These next series of posts will focus on explaining a file infector/encrypter that I wrote a week ago or so. It works with any PE32 executable file, overcomes issues with randomized base addresses, and takes advantage of Visual Studio's C++ compiler to generate the assembly code to inject into the target. This allows for large portions of the injected code to be written in C and greatly speeds up development time. Lastly, the target file is also encrypted by the infector and the decryption routine is written in to decrypt the file image at runtime. The series will be broken up into the four parts listed below:

1. Background
2. PE file modification/section injection
3. Writing the compiled stub
4. Full source code and remarks

Since this post will focus on the background of the project, there will be no (relevant) code contained in it. This post will discuss the high level concepts involved behind the infector, issues that arise while developing something like this, and provide an overview of the architecture of the infector. The usual warnings come with this article such as using it only to enhance your knowledge and to not be a script kiddie and rip the code to spread malware.

A file infector is simply an application that adds code to another process in hopes of executing that code. This code can itself be an infector which continues to spread to other files, or it can just be an arbitrary block of code with some defining purpose. Simply introducing code to a file is not enough though, as the normal control flow of the target process would never invoke it. Therefore, there are two main options: parts of the target file can be overwritten with a jump to the code, usually called a code cave. This includes variations such as writing itself into a subroutine and jumping to a block containing parts of the original code. The other option is to hijack the entry point the target file and modify it so the process starts up and immediately executes the desired code. The two techniques are illustrated below:

```
004153AC  . B8 FF000000   MOV EAX,0FF
004153B1  > E8 8F520000   CALL Dbgview.0041A645
004153B6  . C3            RETN
004153B7  $ E8 C4C00000   CALL Dbgview.00421480
004153BC  .^E9 78FEFFFF   JMP Dbgview.00415239
004153C1  $ 8BFF          MOV EDI,EDI
004153C3  . 55            PUSH EBP
004153C4  . 8BEC          MOV EBP,ESP
004153C6  . 51            PUSH ECX
```

The original control flow of an application

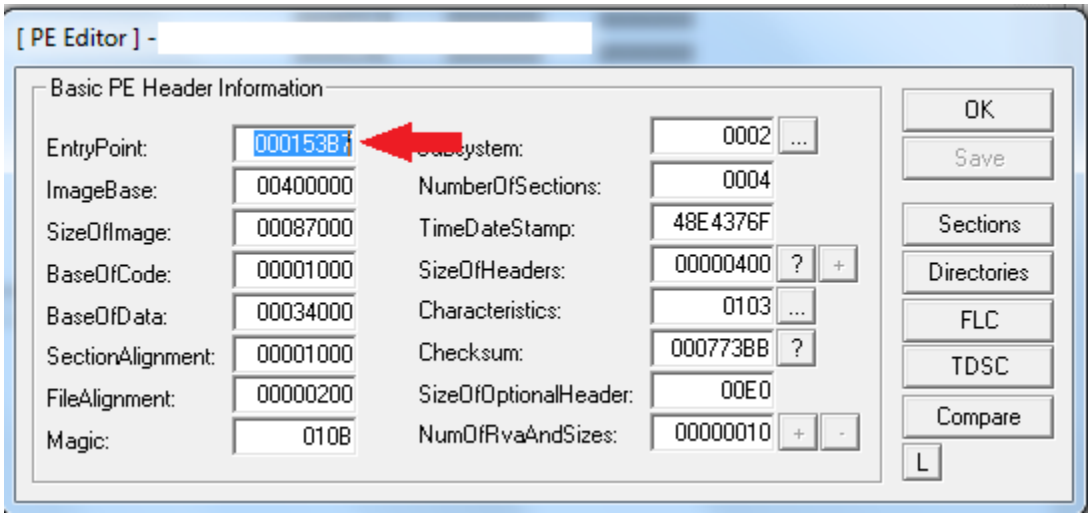
```
004153B1  > E8 8F520000   CALL Dbgview.0041A645
004153B6  . C3            RETN
004153B7   vE9 31E00100   JMP Dbgview.004333ED
004153BC  .^E9 78FEFFFF   JMP Dbgview.00415239
004153C1  $ 8BFF          MOV EDI,EDI
004153C3  . 55            PUSH EBP
004153C4  . 8BEC          MOV EBP,ESP
004153C6  . 51            PUSH ECX
004153C7  . 56            PUSH ESI
004153C8  . 8B75 0C       MOV ESI,DWORD PTR SS:[EBP+C]
```

