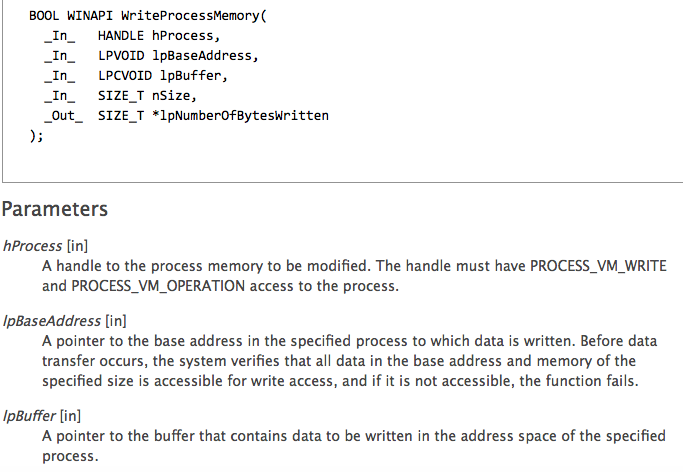


Here the name and module of the running process is retrieved and compared with explorer.exe. Clearly the program is trying to find explorer.exe at loc\_401095 and force it to launch. The return value to sub\_40100 will be process ID of explorer.exe. If PID to explorer.exe is obtained, then the program calls OpenProcess to open it otherwise the program starts a loop until the program finds explorer.exe. See the black line.



The last thing at main function will be implementing process injection at loc\_4012BE. According to our book, VirtualAllocEx will allocate and write the data used by the remote thread, and the WriteProcessMemory will allocate and write the remote thread code. The call to CreateRemoteThread will contain the location of the remote thread code (lpStartAddress ) and the data (lpParameter). LoadLibrary/GetProcAddress will need to be called to access functions that are not already loaded. WriteProcessMemory takes Lab12-02.dll and address of explorer.exe process as parameter, which indicates explorer.exe is infected by lab12-02.dll. It writes data to an area of memory in explorer.exe process (hprocess is A the handle to the process memory to be modified). Then it loads kernel32.dll and create a thread that runs in the virtual address space of explorer.exe.





The picture is retrieved from http://msdn.microsoft.com/en-us/library/windows/desktop/ms681674(v=vs.85).aspx.

**Issues or Problems**

When I first time ran the malware at virtual machine, I click ok and the malware stops running. But second time the malware was running, it popped message window. Another issue I met is at IDA Pro function call sub\_401000. After function call OpenProcess, the program compares process handle with value 0, if process handle is 0, then program directly jumps to location 401095. However, if process handle is 0, then OpenProcess fail. Even if the program jumps to location 401095, strnicmp will fail as well. Therefore, I think sub\_401000 could return fail if OpenProcess return fail.

**Conclusion**

This malware performs a process injection that loads the malicious code into explorer.exe. Once the malware is running, it keeps popping up a message window displaying malware practical analysis X. X indicates the minutes the malware is running. When the minute increases, X also increases.

**Reviewed Questions**

**1. What happens when you run the malware executable?**

A message window with "Malware Practical Analysis X" keeps popping up every minute. If user click OK or close, it will pop again with increase of X.

**2. What process is being injected?**

explorer.exe

**3. How can you make the malware stop the pop-ups?**

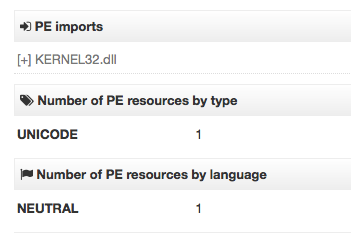
Reboot the system or terminate explorer.exe.

**4. How does this malware operate?**

This malware performs a process injection. It attaches lab12-01.dll to explorer.exe. Once the malware is running, it keeps popping up a message window displaying malware practical analysis X. X is a counter that indicates the minutes the malware is running.

**Lab12-02**

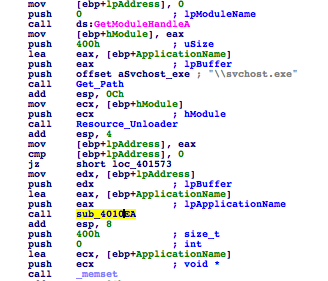
**Steps of Process**



The malware hides itself at resource section with type UNICODE. Therefore, we should use resource hacker to extract the malicious file. Moreover, I noticed ResumeThread and CreateProcess, I assume this malware preforms a process replacement that it inject itself to a common executable program. The common program got suspended and the malicious code is running. After that, ResumeThread could resume the program. Here I didn't see SuspendThread. So the program might be suspended by passing CREATE\_SUSPENDED (0x4) as the dwCreationFlags parameter when performing the call to CreateProcess, which is mentioned at our book at part of process replacement.

