

# **Creating and Reverse Engineering an Entry-Point Redirection Keylogger**

Molly Ziegenfelder

CIS4914 Senior Project

University of Florida

Advisor: Dr. Joseph N. Wilson – [jnw@cise.ufl.edu](mailto:jnw@cise.ufl.edu)

Presentation Date: April 22, 2019

## **Abstract.**

The purpose of this project was to update and reverse engineer an endpoint redirection keylogger from an older version of Windows to Windows 10. Although the original worked well on older versions, patches and updates began to interfere with its functionality. My keylogger and injection method is not the exact same as the original, due to differences between previous versions of Windows and Windows 10, specifically upgrades to security and less allowance for users to access administrative files. However, I was still able to create a keylogger dll and an executable that will persistently inject it into a service on a Windows 10 operating system while remaining true in spirit to the original design and execution. Additionally, for classroom purposes, I reverse engineered my keylogger, pointing out the interesting behaviors and how one could find those on their own.

## **Introduction.**

As operating systems grow more robust, oftentimes programs that once worked may no longer run. This is especially true of malware, which may take advantages of flaws or backdoors in a current version that could be changed in future versions. In *Practical Malware Analysis*, written by Michael Sikorski and Andrew Honig, for lab 11-03, which is a keylogger that injects itself using shellcode and endpoint redirection, this was the case. In fact, the service it originally uses to inject the keylogger into doesn't even exist in current Windows versions.

My solution was to recreate the lab dll and executable to work on a more modern operating system, Windows 10. I had access to the original executables but not the source code, so I relied on reverse engineering to determine the original method of operation and attempted to replicate that.

## **Problem Domain.**

This project occupies both the security and reverse engineering area of computer science engineering. In particular, this is a black hat project, which is generally used to point out vulnerabilities in systems by exploiting them. It also is to be used as an instruction tool as a sample for students to learn how to reverse engineer malicious programs.

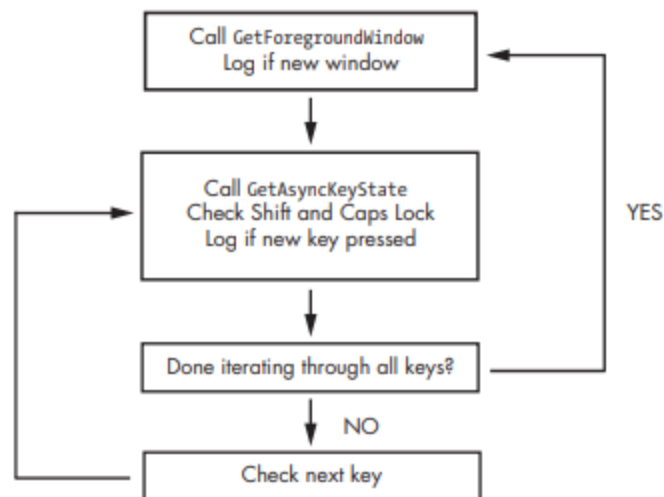
## Literature Search.

The top source I drew from was the same book from which the original lab originated, *Practical Malware Analysis*. Not only did it have guiding questions, but it also went into depth about how keyloggers, shellcode, and backdooring works. Because of the stigma of maliciousness that shellcode has, it was difficult to find legitimate sources regarding it, but gaming forums actually proved to be helpful in my search and pointed to educational tutorials.

## Solution.

### Keylogger Design:

As far as design goes, the keylogger portion of the project was relatively simple. I followed a similar outline to the one pictured in *Practical Malware Analysis* that opens the file to write to, then loops, checking and reporting every time a new window is brought to the foreground and capturing the keystrokes entered in to the current window, as shown in figure 1. To misdirect casual observers, the text file that the key strokes were stored is labeled a dll and written in to the 32 bit system folder, where dlls are not uncommon to find.



