Ideas on advanced runtime Encryption of .NET Executables

Christian Ammann

December 13, 2012

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: As 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.

We presented in our last paper [1] the theoretic aspects of runtime PE [2] encryption and a reference implementation called *Hyperion*. Hyperion generates a random key and uses it to encrypt the input file with AES-128 [3]. The encrypted file decrypts itself upon startup bruteforcing the necessary key. Therefore no AES key is stored inside the file which makes it hard for an anti virus solution to detect the encrypted payload. However, one major aspect is still missing: Hyperion encrypts only regular portable executables and support for .NET [4] byte code (which is used by Microsoft in C# [5], J# [6] and visual basic [7] applications) has to be implemented.

Therefore, this paper reveals the aspects of .NET runtime encryption and presents a proof of concept implementation for Hyperion. It has the following structure: Section 2 describes the basic layout of .NET executables and their integration in native PE files. Afterwards, we discuss in the section 3 the possible implementation approaches of .NET runtime encryption. In the section 3.1, we add .NET support to Hyperion and discover a problem through reverse engineering of a windows DLL (Dynamic Link Library): Even if a decrypted .NET file has been already been loaded by Hyperion into memory, the .NET runtime execution environment reads some header information from the corresponding disk image. The disk image is encrypted which causes the runtime environment to abort the execution. In the section 3.2, we solve this problem with a partial encryption of .NET files and present a proof of concept implementation. Some advanced .NET runtime encryption techniques are described in the section 4 and left as further work.

2 .NET Format

This sections describes the layout of .NET applications and how they are loaded/executed in memory. The .NET framework is a platform created my Microsoft which allows the development and execution of software applications. In contrast to languages like C, a .NET compiler does not generate native code. Instead, the source code is transformed into a byte code format which is called *Common Intermediate Language* (CIL). Additionaly, the *Common Language Runtime* (CLR) is part of the .NET framework which acts as a runtime environment. When a CIL file is executed, it is passed to the CLR framework which transforms the byte code into native instructions and executes the corresponding application. It is also possible to transform .NET source code into native code using the *Native Image Generator* (NGEN).

.NET is developed by Microsoft for the Windows operating system. Therefore, .NET code is embedded into regular PE files which have the following layout:

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	Adress 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.

Like native PE files, .NET applications contain a MZ-Stub, image file header, image optional header, a data directory and the corresponding sections. The concrete semantics of these entries is not part of this paper and explained in [1] or [2] in more detail. When opening a .NET application with a PE editor like Lord PE [8] we see the the following structure:

- A .text section which contains the import table, the import address table and the CIL code.
- A .rsrc section which contains the file icon.
- A .reloc section which contains the relocation table. It has just one entry for fixing the files entry point instruction.

Upon startup, a .NET application is loaded like a regular executable into memory. When this is done and the sections have been copied to their virtual addresses, the PE loader has to determine whether a file contains native or CIL code. If a file contains native code, the loader jumps to the file entry point and the application is executed by the CPU. Otherwise, execution is passed to the CLR environment. Therefore, the data directory of each PE file contains an entry called *CLR Runtime Header* which can be used to distinguish between .NET and native executables. If a file contains CIL byte code, it points to the the *CIL header*. Otherwise, the pointer of the CLR Runtime Header is set to 0.

The PE specification describes the content of the CLR Runtime Header the following way: The format of the metadata is not documented, but can be handed to the CLR interfaces for handling metadata. Luckily, this statement is not correct anymore because Microsoft released the CIL and CLR specifications [9]. Additionally, Microsoft provides the *Shared Source Common Language Infrastructure* (SSCLI) [10] which is a working CIL implementation in C++ and can be used by developers to study the internals of the .NET framework. The license is non-free and prohibits commercial usage.