Main function at this malware is not complicated. The first important function call is sub\_40149D taking svchost.exe and eax as parameter. Within sub\_40149D, the program retrieves the system directory which is also the directory of svchost.exe. The system directory is appended to svchost.exe and return to eax. So I rename sub\_40149D as Get\_Path. The full path is passed to next function call sub\_40132C.

The purpose of sub\_40132C is to find the resource, assign memory space for the resource file, and then free the resource. Since I have already known that the resource file is encrypted, therefore the decryption process might be located at sub\_40132C. So I navigate to function VirtualAlloc and FreeResource. Between the two functions, sub\_401000 is called (before FreeResource function). Double click on it and I notice a loop starting at loc\_40100D. The program put the value at the resource file into [ebp+arg\_8] and them performs XOR encryption with 0x41. Therefore, we can decrypt the encrypted content at resource file. Rename sub\_40132C as Resource\_Unloader.



Resource\_Unloader will return the pointer to PE file and store it at lpAddress and lpBuffer. We can rename lpBuffer as Mal\_Buffer for future analysis.

The next function call sub\_4010EA is very important because the malware are supposed to inject itself to svchost.exe after being unloaded. Double click on sub\_4010EA. The program checks the validity of MZ(0x5A4D) and PE(0x4550). If the values are both valid, MZ and PE are moved to var\_4 and var\_8, respectively. Actually PE is moved to var\_8 before checking validity. If checking validity fail, the program terminates. Then I rename var\_4 and var\_8 as MZ and PE, respectively. MZ is the first 2 bytes of every MS-DOS executable. PE is the signature of the Windows program header that follows.

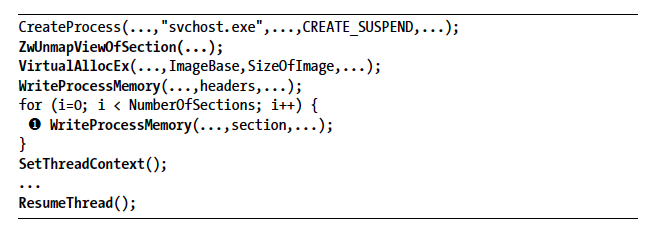
Parameter ProcessInformation has been used many times. Later ProcessInformation is combined with thread. If the malware intends to perform process replacement, then the thread of current target process should be suspended. The target-injected process is svchost.exe. So ProcessInformation means information of svchost.exe. ProcessInformation.hThread means thread at process of svchost.exe. The chapter 12 has already provided hint pseudo code. I analyze the assembly code based on C pseudo code. The purpose of function calls at sub\_4010EA is as following:

CreateProcessA: dwCreationFlags is set to 4. The process of svchost.exe is created but also suspended.

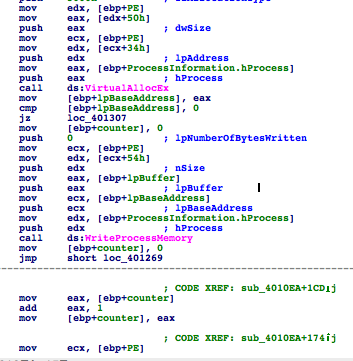
GetThreadContext: get the thread from svchost.exe in order to resume it.

ReadProcessMemory: the malicious program is reading and writing directly to process memory spaces. It takes lpBaseAddress as reading address and 4 as reading bytes.

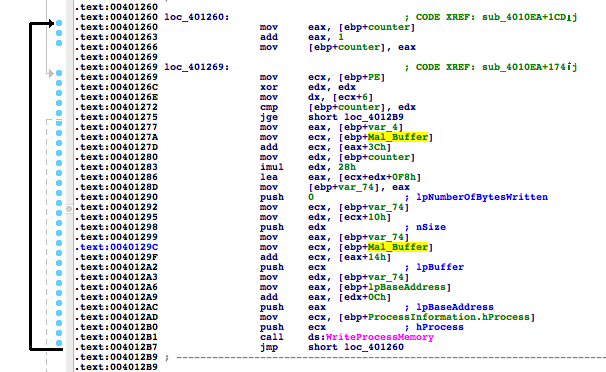
UnmapViewOfSection: this function is combined with GetProcessAddress in order to make change to svchost.exe process.