**Figure 1:** Flow chart from *Practical Malware Analysis* showing the while loop that the keylogger iterates through.

### Injector Design:

The final design of the executable used a combination of c++ code as well as assembly language that the visual studio compiler converted to shellcode. The purpose of the injector executable was to inject shellcode that would launch the keylogger dll in a persistent manner that

would run every time the service was run, while still remaining covert. Because of the switch to Windows 10 from an earlier version of Windows, the first thing I had to accomplish was to take ownership of the system folder so my executable could have access to a service to inject into. For the purpose of misdirection, I copied the dll into the system folder under a different name so it would appear the malware was loading an unrelated dll. Additionally, Windows 10 would not allow me to modify a service in the folder and instead I had to copy it out, edit it, and then copy it back in under a different name so the original service could still exist.

The bulk of the executable is the portion involving injection of the shellcode. The injection worked by finding a “cave” of unused bytes in the PE file of the service and inserting the compiled shellcode into the unneeded section. Due to padding, most services have plenty of room for injection and though I chose to use nslookup, almost any service would have worked. After inserting the shellcode, the injector altered the PE file’s original entrypoint to point to the injected shell code, so that it would run before the service’s actual code. Finally, the injector started the service, effectively running the shellcode and thus the keylogger dll.

The last piece in the executable is the shellcode itself. As before mentioned, the visual studio compiler offered a feature where it could compile assembly code as shellcode, which I took advantage. Because shellcode must be position independent, it had to dynamically find the address of any functions after it was already loaded into the service’s memory. Additionally, because of address randomization in later Windows versions, hardcoded addresses or even addresses found right before insertion will either fail to work on other machines or fail after rebooting. So, in order to call the keylogger, the shellcode had to dynamically load the LoadLibrary function into memory by accessing the function through kernel32, which is one of the few libraries that can be found in the same place in all executables. I had my shellcode walk through the peb block to find kernel32, get its address, and then get LoadLibrary’s address from that. After calling LoadLibrary to launch the keylogger dll, the shellcode then called the original entrypoint of the service, so the rest of the service could run as normal.

Reverse Engineering:

There are five questions *Practical Malware Analysis* asks about the original malware:

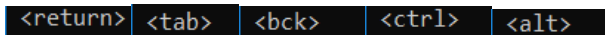
1. What interesting analysis leads can you discover using basic static analysis?
2. What happens when you run this malware?
3. How does Lab11-03.exe persistently install Lab11-03.dll?

4. Which Windows system file does the malware infect?
5. What does Lab11-03.dll do?
6. Where does the malware store the data it collects?

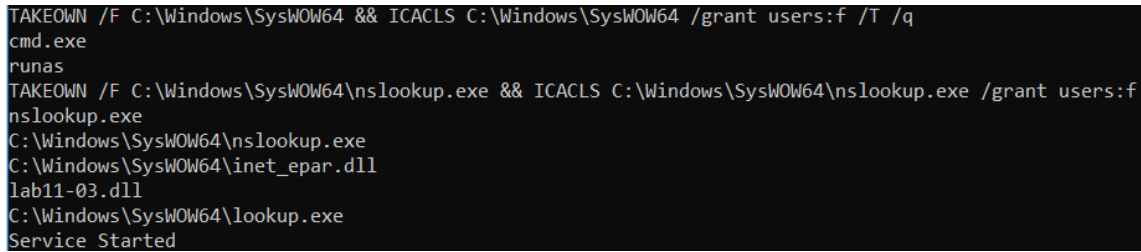
In this section, I will answer those questions as they pertain to my own malware.

### Basic Static Analysis

I started by running strings2 on the files since that is often the quickest and easiest way to learn a lot about a program in a short amount of time. It returned some interesting and telling strings, as shown in figure 3 and 4.



**Figure 2:** These strings found in the dll are strings that are often associated with keyloggers to categorize the different keystrokes which indicates that may be what we're dealing with.