One excellent resource in the SSCLI is the class *PEReader* which can be found in the following subdirectory: *samples/utilities/getcliversion/pereader.cs*. It parses the complete PE header of an input file including the CIL header. Furthermore, the syntax and semantics of the .NET format are also described by Daniel Pistelli [11] in a very good article. According to these resources, the .NET code in a PE file, which can be acquired from the corresponding data directory entry, has the following layout:

1. CIL header
2. CIL code and resources

3. MetaData header
4. Streams
5. MetaData tables

This layout is not 100% correct because the CIL code and the resources do not have to be placed between the CIL header and the MetaData header. In fact, this layout is the result of empirical measurements and the analysis of .NET executables. Furthermore, this is only a brief introduction to the different .NET components. For more details, we recommend the corresponding literature.

The CIL header contains a basic description of the .NET file. Two entries represent the minor and major runtime environment version numbers which are necessary to execute the CIL code. The CIL header also contains a pointer to the .NET resource section and a pointer to the MetaData header.

The MetaData header provides a magic value and also the minor/major runtime version numbers. According to [11], both values are ignored by the loader. The important elements in the MetaData header are the stream headers. Each stream header consists of name, size and an offset. Default streams in each .NET file are e.g. *Strings* or *Blob*. *Strings* contains ascii strings while *Blob* provides binary data.

The elements of these streams are referenced by the MetaData tables which are part of a stream called #~. Therefore, the .NET layout we presented above needs to be corrected: The MetaData tables are part of the streams section. Nevertheless, we wanted to emphasize that its semantics differ from the other streams and put it in an extra section. The MetaData tables contain the following elements:

- Names of classes, their sub classes, etc.
- Variable names, types and initialisation values.
- Name of constants and their corresponding names.
- Method names, their parameters, method body, return values, etc.
- ...

The following example demonstrates the relationship of the CIL code, the MetaData tables and the streams: A .NET application uses the constant a with a value of 1. Furthermore, it provides the class B which implements the method $c()$. The identifiers a, B and c are stored in the *Strings* stream while the constant 1 can be found in *Blob* stream. The necessary MetaData tables will contain entries for a class, a constant and a method. Each entry has pointers to the corresponding stream elements. Additionally, the MetaData table entry of c has a reference to the CIL code which is executed upon a function call. Again this is just a brief description, we recommand to dig in the references for more details.

We know now the structure of .NET applications which are embedded into PE files. One aspect we have not yet described is the execution of CIL code after being loaded into memory. For this task, the loader checks whether the files data directory contains a pointer to the CIL header. When a valid entry is found, the *MsCorEE.dll* is loaded into memory and the *_CorValidateImage()* function is called. This API does some modifcations in the PE header and overwrites the program entry point. The new entry point is the address of the *_CorExeMain()* which passes execution to the CLR environment.

3 Runtime Encryption of .NET Files

We described in the last section the components of .NET, the basic file layout and its execution. This chapter discusses two runtime encryption approaches for .NET files. Afterwards, we implement one approach in Hyperion and run into a problem regarding the .NET loading mechanisms. The problem is solved and a proof of concept implementation is shown which can be used in further work for a more sophisticated runtime encryption of .NET files.

The windows APIs for native applications provide the function *CreateProcess()* which starts a process. One parameter is a string which contains the file name of the disk image. From a runtime crypters point of view, this is a disadvantage because an encrypted file has to be decrypted and dropped on the HD before *CreateProcess()* can be called. Therefore, common runtime encryption tools like Hyperion implement their own PE loader which operates in memory and not on disk images.

CIL code does not suffer from this restriction because it can use the reflection framework. The reflection framework provides the overloaded *Assembly.Load()* function. It accepts, for example, a string, which contains the name of a .NET applications disk image and loads it into memory. Afterwards it's methods can be called. Furthermore, it is also possible to pass a byte array to *Assembly.Load()* which contains the image of a .NET file. Therefore. in contrast to native code and *CreateProcess()*, .CIL code can start another .NET program directly from memory without re-implementing the PE loader.

