Hyperion: Implementation of a PE-Crypter

Christian Ammann

May 8, 2012

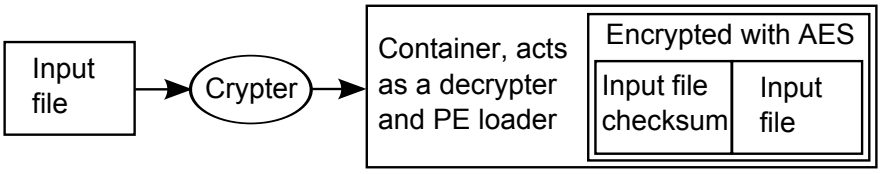

Figure 1: Brief Overview of the PE Crypter Workflow

1 Introduction

Runtime crypter accepts binary executable files as input and transforms them into an encrypted version (preserving its original behaviour). The encrypted file decrypts itself on startup and executes it's original content. This approach allows the deployment of malicious executables in protected environments: A pattern based anti virus (AV) solution detects the signature of suspicious files and blocks their execution. The encrypted counterpart contains an unknown signature, it's content can not be analysed by heuristics and is therefore executed normally without an intervention by the AV scanner. Other uses are protection of binaries against reversing or the replacement of the encryption routine with a packer to reduce the size of an executable.

This paper reveals the theoretic aspects behind run-time crypters and describes a reference implementation for *Portable Executables* (PE) [1] which is the windows file format for dynamic-link libraries (DLLs), object files and regular executables. The encryption of Windows executables requires a general understanding of the following aspects:

- PE layout: The PE header, section headers and data directory entries.
- PE loader: How and where are process images loaded and executed in virtual memory.

We give a beginner friendly introduction to these two important topics in section 2. Afterwards, we present and explain the PE crypter reference implementation *Hyperion* in section 3 for 32-bit executables which can be divided into two parts (see figure 1 for details): A crypter and a container. The crypter (which is explained in more detail in section 3.1) gets a PE binary as input, copies the complete input file into memory, calculates a checksum and prepends the checksum to the input file. Afterwards, a random key is generated which is used to encrypt the checksum and the input file with the AES-128 [2] encryption algorithm. Finally, the encrypted result is copied into the containers data section.

The container (described in more detail in section 3.2) acts as a decrypter and PE loader: It copies the encrypted input file into memory, decrypts it and starts the execution. The decryption key is missing in the container, Therefore, it has to perform a brute-force search through the (reduced) key space using the checksum to verify whether a

key was correct or not. At first sight this is a disadvantage because the encrypted executable needs additional time on startup for the decryption. On the other hand, it is a great protection against static and dynamic analysis of anti virus products.

The crypters run-time workflow is another advantage of our approach: The container is provided as flat assembler (FASM) [3] source code. The encrypted executable is converted into a FASM source code representation (e.g. ”db 0x4d, 0x5a, 0x00, 0x00, ...”) and stored in the *input.asm* file. Input.asm is copied into the containers source code directory and is included in the containers source code with the *include* directive (e.g. include ”‘input.asm”’). Finally, the crypter calls the assembler which generates the corresponding binary.

In comparison, an injection of the encrypted input file into a binary form of the container (e.g. container.exe) would make it necessary to patch big parts of the container manually (image base, section sizes, etc). Our approach delegates these tasks to the compiler which reduces the complexity of the crypter and simplifies extendability and maintainability.

Some aspects like polymorphism and anti-heuristic are still missing in our implementation. Therefore, we present and discuss further work in section 4.

2 Portable Executables and the Windows PE Loader

This section describes the the format of windows portable executables and how they are loaded into memory. There are many papers which cover this topic and we assume that the reader has at least some basic knowledge of modern operating system concepts, like virtual memory, sys calls, etc. Therefore, we give just a brief introduction to the important elements of windows *.exe* files in the following table:

Name	Content
MZ-Stub	MS-DOS header, MS-DOS stub, pointer to the image filer header
Magic PE Value	Signature
Image Filer Header	size of optional header, number of sections
Image Optional Header	Address of entry point, image base, size of image
Data Directories	Pointer to import table, pointer to export table
Section table	List of section header
Sections	*.code* section, *.data* section, etc.