The hijacked version, with a jump to what was an empty part of the process, but now would contain instructions to execute

```
004333EA    00              DB 00
004333EB    00              DB 00
004333EC    00              DB 00
004333ED  > 60              PUSHAD
004333EE  . 6A 00           PUSH 0                          Style = MB_OK|MB_APPLMODAL
004333F0  . 6A 00           PUSH 0                          Title = NULL
004333F2  . 68 0A344300     PUSH Dbgview.0043340A           Text = "Hello World!"
004333F7  . 6A 00           PUSH 0                          hOwner = NULL
004333F9  . E8 20C9BB75     CALL USER32.MessageBoxA         MessageBoxA
004333FE  . 61              POPAD
004333FF  . E8 7CE0FEFF     CALL Dbgview.00421480
00433404  .^E9 B31FFEFF     JMP Dbgview.004153BC
00433409    00              DB 00
0043340A  . 48 65 6C 6C 6F  ASCII "Hello World!",0
00433417    00              DB 00
00433418    00              DB 00
00433419    00              DB 00
0043341A    00              DB 00
```

The added instructions to be executed. The overwritten code is restored at the end and a jump returns control flow back to normal.

The other mentioned technique, modifying the entry point:



The entry point is an offset from the image base and denotes where the program begins execution. It is possible to take control of the application by modifying the entry point to point to the added code block, then jumping from the added code block to the original entry point. One thing to note though is that the ImageBase value is not always reliable, since applications linked with `/DYNAMICBASE` in Visual Studio (or whatever appropriate linker flag with different compilers) will have a "randomized" base address. This means that the jump back into the original entry point cannot have a hardcoded address (0x00400000 + 0x000153B7 in this case), but instead needs to be found by the injected code at runtime.

The next issue arises when the injected code wants to call any Windows API functions. Load addresses of kernel32.dll, ntdll.dll, and user32.dll are not guaranteed to always be the same, and DLLs such as Ws2_32.dll, Shlwapi.dll, and so on are not even guaranteed to be loaded. This means that call addresses to the Windows API cannot be hardcoded, and it also means that additional DLLs may have to be loaded in order to be their functionality. The good news it that since kernel32.dll is loaded into every process, its load added can be obtained from the process environment block (PEB). Then the export address table (EAT) of kernel32.dll can be walked and the address of LoadLibrary can be obtained to load additional DLLs. All exported functions

in the DLL can be found through the function name table and through the usage of the function and ordinal table to obtain the address (more on this in part 3).

The last issue is that functions in the C runtime cannot be used. Again, this issue arises because of randomized base addresses -- the address of the desired function simply cannot be hardcoded into the piece of code to be injected. This means that the functions will have to be implemented in assembly. This really isn't too bad -- for my version I only implemented strlen and a variation on strcmp, both needed when traversing the function name table.

The architecture of the infector has two main components: the injection function which will be injected into the target, and the code to map the file to memory, add the code, modify the entry point, and so on. The injection function will be entirely self contained, and written in C and assembly. The C compiler will be leveraged to generate the assembly instructions that will be injected into the target. At runtime, the infector will calculate the length of the injection function, modify part of the function to insert the correct entry point offset, write the instructions into the target file, and lastly modify the entry point of the target file to execute the function upon loading. Lastly, the file will be encrypted. The role of the injection function is to decrypt the contents at runtime and continue normal execution.