**Figure 3:** These strings from the exe show the program taking ownership of the folder, as well as the service it is interacting with.

Next I looked at the programs in Ida (Appendix A and Appendix B), which confirmed that the dll is a keylogger that continuously loops to find and record the current window, and currently pressed key. It also revealed that the executable creates a heap containing shellcode that runs loadlibrary with a path to a dll, which it appears to insert into the service, thus loading and running that dll.

### Running the Malware

After static analysis has been completed, the next step in the reverse engineering process is to run the malware. Importantly, the malware must be run as an administrator, or else it will be unable to write to the SysWOW64 folder. After running, initially, it appears to do nothing besides output a line saying that a service was started. However opening task manager reveals that nslookup is using much more CPU than usual (figure 4). Additionally, a text file named kernel64x.dll was created (figure 5).

Name	Status	CPU
Microsoft Windows Search Prot...		0%
Microsoft Windows Search Prot...		0%
nslookup (32 bit)		18.0%

**Figure 4:** nslookup usually uses much less CPU, this is a sign that something else is running

```
kernel64x - Notepad
File Edit Format View Help

Inbox - Gmail - Google Chrome: In f <bck> regards to the previous topic we discussed
McAfee AlertViewer:
kernel64x - Search Results in SysWOW64: dll
Cortana: macaff <bck>
lab11-03 - Microsoft Visual Studio : <ctrl>B

Administrator: Command Prompt: c&Administrator: Command Prompt: d <return> &
IDA - lab11-03.exe C:\Users\Molly\source\repos\lab11-03\Release\lab11-03.exe: <ctrl>F
```

**Figure 5:** Kernel64x was created, disguised as a dll file, and records keystrokes

## Persistent Installation

As discussed during static analysis, and shown in Appendix A: figure 7, 8, and 9, the malware persistently installs the dll into the service by injecting shellcode that will run the dll every time the service is run.

## Affected System File

The file shown to be infected by the shellcode in Appendix A, figure 10, is nslookup.exe.

## Dll Operation

In Appendix B, the dll is shown to have many of the hallmarks of a keylogger, and our findings of kernel64x.dll proves that it is logging the user's keystrokes.

## Data Storing

As mentioned, kernel64x.dll, though using a dll extension, is actually a text file that holds the log file that the keylogger creates

## Results.

As is ideal, there is little evidence after running the malware that it is even running. An astute observer may notice that the infected service is using more CPU than it would otherwise, and that some new dlls were placed into the 32 bit system folder, one of which contains a log of

their keystrokes. Furthermore, restarting the computer does not stop this behavior; as long as the infected service is running, the keylogger will run as well.

### **Conclusions.**

The created executable works as desired with some limitations. The copying of files into the system folder is more easily noticed than I'd like, since I had to take ownership of the folder using a command line execution call. Also, due to a greater amount of documentation for 32 bit shellcode, I elected to create both my executable and dll as 32 bit programs. This meant the shellcode could only be injected into a 32 bit service, which unfortunately meant that I did not have the option of selecting a service that was automatically run at startup on Windows 10 by default, so it does not guarantee persistence in that respect. In the future, the option remains to either create 64 bit programs so a 64 bit program could be infected, or to modify registry keys to automatically start the infected service after a reboot to create more persistence.

On the whole, I started this project having never created a keylogger, an injector, or even worked with shellcode. I also had only minimal experience with assembly code, so creating position independent assembly was challenging but was an opportunity to learn a lot about the operating system and how it can be manipulated.

### **Standards and Constraints.**

The main constraint for this project was that it had to be able to run on a Windows 10 operating system. Additionally, I strived to create it so it would be as persistent as possible without having to rerun the executable, even if the user rebooted their computer.

### **Acknowledgments.**

Firstly, I'd like to acknowledge and thank my advisor, Dr. Joseph N. Wilson, for his valuable guidance and feedback throughout the duration of this project as well as for originally sparking my interest in reverse engineering and other security related topics in his class. Additionally, I'd like to express my thanks to my brother for introducing me to computer science and to my fiancé for all his support and encouragement throughout my studies.