VirtualAllocEx(lpAddress, dwSize, flAllocationType, flProtect) is a little bit complicated at 0x401222. The function intercepts with svchost.exe. lpAddress is [PE+0x34]. dwSize is [PE+0x50]. The hex digits are offset of PE file. So I check PE file offset table from http://www.reteam.org/ID-RIP/database/essays/fboyjoe/exe\_hdr.html. [PE+0x34] is ImageBase; [PE+0x50] is SizeOfImage; just like the screenshot from our book. Therefore the starting allocating address is the ImageBase Address of PE file. The allocating size of region is SizeOfImage at PE file. After VirtualAllocEx, the program should start write process into svchost.exe process. WriteProcessMemory(lpBuffer, nSize, lpBaseAddress, hProcess) will modify svchost.exe by writing the PE pointer to the allocating address. The number of bytes to be written is [PE+0x54] which is SizeOfHeader of PE file. The loop starts right after the first WriteProcessMemory. var\_70 is initialized as 0 as counter. So I rename it as counter.

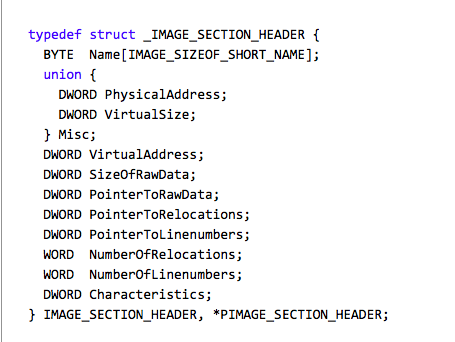


The loop condition is sat at 0x401272. Counter is compared with edx that represents [PE+0x6]. Check the PE offset page, [PE+0x6] means NumberOfSections of the PE file. So the program starts writing data from the beginning until the last section of the PE file.



var\_74 has been used many times. The value stored at var\_74 is eax where var\_4 is stored. So var\_74=var\_4=MZ section header. The section header is moved to eax and added by 0x3C. To check the offset value of MZ, we still go to the same website but focus on the part of SectionHeader. [MZ+0x3C] is FileAligment that could be regarded as the adjoint address of section header and PE header. After the pointer is initialized, counter is multiplied by 0x28 (40 bytes in decimal). Therefore I assume each section is 40bytes. Then the program should load effective address of [PE header+40bytes]. But the program loads [PE header+0xF8] here. Therefore I assume that 0xF8 is not the section size but starting address.

The section header should be a structure. So I check the structure of section header at MSDN.



Except Name[] is 8bytes, NumberOfRelocations is 2bytes, NumberOfLineNumbers is 2 bytes, the rest of parameters is 4bytes. PhysicalAddress and VirtualSize should be regarded as a union.So the union is 4bytes instead of 8bytes. Therefore the size of header section should be 8+4\*7+2=40.

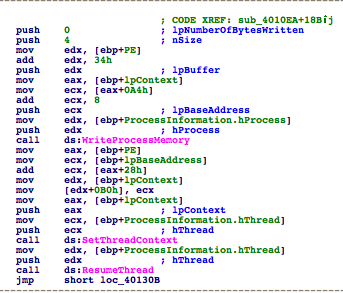
Now nSize is [MZ+0x10];lpBuffer is [MZ+0x14];lpBaseAddress is [MZ+0xC]; hprocess doesn't change as previous.

nSize=[MZ+0x10]=sizofrawdata

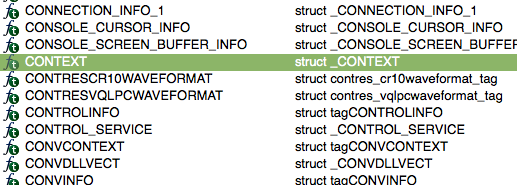
lpBuffer=[MZ+0x14]=pointerToRawData

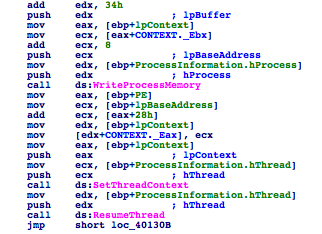
lpBaseAddress= [MZ+0xC]=virtual address

In this loop, the program calls WriteProcessMemory and writes the PE file section by section into svchost.exe to perform injection. When injection process is done, the suspended thread will be unfrozen.



Before the suspended thread is recovered, the program writes data from the PE ImageBase address into [0xA4+8] at svchost.exe. Then the program retrieves the suspended thread at SetThreadContext( ). The function SetThreadContext takes two parameters. The first parameter eax is important because it indicates the location of suspended thread. eax is set by [edx+0xB0] and [PE+0x28]. PE+0x28 means the AddressOfEntryPoint of PE file. [edx+0xB0] refers to the context of EAX. We can change the display name by adding a new CONTEXT structure.





