ECE 8830/CPSC 8810 - Final Exam

Malware Reverse Engineering

Due: 11:59 PM, 05/01/2021

1. (65 points) Short answer questions:

a. What are the differences between malloc() and VirtualAlloc()?

malloc() is a high-level language standard library function that is used to allocate the requested memory and the return value is a pointer to it. This memory allocation is done dynamically. It returns NULL if the requested memory is failed to be allocated. This function has 1 parameter: size.

Whereas,

VirtualAlloc() is a windows API function. It is used to allocate memory in the address space of its own process. It can be used to reserve, commit and change the state of a region of pages in the virtual address space of the calling process. The return value here is the base address of the allocated region of pages. This function has 4 parameters: lpAddress, dwSize, flAllocationType, flProtect.

b. What networking APIs are used create a server listening on a TCP port?

Following are the APIs that are involving in creating a server that would listen on a TCP port:

- Socket() from Winsock2.h: With the parameters "PR_INET, SOCK_STREAM, IPPROTO_TCP". A TCP socket can be created
- Bind() from Winsock.h: It is used to bind the socket to the TCP listening port after setting the port number.
- Listen() from Winsock2.h: Now, we use this to prepare the socket to listen for connections.
- Accept() from Winsock2.h: It is now used to accept incoming connections. It also blocks the process until an incoming connection is received. The socket descriptor is returned for the accepted connection.
- Send(), recv() from Winsock2.h: These are used for communicating with the remote host
- Close() from Winsock.h: It is used to close each socket that was opened after it serves it's purpose.

c. What is "x86 calling convention"?

It is a set of rules about how function arguments are placed, where return values go, what registers functions may use, how they may allocate local variables and so on. It governs how functions on a x86 architecture and operating system interact. There are three calling conventions in windows. They are the following:

- cdel: It is the default calling convention for C and C++ programs, the parameters here are pushed onto the stack from right to left. The caller cleans up the stack when the function is complete.
- stdcall: It is the standard calling convention for the Windows API. Here, the callee clean up the stack when function is complete. Any code calling these API functions will not need to clean up the stack because it's the DLLs responsibility that implemented the code for the API function. We can call "ret" with the number of bytes to pop the parameters off the stack.

- fastcall: It specifies that arguments to functions are to be passed in registers when ever possible. This type of calling convention is only specific to x86 architecture. Here, the caller is responsible for cleaning up the stack. The first few arguments are passed in registers along with the additional arguments being loaded from right to left.

d. Assume you have 50 malware samples. How to determine if they are related, created by the same author, or use the same APIs?

In order to find out and cluster the similar malware samples, following attributes of the files can be used:

i)PDP path: If the pdp path in the samples is same, it determines that those sample are compiled and created by the same author. Since, same pdp path of the file denotes that it has been compiled in the same environment.
ii)Imphash: It is the imported hash of all imported libraries in a PE file. If this collection of malware samples has the same Imphash to them, then it indicates that all those samples use the same set of APIs.

Finally, in order to determine if the samples are related, a couple of attributes can be taken into consideration. Such as, they would have the same file size, same section names, same digital signature, same icon, same packer (if packed), same creation date of the file. Now these attributes individually might not be strong enough to say that the samples might be related. But with a combination of these we can determine that they are related. If they satisfy these conditions.

e. What is the problem of the following assembly code? How to correct it?

mov eax, eip

EIP is an instruction pointer, not a general-purpose register. So, “mov eax eip” will not work because there is no need to read the EIP, it is handled by the processor. We cannot read or write directly to EIP as there is no x86 instruction that can do the work.

Now, looks like the motive of this code is to store the value of the instruction pointer in EAX. The correct way to do so would be to create a function that saves the ESP value into EAX and return. Then, call the created function. What this does is, when the function that we created is called, it pushes the next instruction on to the stack (ESP) which is moved to EAX and returned. This solves the purpose without any problem. Correct code would be something like this:

```
Start:

        Call func_for_eip

func_for_eip:

        mov EAX, [ESP]

        ret
```

f. How to load unpacked form of packed malware?

