3-12-2008

Data Hiding in Windows Executable Files

DaeMin Shin
Korea University

Yeog Kim
Korea University

KeunDuck Byun
Korea University

SangJin Lee
Korea University

DOI: 10.4225/75/57b2748e40cc0
6th Australian Digital Forensics Conference, Edith Cowan University, Perth Western Australia, December 3rd 2008.

Data Hiding in Windows Executable Files

DaeMin Shin, Yeog Kim, KeunDuck Byun, SangJin Lee
Center for Information Security Technologies (CIST),
Korea University, Seoul, Republic of Korea
{grace_rain, yeog, gdfriend, sangjin}@korea.ac.kr

Abstract

A common technique for hiding information in executable files is the embedding a limited amount of information in program binaries. The hiding technique is commonly achieved by using special software tools as e.g. the tools presented by Hydan and Stilo in (Rakan, 2004, Bertrand, 2005). These tools can be used to commit crimes as e.g. industrial spy activities or other forms of illegal data access. In this paper, we propose new methods for hiding information in Portable Executable (PE) files. PE is a file format for executables used in the 32-bit and 64-bit versions of the Windows operating system. In addition, we discuss the analysis techniques which can be applied to detect and recover data hidden using each of these methods. The existing techniques for hiding information in an executable file determine the total number of bytes to be hidden on the foundation of the size of the executable code. Our novel methods proposed here do not limit the amount of hidden code.

Keywords

Portable Executable (PE), Executable file, Program binaries, Hiding information

INTRODUCTION

It is general in hiding information to embed a limited amount of information in media such as image and music files, so the embedding methods could be under surveillance from system managers in an organization that requires the high level of security. This fact requires researches on new hiding techniques and cover objects which hidden information is embedded in. It is the result from the researches to embed information in executable files. It can be considered that Stilo and Hydan represent common techniques for the embedding information in executable files. These techniques make original files modified, so code signing techniques that guarantee the integrity of the code can be used for the detecting hidden information. But, it becomes general phenomenon making executable files as the level of using computer and computing environment has been raised. In addition, it has not been general to use code signing techniques.

Silo and Hydan modify program binaries that have been in optimization, so the performance of the program binaries may fall. In addition, the amount of information to be embedded in executable files by using Silo and Hydan is limited because these tools determine the number of bytes to be hidden on the foundation of the size of the program binaries. Therefore, it needs to carry out researches on new hiding techniques that consider the efficiency of program binaries and the amount of information to be hidden. In this paper, we examine the new methods that consider the efficiency and the amount of information to be hidden. Further we discuss the analysis techniques which can be applied to detect and recover data hidden using each of these methods.

PORTABLE EXECUTABLE (PE)

The Program Loader that is a subset of the Windows System assumes the loading executable files into a virtual memory, so the executable files have the format that the Program Loader can identify, and the format is called PE (Portable Executable). It is necessary to know the PE format and RVA which is an address type used in the PE in order to understand the new methods for hiding information in the PE, so we briefly describe the format and the address type.

RVA

RVA is a position unit in the PE, and the RVA is used as an offset from the start-address of a PE file loaded on the memory. The start-address of a file on the memory is in ImageBase that is one of the attributes of the PE file. For instance, if the ImageBase of a file is 0x00400000 and one position of the file is 0x1000(RVA), the position on the memory will be 0x00401000.

PE Format

The header of PE format starts with MS-DOS stub that is used for printing a message, “This program cannot be run in DOS mode”, if the operating system can’t identify the PE on execution time.

IMAGE_NT_HEADER located in the position after the MS-DOS stub has the information for the execution of a file, and consists of IMAGE_FILE_HEADER and IMAGE_OPTIONAL_HEADER. The IMAGE_FILE_HEA-DER has the information on the file, such as create time and machine type. The IMAGE_OPTIONAL_HEADE-R has the information on functions used in the file and on the start-address of the file on a memory, and the infor-mation is managed by IMAGE_DATA_DIRECTORY.