Therefore, SetThreadContext retrieves the thread suspended at the entry point of PE file. The thread is resumed at ResumeThread( ). That is the final step of process replacement. However, the final step for analysis is to decrypt the resource file. Firstly we need to extract it. Remember that the cipher is XOR; the key is 0x41. I try to decrypt it but I failed to do it. See issues or problems.

**Issues or Problems**

I have a lot of issues during analysis process. But most of problems could be solved by Internet search. The most confused issue is the decryption process. I tried to create a program to decrypt it. However some of ASCII code cannot be intercepted with fopen, fgetc and fputc.

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int Decryption(int Cipher\_Nbr)

{

Cipher\_Nbr=Cipher\_Nbr ^ 0x41;

return Cipher\_Nbr;

}

int main()

{

FILE \* Cipherstring;

int i=0;

int counter=0;

int Ciphertext;

int DEC;

int CipherNbr;

int flag=1;

char P\_Nbr;

Cipherstring=fopen("ciphertext.txt","rt+");

if(Cipherstring==NULL)

{

printf("open file ciphertext.txt failed!\n");

exit(1);

}

i=5;

fseek(Cipherstring,i+counter,SEEK\_CUR);

while(flag<=2)

{

for(counter=0;counter<3;counter++)

{

Ciphertext=fgetc(Cipherstring);

fseek(Cipherstring,0,SEEK\_CUR);

printf("%d ",Ciphertext);

CipherNbr= Decryption(Ciphertext);

}

P\_Nbr=(char)(CipherNbr);

fputc(P\_Nbr,Cipherstring);

printf("%c \n",P\_Nbr);

}

//counter++;

fseek(Cipherstring,i+1,SEEK\_CUR);

flag++;

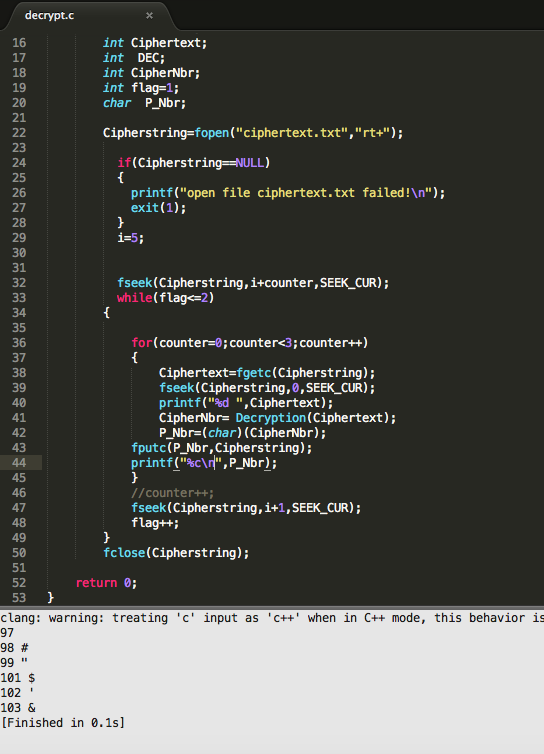
}

fclose(Cipherstring);

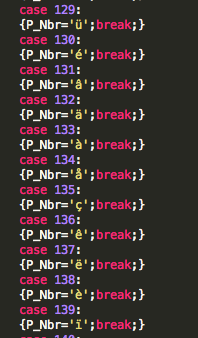
return 0;

}

The program could resolve decryption process only if the ASCII code could be displayed. For example if the file is ABCD abc(carriage return )EFGH efg. Just like the ciphertext from UNICODE->LOCOLIZATION->0, the first section is consist of 60 hex digits; the second section is consist of 16bytes characters. In this case, uppercase letter could be regarded as first section; lowercase letter could be regarded as second section; The decrypt result is like the following:



Some of ASCII code at the original ciphertext cannot be displayed. If I manage to decrypt the special ASCII code manually by switch statement, the program will show the error that character too large for enclosing character literal type.



**Conclusion**

This malware is difficult to me because I wasn't familiar with the structure and offset of PE file. But via analyzing this lab, I think I gain more knowledge about PE file and process replacement attack. This malware performs process replacement injection that launches another program between svchost.exe is suspended and resumed.

**Reviewed Questions**

**1. What is the purpose of this program?**

This malware preforms a process replacement to inject the malicious program into svchost.exe process without drawing attention.

**2. How does the launcher program hide execution?**

The program uses process replacement to hide execution. It suspends the thread and implement sthe another program. Then, it resumes the suspended thread to keep the process functional.