If loading to any tool like debugger/bintext/peviewer is under the question, such that the unpacked form of the malware is a stand-alone program that could be analyzed in itself. Then, one of the ways to get the unpacked form from the packed malware would be to use the debugger to find the original entry point of the malware and then dump the binary from that point. This dumped binary here has a broken Import table that needs to be re-constructed in order to be able to have a complete unpacked malware to load into tools and analyze. One of the tools that can be used to fix the imports of the dumped unpacked binary is "Import re-constructor". Now that's done, this unpacked form of malware can be executed and loaded into the memory.

As an example, for a UPX packed file, here's how you can get to the OEP from the entry point and then dump it. First after encountering pushad -> search for popad ->take the long jump after it ->you reached the OEP.

g. Explain the purpose of the following code.

mov eax, fs:[30h]

mov ecx, [ecx+0x30]

xor eax, eax

mov al, [eax+0x2]

ret

Looks like the code as a whole has no meaningful purpose to it because the load of EAX from fs:[30h] is overwritten by xor-zeroing before the value is used. The next instruction after it, that is " mov al, [eax+2]" is loading from absolute address 2 because EAX is just zeroed. So, I'll just explain my understanding of each instruction below:

mov eax, fs:[30h] – This step looks like it's loading the PEB into EAX

mov ecx, [ecx+0x30] – PEB is moved to ECX by avoiding NULL byte

xor eax, eax – zeroing the value of EAX

mov al, [eax+0x2] – data from absolute address 2 is loaded and moved to al.

ret – function is returned to the caller

h. What are the differences between stack-based buffer overflow and heap-based buffer overflow? What are the mitigation strategies for each?

Stack-based buffer overflow corrupts the memory on the stack which in-turn affects the values of the local variables, function arguments and return addresses. For example, when a function is called which recursively calls itself, there is no termination, leading to stack-based overflow because each function call creates a new stack frame and the stack in this case will eventually take up more memory than what is allocated for it.

Whereas,

Heap-based buffer overflow corrupts the memory located on the heap which in-turn affects the global variables and other program data. For example, heap-based buffer over is caused if we write past the end of an array allocated from the heap.

Mitigation strategies –

For stack-based overflow: We can track the stack pointer and periodically check the location of the stack pointer to record the largest value of it such that we can make sure it does not grow beyond that value.

For heap-based overflow: We can separate the code and data in order to prevent the execution of the payload and randomization can be introduced so that the heap is not found at a fixed offset. This will ensure heap-based overflow is prevented.

i. List the different approaches of process injection on Windows.

Out of the many different approaches used in windows for process injection, here are two of the most common techniques:

- Classic DLL injection via CreateRemoteThread and LoadLibrary: Here, in the virtual address space of another process, malware writes the path to its malicious dll such that when ever this injected process is executed, it also loads and executes the malicious dll along with it. Since this dll execution is being done by a legitimate file. The source of the malicious intent goes undetected. Malware uses “CreateToolhelp32Snapshot”, “Process32First” and “Process32Next” APIs to search through the processes to inject. After the target process is found, handle of the target process is called by using OpenProcess. First, with VirtualAllocEx, space is created in the remote process. Now, with WriteProcessMemory, the path to the malicious dll is written in it. Finally, to execute the code in another process, CreateRemoteThread is used such that remote process executes the dll on behalf of the malware.

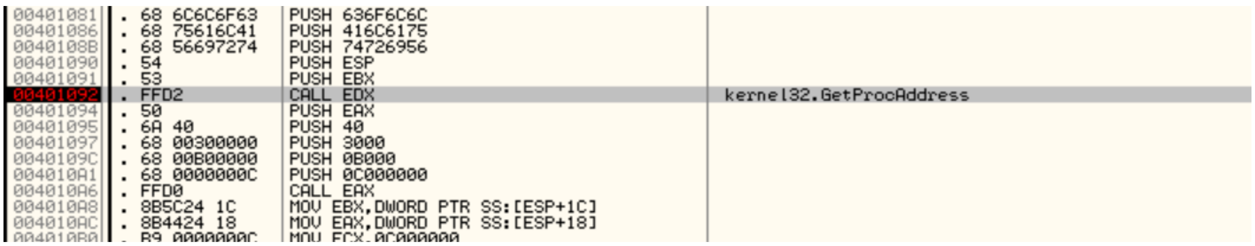
- Portable Executable Injection: Here, malware copies its malicious code into an existing open process and executes it either via shellcode or by using CreateRemoteThread. There is no malicious dll on the disk in this case. It uses VirtualAllocEx to allocate memory in a host process to write its malicious code by calling WriteProcessMemory. By doing this, there is a new base address which is unpredictable and to overcome this, malware finds its relocation table address in the host process and resolves the absolute addresses. This technique of process injection is actually quite popular among crypters.