A PE file except the header is composed of several sections that are basic unit of code or data within a PE or COFF file (Microsoft, 2006). IMAGE_SECTION_HEADER that is located in the position following to IMAG-E_OPTIONAL_HEADER has the information on each section. The information consists of PointerToRawData, SizeOfRawData, VirtualAddress, and VirtualSize. The PointerToRawData and the SizeOfRawData respectively mean the position of each section and the size of each section on the file. The VirtualAddress and the VirtualSize respectively mean the position of each section and the size of each section on the memory. The size of each section on the file is a multiple of FileAlignment that is in IMAGE_OPTIONAL_HEADER. If the amount of the data of a section is smaller than the size of the section that is allotted on compile time, the slack space of the section occurs.

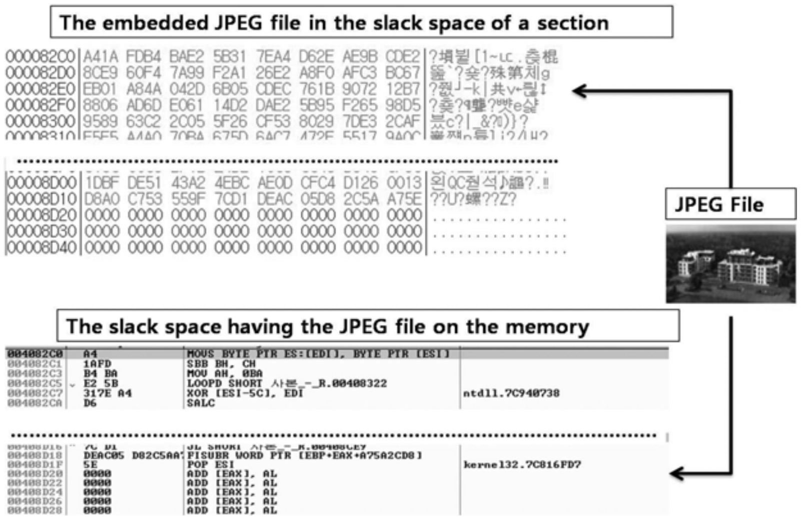
The common sections used in the PE include a .text that has program binaries, .data, .idata that has information on export and import functions, .edata, and .rsrc section. An .idata section has the information on import functions used in executable files during the period of an execution. When loading a file on a memory, the Loader refer to the .idata section to move the address of each import function used in the file on Import Address Table(IAT) which the IMAGE_DATA_DIRECTORY of the file has the position of.

NEW METHODS FOR HIDING INFORMATION

In this section, we will see two methods that embed information in executable files for hiding information. First method is the using the slack space of each section. Since slack spaces are the unused space of each section, it has an advantage that does not generate error on execution time to embed information in the slack space. The method, however, is not applicable to the case when much information has to be embedded. Second method is the embedding information in the .text section of executable files. This method increases the entire size of executable files, and the increased size affects the sections that have been located in the position following to the .text section, which makes the executable files unexecuted. Therefore, it needs extra tasks to execute clearly the executable files of which the information has been embedded in the .text section, but this method has advantages that can maintain the original program binaries that have been optimized on compile time, and do not limit the amount of information to be hidden

Embedding in Slack Spaces

Slack spaces are filled with 0x00 by less than the FileAlignment (Goppit, 2005). The upper picture in figure 1 shows that an encrypted jpeg file has been embedded in a slack space. The slack space is from 0x82C0 to 0x8FF0, and the first part that is not filled with 0x00 in the upper picture is the embedded jpeg file, and the following part that is filled with 0x00 is the slack space. The under picture in figure 1 shows the slack space loaded on the memory. It doesn’t generate error on execution time to embed information in slack spaces for hiding information. We have to know the start position of the slack space of each section in the executable file and the encrypt key that is used for encrypting information to extract hidden information.