**3. Where is the malicious payload stored?**

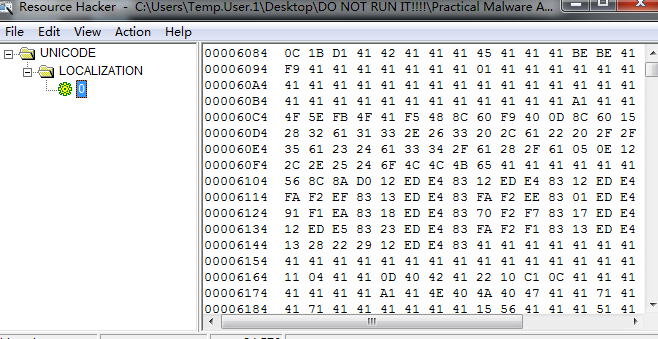
Open it at Resource Hacker. UNICODE->LOCALIZATION->0

**4. How is the malicious payload protected?**

The malicious program is stored at recourse section named LOCALIZATION. When the malware free the resource, it will decrypt it by XOR 0x41.

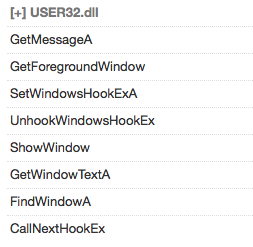
**5. How are strings protected?**

The string is encrypted by XOR 0x41

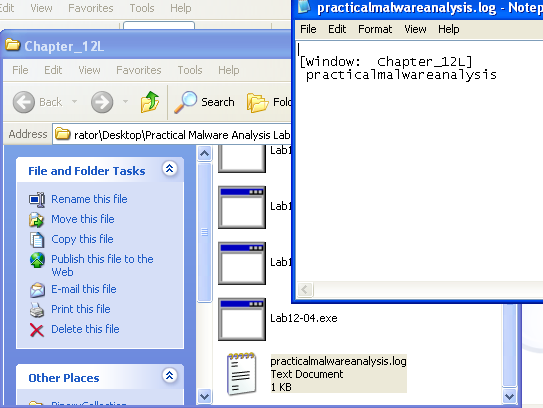


**Lab12-03**

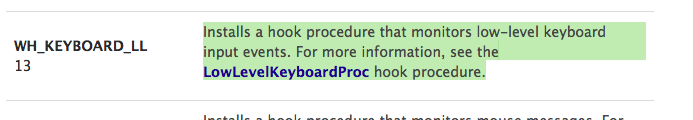
**Steps of Processes**



Analyzing the import functions at user32.dll, based on GetForegoundWindow, ShowWindow, GetWindowTextA and FindWindowA( ), I believe this is a keylogger. Based on SeWindowsHookEx, UnhookWindowsHookEx and CallNextHookEx, I think it is a keylogger that uses hook. If the malware is a keylogger, it will not cause too much damage or trouble, so we can run it right now. Few seconds later, I can see a file named practicalmalwareanalysis.log at same directory of the executable malware. Open the log file and the content is consist of the current window's name.



I put the executable file into IDA Pro. The main function is not complicated. The program calls FindWondow with parameter ConsoleWindowClass. It retrieves a handle to window with the class name ConsoleWindow; then the program sets the window's show state by calling ShowWindow at 0x40102F. After the state of window has been set up, the program will execute hook function. It calls SetWindowsHookEx with parameter hmod, lpfn, and idHook. hmod is the return value from previous GetModuleHandle(). It should be a handle to the DLL containing the hook procedure pointed to by the lpfn parameter. lpfn is a function pointer where the malicious code is stored. We can rename it as Hook\_Procedure. idHook indicates the type of hook to be installed. The value of idHook is set to 0x0D that installs a hook procedure that monitors low-level keyboard input events. The picture is retrieved from MSDN.

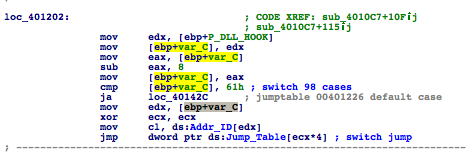


Now we double click on offset fn to analyze the hook procedure. The function takes three parameters, ncode, lParam, wParam. If we google the three parameter with hook function, we will learn that this function is LowLevelKeyboardProc Callback function. If ncode value is 0, the wParam and lParam parameters contain information about a keyboard message. The program can keep going until next comparison operation. Otherwise the program will use CallNextHook function. The next comparison is between wParam and 0x104. wParam is the identifier of the keyboard status. When the value is 0x104, it indicates that the user presses the F10 key or holds down the ALT key and then presses another key. It also occurs when no window currently has the keyboard focus. So if the user's behavior is just like the above, the program will jump to loc\_4010A1.

loc\_4010A1 contains a function call sub\_4010C7. This function takes lParam (a pointer to DLL\_HOOK structure) as parameter, which is used to loop up hook table in the future. Rename buffer as P\_DLL\_HOOK.