2. https://clemson.box.com/s/5n7c931o8y1s1n8f9griqeroixluu161

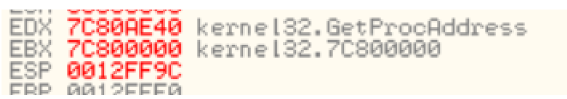
Static analysis is not a good idea for this one. You need to perform some dynamic analysis (35 points).

1) Set a breakpoint at 0x00401092, what is this sample calling?

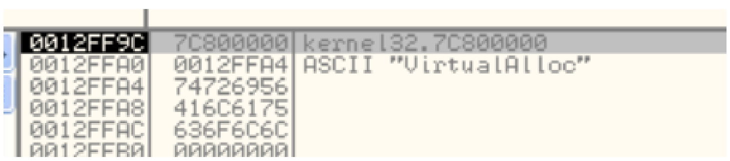
First, we start searching the address “0x00401092” by pressing ctrl+G and entering the address. After placing a breakpoint on it by “F2” and executing until breakpoint “F9” We can see that the sample is making a call to the function “GetProcAddress” from “kernel32.dll”. We can see it in the figure below on the description next to “Call EDX”.



We can also figure out the same by looking at the value stored in EDX. As you can see in the image below.

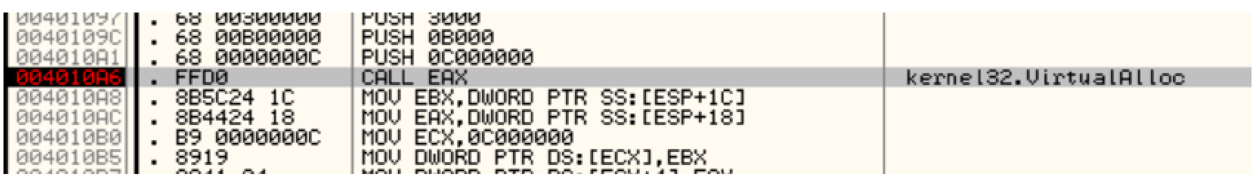


For further confirmation, we can see the API is loaded into the stack (shown in the figure below)



2) What is being called at 0x004010A6? What is the callee doing?

After executing until the address 0x004010A6 we can see that “VirtualAlloc” function is being called by the sample from “kernel32.dll”. We can see the same in the image below.



For further confirmation, we can also see the address stored in EAX that’s being called is of “VirtualAlloc”. It can be seen in the figure below.

In this case, the callee refers to “VirtualAlloc”. The purpose of it in this code is that, it allocates virtual memory in the virtual address space of the calling process and initializes it to zero.

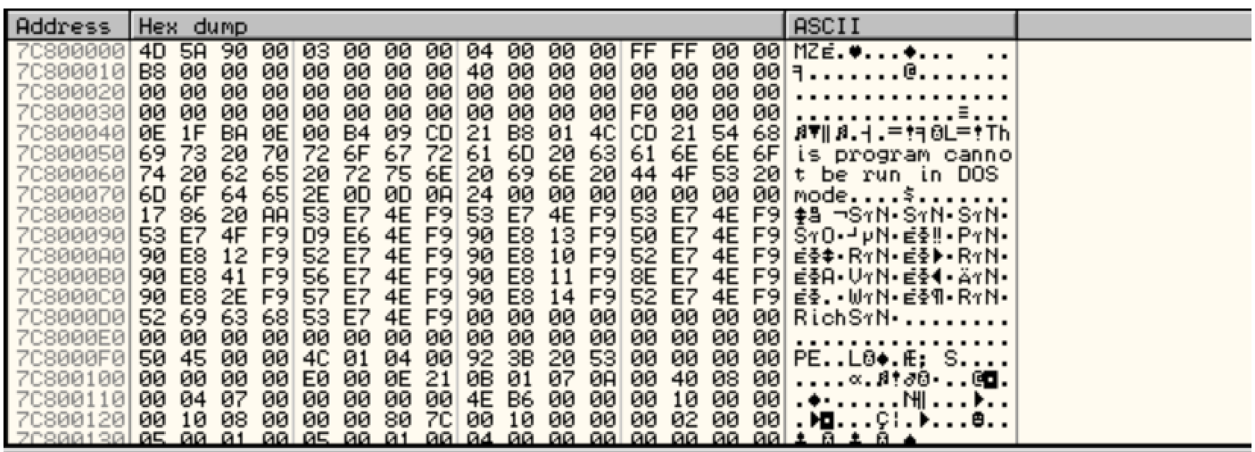
3) What is sub_401360 doing? What about sub_401372 and sub_401388?