## References.

- [1] Dtm. "PE File Infection." *Ox00sec*, 19 May 2016, [0x00sec.org/t/pe-file-infection/401](http://0x00sec.org/t/pe-file-infection/401).
- [2] Honig, Michael Sikorski. Andrew. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st ed., No Starch Press, 2012.
- [3] Ikriv. "Win32: Who Loads the Library That Implements LoadLibrary?" *Ivan Krivyakov*, 3 July 2015, [ikriv.com/blog/?p=1650](http://ikriv.com/blog/?p=1650).
- [4] "Keyloggers: How They Work and How to Detect Them (Part 1)." *Securelist - Kaspersky Lab's Cyberthreat Research and Reports*, [securelist.com/keyloggers-how-they-work-and-how-to-detect-them-part-1/36138/](http://securelist.com/keyloggers-how-they-work-and-how-to-detect-them-part-1/36138/).
- [5] "Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques." *Endgame*, 18 July 2017, [www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process](http://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process).
- [6] Popescu, Ionut. "Introduction to Windows Shellcode Development – Part 3." *Security Café*, 15 Feb. 2016, [securitycafe.ro/2016/02/15/introduction-to-windows-shellcode-development-part-3/](http://securitycafe.ro/2016/02/15/introduction-to-windows-shellcode-development-part-3/).

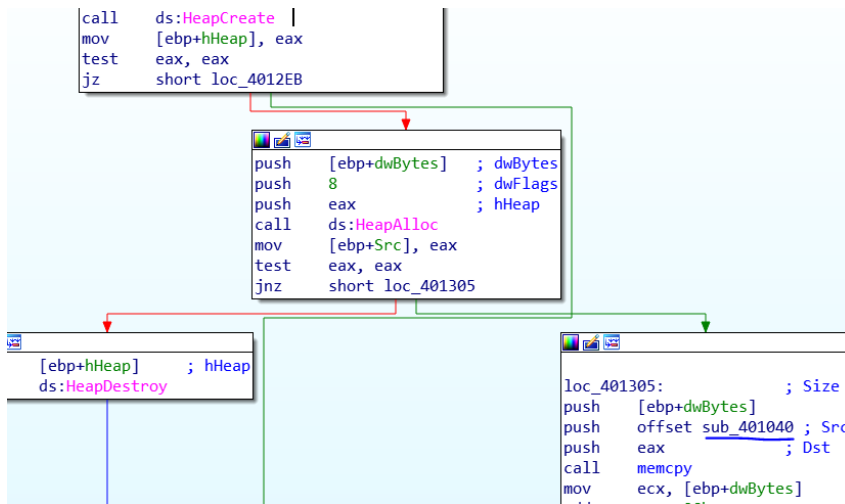


## Appendix A.

Examining the executable in Ida:

```
loc_4015B4:
call    __p__argv
mov     edi, [eax]
call    __p__argc
mov     esi, eax
call    _get_initial_narrow_environment
push    eax
push    edi
push    dword ptr [esi]
call    sub_401150
add     esp, 0Ch
mov     esi, eax
```