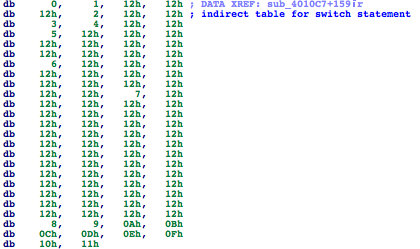
The program creates a file name practicalmalwareanalysis.log in order to store the key states. Then it calls SetFilePointer, GetForegroundWindow, GetWindowTextA, and WriteFile to record the name of the current window. Later, the program jumps to switch jump table. To analyze the jump table, we should firstly find the location of jump table. IDA Pro help us label the jump table as switch jump at loc\_401202. Here the jump table is stored at off\_401441[ecx\*4]. This jump table indicates where the program will jump. The jump location identifier is store at 40148D. The jump condition is stored at var\_c. So I rename the variables like following:



Let's say ebp+var\_c is offset;

offset=eax-8=X; The value of eax is determined by program. But we can check the value of Virtual Key Codes at http://msdn.microsoft.com/en-us/library/windows/desktop/dd375731(v=vs.85).aspx.

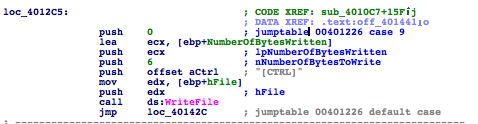
Then we look up what is the Xth number at Address location identifier table.



We denote the address location number with K. (Xth number in Addr\_ID table is K). Now double click on Jump\_Table and check what is the Kth string. That string will be the jump location to the program. To verify our analysis, I will take CTRL key as an example. The virtual-key code for CTRL is 0x11.

offset=0x11-0x8=0x9.

the 9th number at address location identifier is 4. The 4th string at jump table is loc\_4012C5. We just double click on loc\_4012C5 and see the codes at the location.



**Issues or Problems**

edx is passed to Address location table as a parameter at loc\_401202. However, the value in edx is calculated from buffer where lParam is stored. Therefore if I want to get the value of edx in order to find jump address, I should check the value of lParam. But in this case, the jump address is bond with wParam, which makes me concern that why the value of lParam could be substituted by virtual code of wParam.

**Conclusion**

**Reviewed Questions**

**1. What is the purpose of this malicious payload?**

It is a keylogger.

**2. How does the malicious payload inject itself?**

The malware injects itself by using hook function.

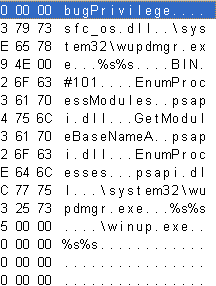
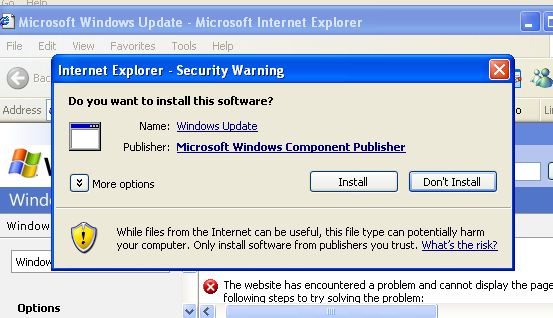
**3. What file system residue does this program create?**

The malware creates practicamalwareanalysis.log under the same directory of Lab12-03.exe

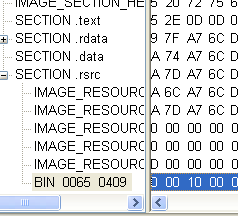
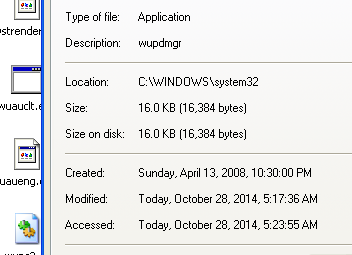
**Lab-04**

**Steps of Process**

Put the file into PEview firstly. I notice some useful strings as the picture below. The malware might try to intercept with wupdmgr.exe, winup.exe and get access to www.practicalmalwareanalysis.com. I launch the malware and an explorer page immediately popped up. The malware opens website of Microsoft Window Update and try to download a window updater executable file.