sub_401360 – This subroutine invokes the ExitProcess function of the "kernel32.dll". You can say that because the return here takes back the control to the address 0x0040110F.

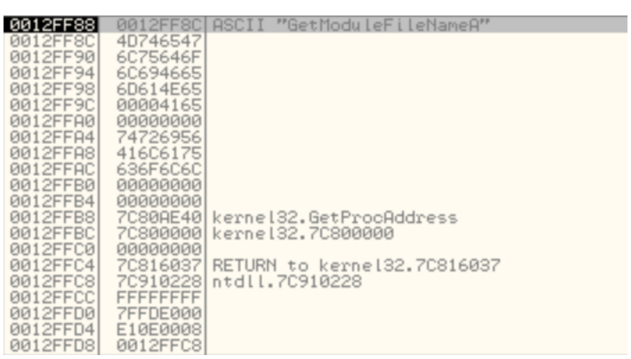
When we step-over and check each instruction in the subroutine, we can see, first, it POPs the the address 0x004010DF out of the stack.

Next, the address 7C800000 which represents DS:[0C000000] is stored into ECX and pushed on to the stack.

When you dump the ECX, you can see the executable file in the dump

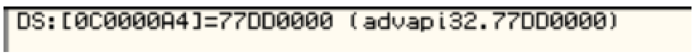


The process address is then called along with the file name and the control flow of the code is returned back to 0x0040110F.



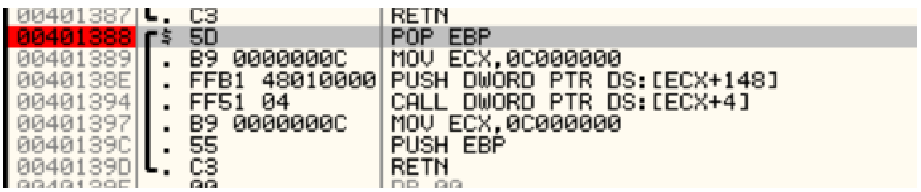
sub_401372 - This subroutine invokes the RegCreateKeyA function of the "advapi.dll". You can say that because the return here takes back the control to the address 0x004011D8.

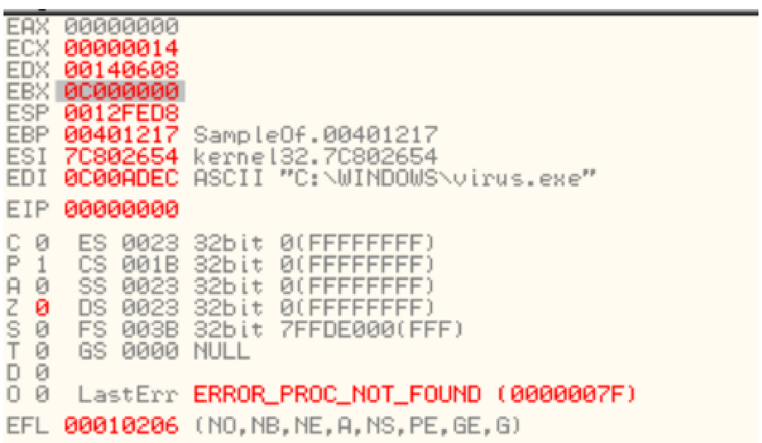
When you look through the subroutine, you can see the function "RegCreateKeyA" which is at the location 0x77DD0000 is being pushed on to the stack and called.