The table contains the structure of a PE image and not the structure of a PE file which is loaded into memory. Each windows executable starts with a MZ-Stub. The stub is a MS-DOS program which displays the message ”You can not run this program in DOS mode”

(or something comparable). Therefore, when executed in an MS-DOS environment, the DOS exe loader recognizes the DOS header, displays the message and terminates.

The header of the MZ-Stub contains an additional (and last) element pointer to the windows PE header which starts with the magic value "'P, E, 0x0, 0x0"' and is followed by the image file header. The image file header has a static size and contains information about the supported machine type (e.g. x86 [4] or ARM [5] architecture), flags (indicating for example whether the PE file is a DLL or not), etc. The important entries for the implementation of the PE crypter are the total amount of sections and the optional header size. They are necessary for parsing the PE header because the total amount of sections and the amount of entries in the data directory are not fixed.

The image file header is followed by the image optional header (which is not optional at all). It contains various informations like the size of executable code, the size of data, etc. See [1] for detailled informations. The important entries in the optional header are the *image base* and the *size of image*. We already mentioned that the container of the PE crypter acts as a decrypter and PE loader. Therefore, the container has to allocate memory at the *image base* address (the usage of the image base is not necessary if the input file provides a relocation table) with a size specified by *size of image* entry. Afterwards, the decrypted file is copied to this location and executed.

The data directory is also a part of the image optional header. It is basically a list which provides the address and size of the relocation table, the export table, the import table, etc. The most important entry for the PE crypter is the pointer to the import table. The import table contains a list of API (*Application Programming Interface*) names. APIs are functions which are located in DLLs und are used by applications to interact with the operating system (e.g. an application which wants to display a message box has to call the API *MessageBox()* located in the *user32.dll*). The import table contains basically the DLL names, the API names and an empty list of function pointers. The container has to parse the import table of the decrypted input file, load the corresponding DLLs, get the addresses of the necessary APIs and write them into the function pointer list.

The next part of the PE header is a list of section headers. Sections provide data and code in a PE file and each section has a corresponding section header. A section header contains a section name, some flags (read, write execute, etc.), the sections address and the section size. The section size consists of the *size of raw data* and the *virtual size*. The *size of raw data* represents the section size in the PE image (e.g. on a hard disk) while the *virtual size* is the section size after being loaded into memory. Also the address entry consists of values: *virtual address* and *pointer to raw data*. Again, the *pointer to raw data* is the section address in the PE image while the *virtual address* is the section address after being loaded into memory. The last section header is followed by the sections.

We have described the structure of PE files and will now introduce the different tasks of the PE loader. Before we can describe the PE loader, we have to discuss the addressing mechanisms in a PE file: Windows 32-bit and 64-bit executables have almost similiar PE headers, The only difference: Depending on the architecture, some address entries

(e.g. the entry point) have a width of 32- or 64-bit. The crypter which is described in this paper supports only 32-bit executables and therefore we assume a 32-bit PE header.

Another import aspect is the absolute and relative addressing: Most entries in a PE header are relative virtual addresses (RVAs). On the other hand, code in a windows executable may use absolute addressing and assumes the PE file is loaded at its image base. If a PE file is loaded to another memory location, the relocation table (which is not mandatory in a regular *.exe* file) has to be used to fix the absolute addressing. We describe now the basic mechanisms of the windos PE loader:

- The amount of memory which is specified in *size of image* is allocated at the *image base* address.
- The complete PE header is copied to the *image base* address.
- The sections are copied to their corresponding virtual addresses.
- The import table is read and the corresponding DLLs are loaded. The addresses of the APIs are written into the previously described function pointer list.
- The section permissions are set (read, write, execute).
- Execution is passed to the PE file and the loader jumps the the files entry point.

This is just a basic and simplified description for a better understanding of the following sections. Some advanced topics are missing and we discuss them in section 4.

3 Hyperion

Hyperion can be divided into two parts: A crypter and a container. The interaction of both components is shown in figure 2 and described in detail in the following two sections.

3.1 The Crypter

The crypter is a command line application and developed in C/C++. It encrypts the input file and injects it into the container. Our first approach was to provide a precompiled container binary. The injection of an input file into a binary container is challenging because the PE header of the container has to be heavily modified.