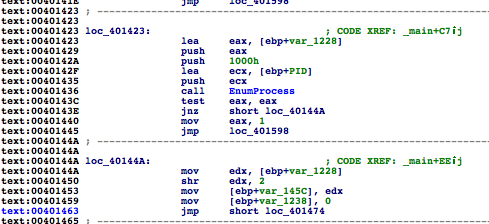
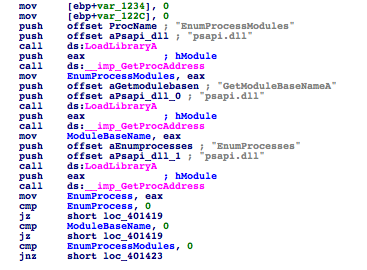


wupdmgr.exe is a process which belongs to the Windows Update procedure which controls updates for Microsoft Windows. I guess the purpose of the malware is to update the executable file. Navigate to the directory of wupdmgr.exe and check the properties. I found that the file has been modified when the malware is launched. It also contains a malicious file at resource section. We can either extract BIN at resource hacker or just let the malware drop it.



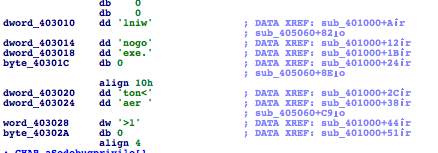
To see the detail of Lab12-04, we should put it into IDA Pro.

The main function is very long so that I divide the function into different parts. The first part of main function is from 0x401350 to loc\_401419. The first part contains some familiar function calls like LoadLibrary and GetProcAddress. For each pair of LoadLibrary/GetProcAddress, IDA Pro has labeled the module name and dll name. Therefore I rename each return value as more reasonable words. dword\_40312C is EnumProcessModules; dword\_403128 is ModuleBaseName; dword\_403124 is EnumProcess. The first two functions are used to help the third function EnumProcess to get the PID of all process because EnumProcess takes [ebp+PID] as parameter at loc\_401423. If none of function return value is 0, the program will jump to loc\_401423. I guess the malware intends to find its target injection file by searching the corresponding PID. I also see main function has a parameter named dwProcessID. I rename it as PID.



The second part of main function is from 0x401465 to 0x4014CF. The only function call at this code is is sub\_401000. The function has parameter labeled dwProcessID which is PID. PID is stored at an int arrary. So each time the function is called, the value of PID is push onto stack and passed to the function. If the function return value is 0, then the program starts a loop from 0x401465 to 0x4014CF until the return value is not 0. If the return value is not 0, then the program jumps to loc\_4014D1. Double click on sub\_401000 and analyze the details.

I divide sub\_401000 into two parts. The first part is from the beginning of the function to loc\_4010C2. The rest of codes is the second part. At the first part, the program uses many variables starting with dword.We need to figure out the value here because those values make up string1 and string2. To determine the value stored in dword\_Number, I double click on dword\_Number and convert them into character format.



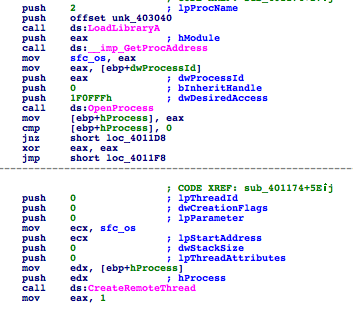
The order is reversed but I can still figure out that string2 is consist of dword\_403010, 403014, 403018, which is winlogon.exe; string1 is consist of 403020, 403024, and 403028, which is <not real>.Then the program calls OpenProcess, EnumProcessModules,and ModuleBaseName in order to get information from the processes at memory.ModuleBaseName takes string1 as parameter. Stirng1 here is considered as lpBaseName. It is a pointer to the buffer that receives the base name of the module. Therefore if the function call success, string1 value will be changed to the base name. Now we can analyze the second part of sub\_401000. It compares string1 and string2 in lowercase. If they are same, the function will returns PID to main function otherwise it returns 0. Therefore the purpose of sub\_401000 is to find which process is winlogon.exe and return its PID. After calling sub\_401000, PID is stored at eax and passed to var\_1234 at 0x4014C7.

The third part of main function is from loc\_4014E4 until the end. The program will jump to loc\_4014E4 only if sub\_401000 returns the PID.

The first function call at third part is sub\_401174 which takes PID of Winlogon.exe as parameter. One of purposes of sub\_401174 is clearly labeled as SeDebugPrivilege. The program gets the current process and its token; adjusts the token privilege from enable to disable. SeDebugPrivilege function will return ERROR\_SUCCESS if the function adjusted all specified privileges. The other purpose of sub\_401174 is to intercept with a DLL file. To determine which dll file is being loaded, we double click on the offset before LoadLibrary. The string is displayed as sfc\_ll at IDA Pro demo version. By Internet search, I know that sfc\_ll indicates sfc\_os.dll that is a executable portion of Windows File Protection. The information related to sfc\_0s.dll is retrieved from https://bitsum.com/aboutwfp.asp.