Once done, the control flow of the code is returned back to 0x004011D8.

sub_401388 – In this case, when I try to set a break point on the address 0x00401388 and check what function it is invoking, my analysis is impeded by EIP value pointing to NULL. It is shown in the second image below.

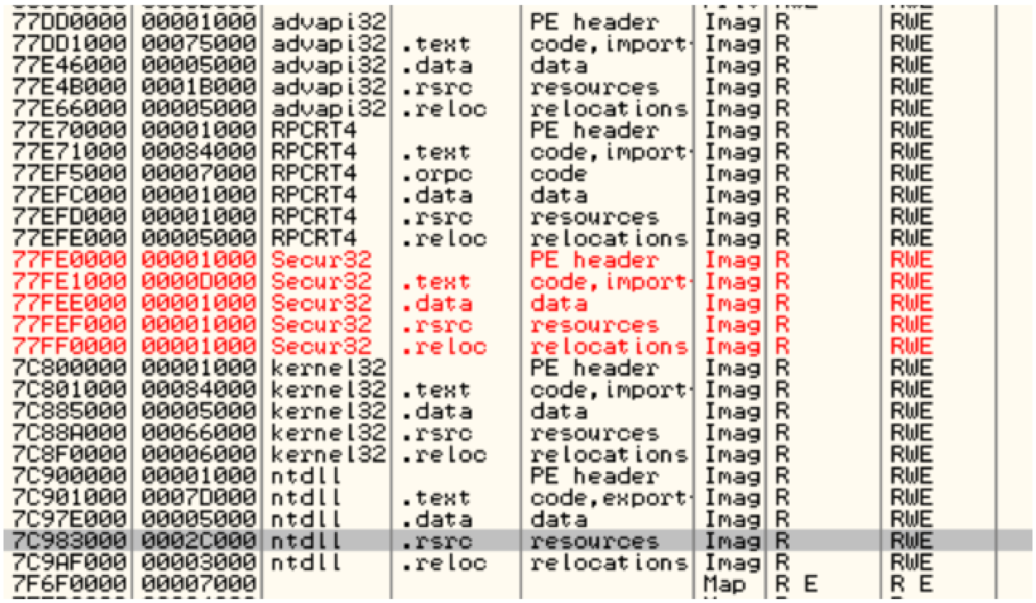




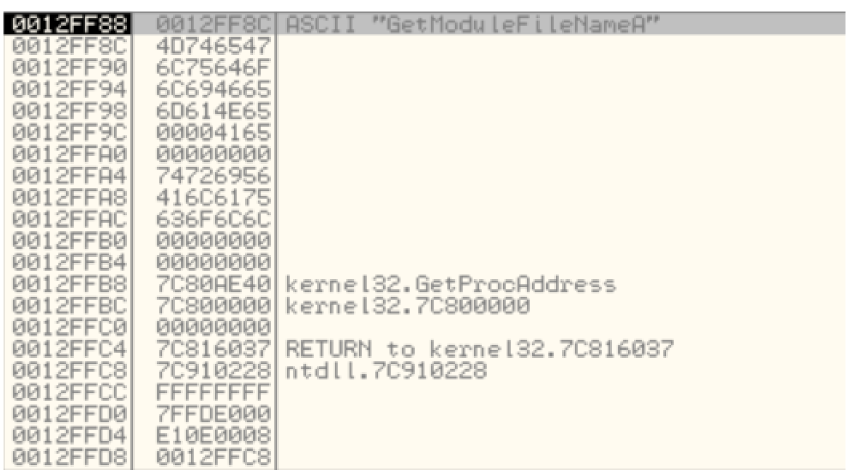
And my execution of the program couldn't go further until the point where I could execute 0x00401388. From what I executed and saw in the code and these three similar structured subroutines is that, all the three subroutines are used to load and execute function calls from the dlls. It is done in this way such that, the dlls are loaded dynamically during the runtime and would be seen while doing static analysis. According to this logic, sub_401388 looks like it is also used for invoking a function call.

4) What Windows API functions did the sample import?