The embedded JPEG file in the slack space of a section
JPEG File
The slack space having the JPEG file on the memory

```
000082C0|A41A FDB4 BAE2 5B31 7EA4 D62E AE9B CDE2|?塡뷜[1~ㄷ.춪棍
000082D0|8CE9 60F4 7A99 F2A1 26E2 A8F0 AFC3 BC67|鑑`?숒?殊箕제g
000082E0|EB01 A84A 042D 6B05 CDEC 761B 9072 12B7|?쬢┘-k|共v←뤏↕
000082F0|8806 AD6D E061 14D2 DAE2 5B95 F265 98D5|?숒?¶鸚?뺫e샾
00008300|9589 63C2 2C05 5F26 CF53 8029 7DE3 2CAF|븄c?|_&?)}?
00008310|F5F5 A4A0 70BA 675D 6AC7 472F 5517 9A0C|[illegible]
..............................................................
00008D00|1DBF DE51 43A2 4EBC AE0D CFC4 D126 0013|왼QC쥔석♪譴?.‼
00008D10|D8A0 C753 559F 7CD1 DEAC 05D8 2C5A A75E|??U?蝶??Z?
00008D20|0000 0000 0000 0000 0000 0000 0000 0000|................
00008D30|0000 0000 0000 0000 0000 0000 0000 0000|................
00008D40|0000 0000 0000 0000 0000 0000 0000 0000|................
```

```
004082C0   A4               MOVS BYTE PTR ES:[EDI], BYTE PTR [ESI]
004082C1   1AFD             SBB BH, CH
004082C3   B4 BA            MOV AH, 0BA
004082C5 ˅ E2 5B            LOOPD SHORT 사본_-_R.00408322
004082C7   317E A4          XOR [ESI-5C], EDI                        ntdll.7C940738
004082CA   D6               SALC
..............................................................
00408D16   7C D1            JL SHORT 사본_-_R.00408CE9
00408D18   DEAC05 D82C5AA7  FISUBR WORD PTR [EBP+EAX+A75A2CD8]
00408D1F   5E               POP ESI                                  kernel32.7C816FD7
00408D20   0000             ADD [EAX], AL
00408D22   0000             ADD [EAX], AL
00408D24   0000             ADD [EAX], AL
00408D26   0000             ADD [EAX], AL
00408D28   0000             ADD [EAX], AL
```

Figure 1: Embedding a JPEG file in the slack space of a section.

Embedding in .Text section

This section introduces the method for embedding information in a .text section, and figure 2 shows the flow diagram of the method. First of all, we have to secure the enough space in a .text section in which information has to be embedded for hiding it, and then modify the section table that has the size and the position of each section, and the IMAGE_OPTIONAL_HEADER that has the size of the entire file. Once the information by 4096 byte has been embedded in the .text section, the PointerToRawData of both .rdata and .data section located in the position following to the .text section are moved back by 4096 byte due to the embedding. It is needed to modify the positions information of the sections in the section table. In addition, since the entire size of the executable file has been changed, we have to modify in the IMAGE_OPTIONAL_HEADER both the SizeOfCode that has the sum of all section sizes and the SizeOfImage that is the enough size which the executable file needs on the memory to be loaded. Otherwise, the executable file may have the mark of the embedding because of the difference between the entire sizes of the file with the size information that exists in the IMAGE_OPTIONAL_HEADER.

Then, we have to modify the values of the Import Table that has been moved back, the position of the table, the values of the IAT that has the addresses of the import functions used in the executable file, and the position of the IAT. The Import Table has the information of the functions used in the executable file. The Loader on execution time locates the address of each function in the IAT by the referring to the Import Table.

Finally, we need a procedure that confirms whether or not absolute addresses used in the .text section change due to the embedding data. If the absolute addresses have changed, we have to make the absolute addresses modified applicably to refer to the reference positions prior to the embedding, for which we can use a .reloc section. The absolute address decided on compile time is the reference position on the memory, and it is used in global variables, function pointers, flow control instructions (break, continue, goto, throw), CALL, JMP, and so on.

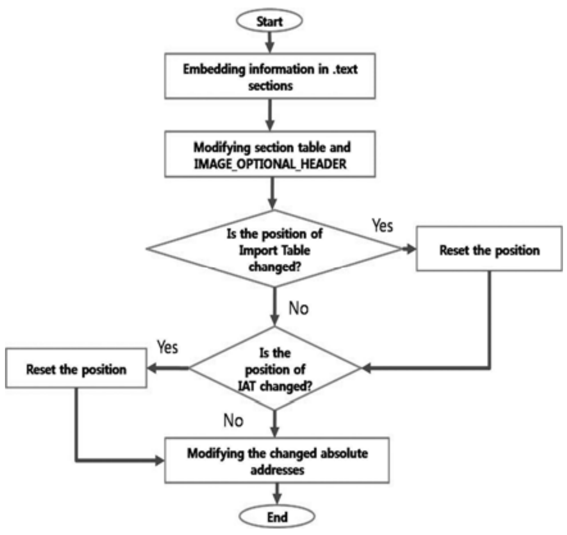

Start
Embedding information in .text sections
Modifying section table and IMAGE_OPTIONAL_HEADER
Is the position of Import Table changed?
Yes
Reset the position
No
Is the position of IAT changed?
Yes
Reset the position
No
Modifying the changed absolute addresses
End