LoadLibrary has another parameter "2". Normally it only takes one pointer parameter. So we can ignore the meaning of 2 for now. The program loads sfc\_os.dll library to open the process with PID of winlogon.exe. The return value should be the handle stored at hProcess. And the pointer to sfc.dll is stored at lpStartAddress. dwDesireAccess for OpenProcess( ) is set to 0x1F0FF. I didn't find the specific meaning for 0x1F0FF. But I found the general meaning for 0x1F0FF is "for all access."

The last function call at sub\_401174 is CreateRemoteThread(lpThread, dwCreationFlags,lpParameter, lpStartAddress, dwStackAddress). The purpose of CreateRemoteThread creates a thread that runs in the virtual address space of another processor and optionally specifies extended attributes. The program will create a remote thread for winlogon.exe and inject sfc\_os.dll into the thread. In this case, sfc\_os.dll is the key to the malware because it is related to Windows File Protection. Recall that sfc\_os.dll is combined with value "2" at LoadLibrary function. I assume number 2th exports of sfc\_os.dll might damage the protection mode if it didn't run properly. So far I didn't find anything about (2, sfc\_os.dll). Therefore the conclusion is more based on my assumption.

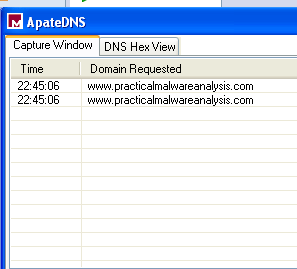


We keep analyzing the third section at main function. The program create two files. The first file already exists. It is wupdmgr.exe at C:\Windows\System32. Therefore the program might want to replace the old version by the new infected wupdmgr.exe. The second file is stored at %s which is temp location. The file is winup.exe stored at lpNewFileName. After the file is create, the program calls sub\_4011FC where extract the file hidden at the resource section.

At sub\_4011FC, the program format the string "C:\Windows\System32\wupdmgr.exe" again

and get the handle from wupdmgr.exe in order to perform injection. It extracts the file from resources section BIN and writes it to wupdmgr.exe. That is how the malware replaced the old version of wupdmger.exe.





BIN is not complicated compared to Lab12-04.exe. I found that BIN is identical with the new version of wupdmgr.exe. The malware overwrite wupdmgr.exe by BIN. The program firstly formats the string Temp\winup.exe and then takes it as parameter to call WinExec. The purpose is to run the original version of Window Update binary. Then the program downloads updater.exe from www.practicalmalwareanalysis.com. The path of updater.exe is same with wupdmgr.exe. But if the user decides to download the executable file, the executable will be put into C:\Windows\System32\wupdmgrd.exe. There is a difference between wupdmgrd.exe and wupdmgr.exe. The former one could be more malicious that updates the malware. The message box doesn't give an option to download it or not. The updater.exe is downloaded manually. The only option is to install it or not install it.

**Issue or Problem**

This lab is really difficult because it contains a lot of knowledge that I am not familiar with. The first issue is (2, sfc\_os.dll). I can't figure out what value 2 represents for. Besides, the program format C:\Window\System32\wupdmgr.exe at main function. What if the program passes it as parameter to sub\_4011FC instead of formatting the string again at sub\_40011FC? I think the author might have another intention but I cannot figure it out.

**Conclusion**

This lab is designed to disable Windows File Protection to wupdmgr.exe in order to overwrite it. It injects a malicious file from resource section to wupdmgr.exe in order to download an updater from www.practicalmalwareanalysis.com.

**Reviewed Questions**

1. What does the code at 0x401000 accomplish?

function at 0x401000 search the process of winlogon.exe and return the PID.

**2. Which process has code injected?**

winlogon.exe

**3. What DLL is loaded using LoadLibraryA?**

sfc\_os.dll

**4. What is the fourth argument passed to the CreateRemoteThread call?**

The 4th argument is the pointer to sfc\_os.dll.

**5. What malware is dropped by the main executable?**

If we use resource hack, the dropped file is called BIN. If we run the malware and let it drop the file automatically, the file will be overwritten version of wumpdmgr.exe.

**6. What is the purpose of this and the dropped malware?**

This malware will disable Window File Protection in order to perform injection and overwrite the original wupdmgr.exe. The dropped malware will download an updater from www.praticalmalwareanalysis.com in order to update the malware and download malware(s).