Furthermore, it is possible that the input file does not contain a relocation table and has to be loaded at its original image base before execution. In this case, the container has to be loaded at the input files image base address and is overwritten after the encryption

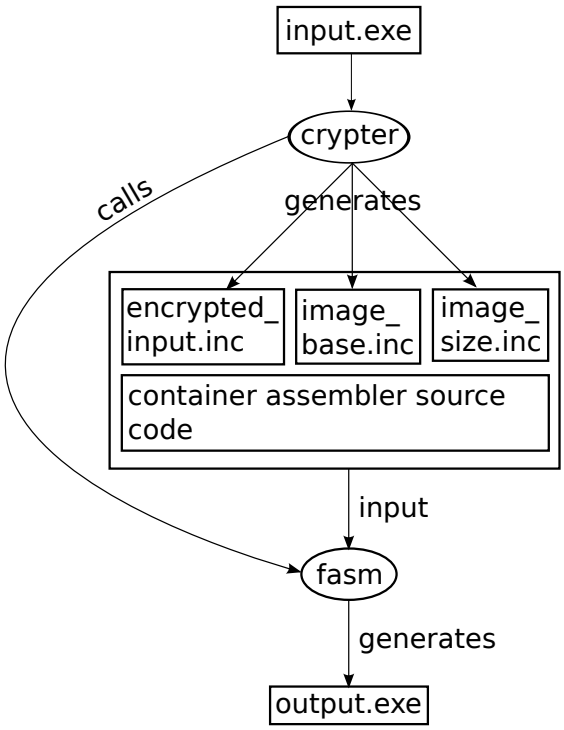

Figure 2: Detailled PE Crypter Workflow

by the input file. This makes it necessary to patch the containers image base PE header entry. The modification of the image base enforces an update of each element in the container (code or data) which relies on absolute addressing.

This paper avoids these problems and presents a new workflow for run-time PE crypter: We inject the encrypted input file into an assembler source code representation of the container. Afterwards, Fasm is called and the container executable is generated. The advantage is: The modification of the PE header, patching the absolute addressing, etc. is not neccessary anymore because this is done by Fasm. Therefore, the crypter relies on the following components:

- The assembler source code of the container.
- A Fasm binary which is called at run-time.
- A DLL which provides 128-bit AES encryption.

We have described the general structure of the crypter and describe now the implementation details: The crypter is called by the user and gets an input and an output file as parameters. The input file is opened, copied into memory and validated:

- The MZ header has to begin with the magic *MZ* value.

- The pointer to the PE header has to be valid pointer.
- The PE header has to begin with the magic *PE* value.
- The input file has to be a 32-bit executable (64-bit executables are not supported yet).

Afterwards, the PE header is analyzed and the *Image Base* and *Size of Image* entries are extracted. Some other values are also parsed (e.g. the section headers) but they are not important for the crypter workflow and just printed on screen as verbose informations for the user.

The next step is the encryption of the input file. A checksum with a size of 4 bytes is created and prepended to the input file buffer in memory. AES is a block cipher and each block has a size of 16 bytes. Therefore, the input file buffer is increased to a size multiple of 16 (the additional space is filled with zeros). After the modification of the input file buffer, a random encryption key is generated. Hyperion uses an AES-128 encryption algorithm which leads to a key size of 16 bytes. The container has to bruteforce the encryption key which would consume a large amount of time if the complete key space is used. Therefore, the key space is reduced and the key is generated using the following algorithm:

Listing 1: AES Key Generation Algorithm

```
1 unsigned char key[AES_KEY_SIZE];
2 for(int i=0;i<AES_KEY_SIZE;i++){
3   if(i<KEY_SIZE) key[i] = rand() % KEY_RANGE;
4   else key[i] = 0;
5 }
```

The listing uses two important constants (which are defined in the crypters source code): *KEY_SIZE* and *KEY_RANGE*. *KEY_RANGE* specifies the the key size and can have a value between 0 and 15 (unused bytes are filled with zeros). The maximum value of each key element is specified in *KEY_SIZE* and can be a value between 0 and 255.