Sample is importing the API functions during runtime, which means we cannot get the imported functions by static analysis. However, we can view the APIs used by the malware sample. This could be done by going to the "Memory map" view in the ollydbg once the sample is loaded into it. It is shown in the image below, that this sample is importing and using the functions from the following APIs: “advapi32”, “RPCRT4”, “Secur32”, kernel32”, “ntdll”.

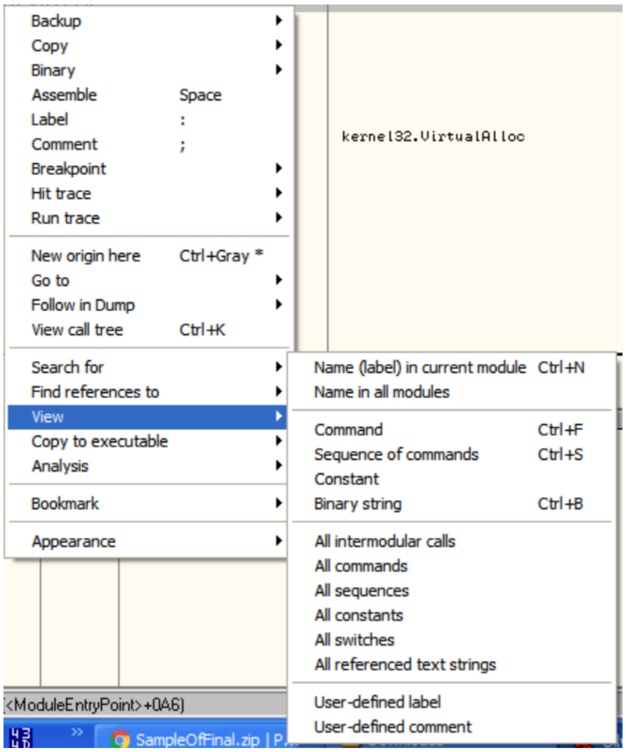


Now, what specific functions from these APIs did the sample import can only be seen while executing the program using step-into and step-over because this sample imports the functions during runtime. So here are some of the import functions I found: RegCreateKeyA, ExitProcess, GetModuleFileNameA, GetProcAddress.

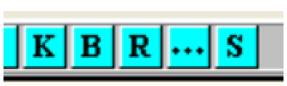


5) How did you find the Imported functions?

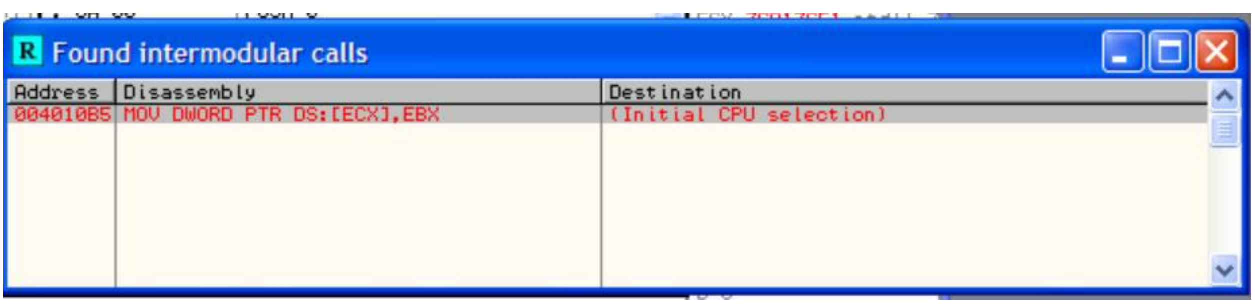
In Ollydbg, we can find the imported functions by going to the “intermodular calls”. We can go to it by right clicking -> Selecting “search for” -> Clicking “All intermodular calls”



Or, we can also clock on the "R" tab in the toolbar of the ollydbg to go directly to the intermodular calls. It is shown in the figure below.



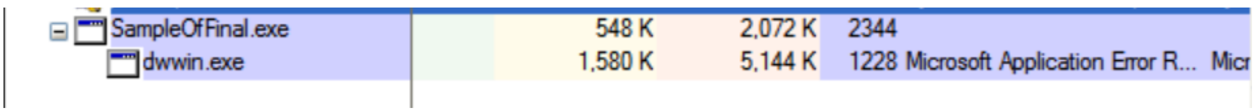
It should ideally show all the imported functions in the box display after performing the steps above. But since this sample loads them during runtime (like how I explained in question 4). In my case, it displays something like this (Figure below). Hence, I checked the import functions by stepping into and stepping over the sample code in Ollydbg.