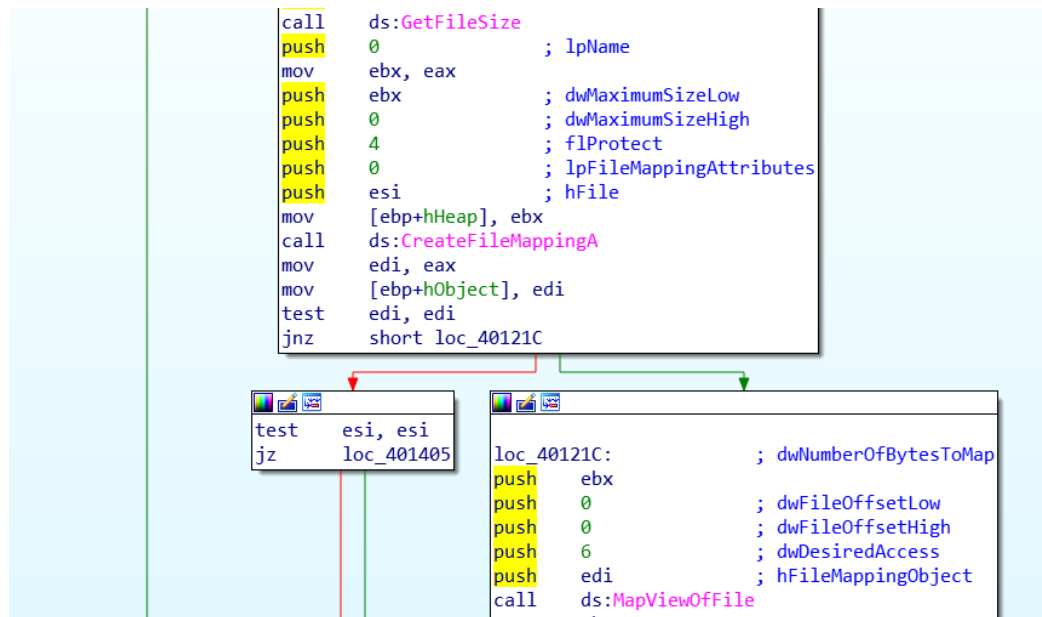
**Figure 6:** From start, we go to sub routine sub\_401150 to find the interesting sections



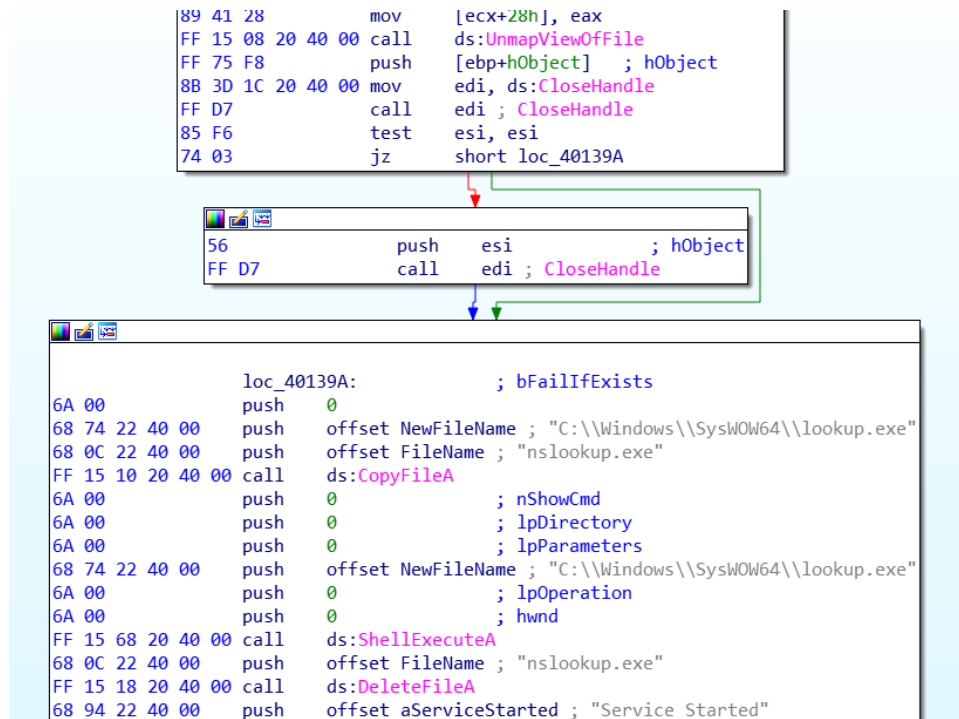
**Figure 7:** The executable is shown creating a heap and copying something in; we can navigate to the source to discover what is being copied.

```
push    ebx
push    edx
push    ecx
push    41797261h
push    7262694Ch
push    64616F4Ch
push    esp
push    ebx
call    edx
add     esp, 0Ch
pop     ecx
push    eax
lea     esi, aCWindowsSyswow_2[ebp] ; "C:\\Windows\\SysWow64\\inet_epar.dll"
push    esi
call    eax
popa
push    0AAAAAAAh
retn
ShellcodeStart endp
```