After the generation of the key, the input file is encrypted. We use an AES implementation for Fasm [6] and compile it as a DLL to make it accessible for our C/C++ crypter implementation. The crypter loads the DLL, the API *aesEncrypt()* and encrypts the input file buffer (containing the checksum, the input file and the gap to make its size a multiple of 16) using the generated key. The encrypted file is converted into the following ASCII representation:

Listing 2: Encrypted Input File converted to Fasm Array

```
1 db 0xf3, 0x64, 0x24, 0xa, 0x3e, 0x7e, 0x4c, 0xa6, 0xcd, 0x91, \
2 0x47, 0x2b, 0x5b, 0x3d, 0xd1, 0x2a, 0xa2, 0xff, 0x38, 0x40, \
3 0xe5, 0x5b, 0xa6, 0x8a, 0x44, 0xff, 0xc, 0x47, 0x6a, 0x7f, \
4 ;...
```

The content of the listing is compatible with Fasm, stored in the file *input.asm* and copied into the containers source code directory. Afterwards, the *Image Base*, the *Size of Image*, *KEY_SIZE* and *KEY_RANGE* are also converted into a Fasm representation (the semantic of the corresponding Fasm code is described in section 3.2) and copied into the containers source code directory because they are necessary at run-time (see section 3.2 for details).

The *Image Base* of the input file is converted into the following Fasm representation (assuming the image base address is *0x1000000*), stored in the file *imagebase.asm* and copied into the containers source code folder:

Listing 3: Image Base of Input File converted to Fasm Syntax

```
1 format PE GUI 4.0 at 0x1000000
```

The *Size of Image* is converted into the follwing string (assuming its value is *0x8000*) and stored in the file *sizeofimage.asm*:

Listing 4: Size of Image of Input File converted to Fasm Syntax

```
1 db 0x8000 dup (?)
```

Finally *KEY_SIZE* and *KEY_RANGE* are converted into the following Fasm source code:

Listing 5: Key Space Constants converted to Fasm Syntax

```
1 REAL_KEY_SIZE equ 6
2 REAL_KEY_RANGE equ 4
```

Afterwards, the crypter calls the Fasm binary, compiles the container source code and generates an encrypted version of the input fle.

3.2 The Container

The container basicallt acts as a decrypter and PE loader and is written with Fasm. Listing 6 contains a part of the *main.asm* source code and demonstrates the general structure of the container. *Main.asm* begins with an *include* statement which is comparable to the corresponding C preprocessor directive and includes the content of listing 2. The included statement enforces Fasm to generate a PE file for GUI's which is loaded at the specified image base. Due to this technique, we can ensure that the container is always loaded at the image base of the encrypted input file.

Line 4 contains the *entry* statement which i used fasm to achieve the address of the entry point. It is followed by some included files which are comparable to C/C++ header files for important APIs and libraries. Line 7 includes the file *aes.inc* which is part of the Fasm AES implementation [6]. It is used in the crypter for encryption and used in the container for decryption.

The source code between line 13 and line 20 creates a data section called *.bss*. The content of this section is shown in listing 4. It creates an empty byte array which has a

Listing 6: Main.asm

```
; Hyperion 32-Bit container

include 'imagebase.asm'
entry start

include '..\..\Fasm\fasminclude\win32a.inc'
include '..\..\FasmAES-1.0\aes\aes.inc'
include 'hyperion.inc'
include 'createstrings.inc'
include 'pe.inc'
include 'keysize.inc'

;--------------------------------------------------------

;empty data section with a size equal to image size
;of the encrypted input file
section '.bss' data readable writeable

        decrypted_infile: include 'sizeofimage.asm'

;--------------------------------------------------------

;data section which contains the encrypted exe
section '.data' data readable writeable

        packed_infile: include 'infile.asm'

;--------------------------------------------------------

section '.text' code readable executable

start:  stdcall MainMethod

proc MainMethod stdcall
        ;decrypt input file
        ;load input file
        ;execute input file
        ;...
endp
```

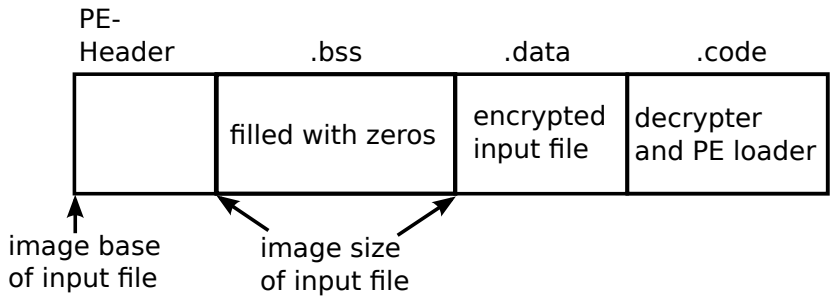

Figure 3: Container in Memory before Decryption

size equal to the image size of the encrypted input fle. Due to this, the *.bss* section has a raw size of 0, a virtual size equal to the input files image size and is located directly after the containers PE header.