6) What does this sample do? Is there anything that impeded your analysis? How so? How might you overcome this?

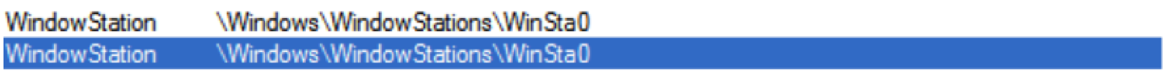
The sample makes a copy of itself to "C:\WINDOWS\" location under the name "virus.exe" (shown below).

And executes with name “dwwin.exe” (shown below).

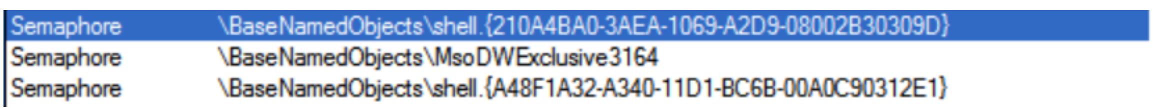


It might be doing this to create persistence. Another persistence technique used by this malware is to setting the registry value in “HKLM\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN" to auto-execute on re-start.

It also adds value to “WinSta0” for being executed on log-on (shown below).

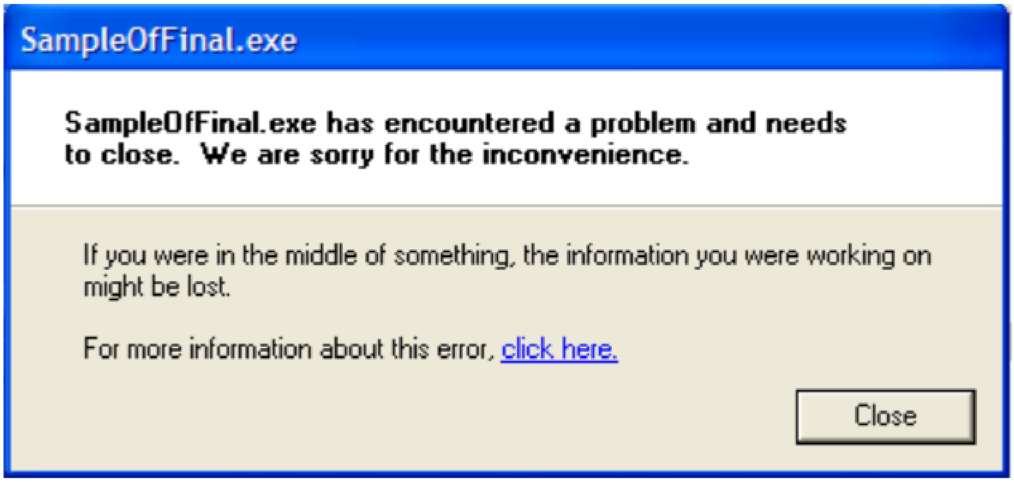


It also looks like the sample is executing shell commands. You can see that from the figure below.



Apart from that, by looking at the calls, the sample also seems to trying to make internet connections. You can see the calls below.

As in all, when you execute this malware, the analysis is impeded because it doesn’t show any GUI of a legitimate application or any activity. An error message is displayed and the process terminates itself after. The error message is shown below



This might be cause the sample malware is having anti-malware techniques involved in it. But we can overcome this, one way of removing the evasion techniques here would be by replacing the registers and the lines of logics used for the check, in assembly language with NOP values. Saving the changes then and executing the saved file would show us the actual intention of the sample.

7) What do you think is the purpose of this malware?

This malware looks like a "Generic Trojan" where it shows the properties of a malicious intent code. Such as, creating persistence, making a copy of itself in the windows location, executing the copy with a different name, executing shell commands and attempting to make connections over the internet. But as in whole, though the analysis was impeded, it looks like its purpose is to download and execute the shellcode to carry out its malicious activity by hiding itself.