Figure 2: The flow diagram for the embedding process in a .text section

The .reloc section (Microsoft, 2006) is the array of the basic relocation block shown in figure 3. The Page RVA of the block is the start position of the program binaries which this relocation block applies to. The Block Size of the block is the size of this block. The last array consisting of 2byte offsets has the position of each absolute address on the file, and each offset is a distance from the Page RVA to each absolute address. Therefore, we can modify changed absolute addresses on the embedding time by using the .reloc section if the embedding data in the executable file makes the absolute addresses changed.

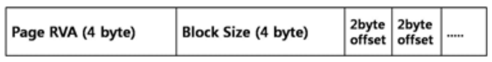


Figure 3: Basic relocation block

Although this method has a disadvantage, which requires additional tasks for a clear execution due to deliberately modifying the .text section, it does not limit the amount of information to be hidden. In addition, it does not make differences in the execution result between original files and the embedded files.

If an executable file that has already existed is used for hiding information, it can be easily detected by code signing techniques or by comparing with attributes such as the file size of same files. Thus, it makes sense that the executable file that information is to be embedded in has to be unique.

It is most cases to embed encrypted and compressed information in a .text section in order to have a difficulty in recovering the information when the information is detected. It can be, therefore, used as a detecting method to verify that the format of program binaries consists of instructions and the operands of them by applying disassemble softwares to the executable file, so it requires that the method embedding information in a .text section have a countermeasure against discriminating the information from the program binaries by using disassemble softwares.

Embedded information can be translated into the format of program binaries by using a method that inserts the embedded data in the operands of instructions. The process of the method is shown in figure 4, and figure 5 shows the embedded information that is on the black underlines after the process. As a result, it can highly lower a possibility that the embedded data is discriminated from the program binaries by using disassemble softwares to translate the embedded data into the form of program binaries.

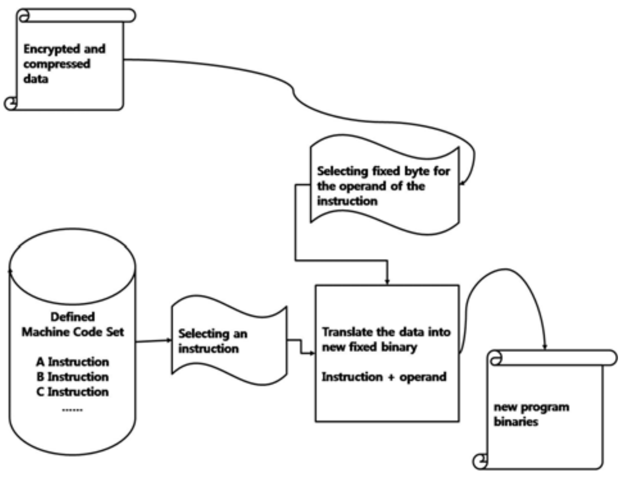
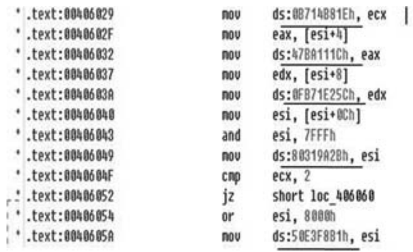


Figure 4: The process for translating embedded data into new program binaries.



```
.text:00406029      mov     ds:0B714B81Eh, ecx
.text:0040602F      mov     eax, [esi+4]
.text:00406032      mov     ds:47BA111Ch, eax
.text:00406037      mov     edx, [esi+8]
.text:0040603A      mov     ds:0FB71E25Ch, edx
.text:00406040      mov     esi, [esi+0Ch]
.text:00406043      and     esi, 7FFFh
.text:00406049      mov     ds:80319A2Bh, esi
.text:0040604F      cmp     ecx, 2
.text:00406052      jz      short loc_406060
.text:00406054      or      esi, 8000h
.text:0040605A      mov     ds:50E3F8B1h, esi
```

Figure 5: The embedded data after the process in figure 4.

Encrypt key and the start position of embedded data in a .text section are needed in an extraction process. In the process, the first of all, the .text section in which information is embedded is parsed from the executable file. Then, the embedded data is extracted from the parsed .text section by using previously shared key and the start position of the hidden information.