**Figure 8:** In the source, a method called Shellcode start exists. It consists of assembly code. At the line with a red dot next to it, the hex ascii values for the string LoadLibraryA get pushed onto the stack. Later, a string to a dll is pushed onto the stack and a call is made to eax. It seems safe to say that LoadLibrary is being called with the dll address as its argument, thus running the dll.



**Figure 9:** In this figure, a file mapping of nslookup.exe is being created, this is often used to find room in the file to insert outside code. So, it is a reasonable assumption to make, that the previously found shellcode is being inserted into nslookup.exe



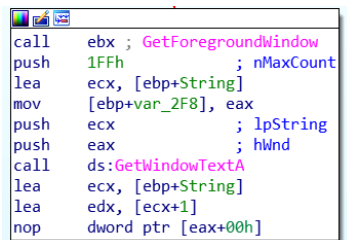
**Figure 10:** Finally, after unmapping the file and performing clean up, the executable appears to copy nslookup.exe back into the sysWOW64 system folder under the new name of lookup.exe, start the service, and then delete the nslookup.exe file still left in the current folder.

## Appendix B.

Examining the dll in Ida:

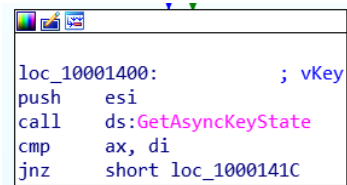
```
68 10 42 00 10    push    offset aCWindowsSyswow ; "C://Windows//SysWOW64//kernel64x.dll"
C7 45 FC 03 00 00+mov    [ebp+var_4], 3
FF 15 40 40 00 10 call    ds:?._Fiopen@std@@YAPAU_iobuf@@PBDHH@Z ; std::_Fiopen(char const *,int,int)
```

**Figure 11:** Knowing that this is likely a keylogger, from strings analysis, it seems likely that this path may be the path to the log file.



```
call    ebx ; GetForegroundWindow
push    1FFh ; nMaxCount
lea     ecx, [ebp+String]
mov     [ebp+var_2F8], eax
push    ecx ; lpString
push    eax ; hWnd
call    ds:GetWindowTextA
lea     ecx, [ebp+String]
lea     edx, [ecx+1]
nop     dword ptr [eax+00h]
```

**Figure 12:** These calls are more evidence towards this dll being a keylogger that records which window the keys are being pressed in.



```
loc_10001400: ; vKey
push    esi
call    ds:GetAsyncKeyState
cmp     ax, di
jnz     short loc_1000141C
```

**Figure 13:** Again, this is a call that only program accessing key information would need, confirming that this must be a keylogger.

**Biography.**

Molly Ziegenfelder was born May 7, 1998 and was homeschooled until her first class at the University of Florida in 2013. She graduated from Santa Fe college with her high school diploma and associates degree in spring 2016 and began her baccalaureate degree in Computer Science Engineering at the University of Florida the following fall, which she expects to graduate from in May of 2019. Ms. Ziegenfelder will be moving to Boston, MA following her graduation to start a new position as a software engineer at the Depositary Trust and Clearing Corporation in July and to marry her soon to be husband in October 2019. She hopes to continue to be able to work on challenging projects throughout her career and has intentions of eventually pursuing an advanced degree in Computer Science.