The source code between line 21 and 27 creates another data section. It's content is shown in figure 2 and is basically a byte array which contains the encrypted input file. The *.data* section is followed by the *.code* section which decrypts and executes the content of the *.data* section.

We have explained the basic structure of the container and will now illustrate its memory layout. Figure 3 illustrates the memory layout of the container directly after being started and before the decryption process. When the container is executed, it loads dynamically some missing libraries and allocates the adresses of additional APIs. This is necessary because the container invokes APIs like *MapViewOfFile()* for logging support but its import table contains only the APIs *LoadLibrary*, *GetProcAdress* and *ExitProcess*). Afterwards, the container searches for the *.data* section offset using *GetModuleHandle* (which returns its image base) and parsing its PE header. When the *.data* section is found, the following decryption algorithm is applied:

1. Copy the encrypted file into memory as a backup.
2. Guess a key.
3. Decrypt the *.data* section.
4. Verify whether the key was correct using the input files checksum.
5. Wrong key: Restore the *.data* section from the backup and go to 2.

After the encryption, the containers PE header is overwritten with the input files PE header. Furthermore, the sections of the input file are copied into the *.bss* section at their virtual addresses. Finally, the import table of the decrypted input file is processed: Each DLL is loaded (using *LoadLibrary*), the corresponding API offsets are located (using *GetProcAdress*, and written into the adress table. Finally, the crypter passes execution to the input file and jumps to the corresponding entry point. The memory layout of the container after decrypting and loading the input file is shown in figure 4.

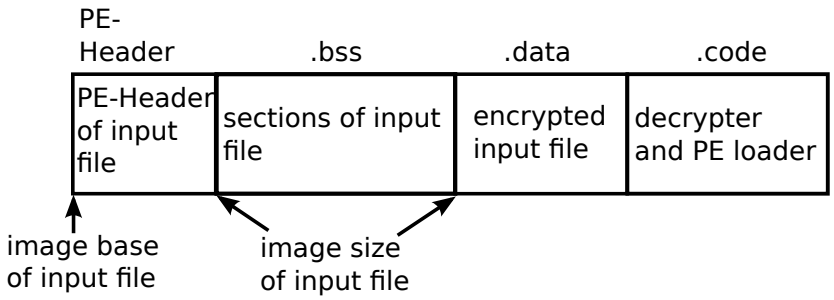

Figure 4: Container in Memory after Decryption

4 Conclusion and Further Work

This paper describes the basic concepts of *Hyperion*, a runtime PE crypter. The complete source code of *Hyperion* will be published on the *Nullsecurity* home page under an open source license.

Some important aspects are still missing in the *Hyperion* implementation: .NET executables are not yet supported and the code of the container has to be refactored to make it fool AV heuristics. Furthermore, concepts like *late API binding* are missing. The most important part which still has to be implemented is polymorphism: The current implementation of *Hyperion* just encrypts the input file. It is a worthwhile goal to encrypt the complete container and generate a small decrypter stub using polymorphism.

5 Acknowledgement

I would like to express my gratitude to the whole *Nullsecurity* team for supporting this work. Special thanks to Kelsey for proof reading and Malfunction for our last coding session and the inspiring conversations.

License

References

[1] Microsoft Cooperation. Microsoft PE and COFF Specification. http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.

[2] Information Technology Laboratory (National Institute of Standards and Technology). *Announcing the Advanced Encryption Standard (AES) [electronic resource]*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD :, 2001.

[3] Tomasz Grysztar. Flat Assembler. http://flatassembler.net/.

[4] R.C. Detmer. *Introduction to 80x86 Assembly Language and Computer Architecture*. Jones and Bartlett, 2001.

[5] David Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[6] Christian Ammann. AES Implementation for Flat Assembler. http://www.nullsecurity.net/tools/cryptography/fasmaes-1.0.tar.gz.