ANALYSIS TECHNIQUES

In this section, we discuss analysis techniques which can be applied to detect and recover information hidden using proposed here new methods. Figure 6 shows a flow diagram for the analysis process.

Each data of sections is in fixed format as e.g. program binaries or relocation blocks in a relocation section. Hidden information in section slacks is generally encrypted and compressed, so it can helps decide whether or not each section has hidden information to compare the format of each section and the values of each section.

Program binaries are composed of several basic blocks that consist of instructions. The starting position of all basic blocks in program binaries must be referred to by instructions relating to jump as e.g. CALL, JMP, JNZ. Otherwise, the basic blocks of which the instructions in the program binaries don’t refer to the starting position may be unused parts that have hidden information in the file, so it is used as an analysis technique that can be applied to detect the hidden information in the .text section to confirm that the starting position of all basic blocks is referred to by the instructions relating to jump.

The reference range of the operands of instructions relating to jump in program binaries is limited within the memory area assigned to the file on execution time, but since hidden information is encrypted and than used as the operand of the instructions, the reference range of the operands can be out of the memory area assigned to the file on execution time. It is, therefore, used for an analysis technique to confirm that the reference range of the operand of all instructions is within the memory area assigned on execution time.

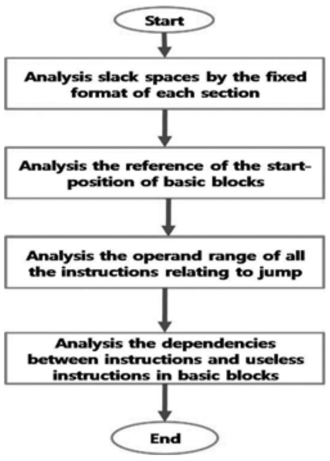

Start
Analysis slack spaces by the fixed format of each section
Analysis the reference of the start-position of basic blocks
Analysis the operand range of all the instructions relating to jump
Analysis the dependencies between instructions and useless instructions in basic blocks
End

Figure 6: Analysis techniques for the detection.

Typically a compiler's back-end goes through 4 phases, instruction selection, register allocation, instruction scheduling, code layout. Instruction scheduling phase finds all possible orders for an instruction process considering a dependency between instructions, and than selects one order of them. Therefore, dependencies between instructions in a basic block exist (Bertrand, 2005). But, it is difficult to make embedded binaries exist with dependencies between instructions on the embedding time due to randomly selected instructions which random information is to be embedded in as operands, so it can be useful to confirm whether or not dependencies between instructions exist in program binaries when analysing suspicious executable files that may have hidden information.

In addition, even though program binaries that are decided on compile time have been in optimization, new program binaries that have hidden information can be composed of useless (suboptimal) instructions by randomly selecting instructions. Thus, it can be an analysis technique to confirm useless instructions constantly exist in program binaries.

CONCLUSION:

New method proposed in this paper is the advanced result from previous researches on the embedding information in executable files for hiding information. In addition, it has the strong advantages that don’t limit the amount of information to be hidden and don’t alter optimized program binaries for the hiding information.

Although analysis techniques that we discussed may be helpful in detection and recovery of the hidden data, the process is time consuming without automated softwares. Thus, in the future, we have to carry out the researches on automated software development and recovery techniques from detected information.

ACKNOWLEDGEMENT:

This research was supported by Bio R&D program through the Korea Science and Engineering Foundation funded by the Ministry of Education, Science and Technology (M10640030004-08N4003-00410).

REFERENCES:

Rakan El-Khalil and Angelos D. Keromytis. (2004) Hydan: Hiding Information in Program Binaries, In *Proceedings of the 6th International Conference on Information and Communications Security (ICICS)*, Pages 187-199.

Bertrand Anckaert, Bjorn De Sutter, Dominique Chanet, and Koen De Bosschere. (2005) Steganography for Executable and Code Transformation Signatures, In *Proceedings of the 7th Information Security and Cryptology (ICISC)*, Pages 431-445

Microsoft Corporation. (2006) Microsoft Portable Executable and Common Object File Format Specification Revision 8.0 May 2006

Goppit. (2005), Portable Executable File Format-A Reverse Engineer View. *Code Breaker Journal*, vol 2, No 3, August 2005