With the reflection API, a runtime encryption tool for .NET applications can be implemented in the following way: The runtime encryption tools opens an input file, start encryption and deploy the decrypted result in the dropper executable (e.g. as a resource). Upon startup, the dropper decrypts the payload in memory and executes it using *Assembly.Load()*. This approach is clear and simple but leads to the following disadvantages:

- *Assembly.Load()* is restricted for CIL code.
- It is easy for an AV product to detect *Assembly.Load()* calls and therefore triggers the heuristics.

To avoid these problems, we use another approach in this work. According to the section 2, .NET files are loaded like native PE files into memory and started with *_CorExeMain()*. Therefore Hyperion can be modified the following way: When a file has been loaded into memory, its data directory entries can be verified. When a pointer to the CIL header is found, *_CorValidateImage()* is called. Finally, the loader jumps into *_CorExeMain()* and begins the execution of the CIL code. *_CorExeMain()* and *_CorValidateImage()* have to be dynamically loaded using *GetProcAddress()* and other obfuscation techniques which allow an effective protection against AV detection.

3.1 Hyperion 1.0 and .NET

We have discussed two runtime encryption techniques in the last section. Based on this evaluation, we have chosen to implement the second approach in Hyperion. The workflow of the current Hyperion implementation when unpacking an encrypted file is simple: First, it decrypts the payload using a bruteforce attack to acquire the encryption key. Afterwards, the encrypted file is loaded into memory according to its PE- and section headers. Finally, a jump to the entry

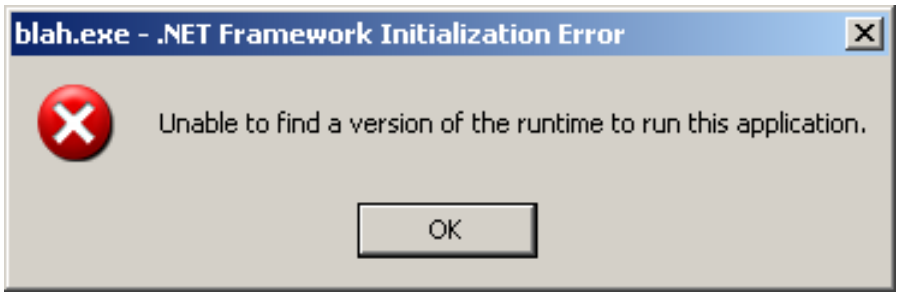

Figure 1: Error Message of Hyperion after unpacking a .NET File

point is performed and execution is passed to the unpacked file. The next listing shows how Hyperion jumps to the entry point of the previously decrypted payload:

```
01. ;...
02. mov edx,[image_base]
03. mov eax,[edx+IMAGE_DOS_HEADER.e_lfanew]
04. add eax,edx
05. add eax,4
06. ;image file header now in eax
07. add eax,sizeof.IMAGE_FILE_HEADER
08. mov eax,[eax+IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint]
09. add eax,[image_base]
10. ;entry point of original exe is now in eax
11. jmp eax
```

In the listing above, the base image of the previously decrypted and loaded file is stored in edx. Afterwards, the location of the PE header is calculated and copied to eax. The PE header is used to acquire the image optional header which contains the correponding entry point. Finally, a jump to the the entry point is executed. An extension for .NET files is simple: A function has to be inserted in line 1 which gets the base image of the decrypted file as a parameter and parses the PE headers until the data directory is reached. If a pointer to the CIL header is present, the API to *_CorValidateImage()* is called and the function returns. Hyperion will now jump into *_CorExeMain()* at it's end because *_CorValidateImage()* modified the files entry point.

We added this modification to Hyperion which allows the runtime encryption of .NET files. The implementation was tested with a simple hello world application called *blah.exe*. The result can be seen in figure 1 which indicates that the runtime environment can not read the file's runtime version. The minor/major runtime version is stored in the CIL header (see section 2 for details).

First possible solution: There is a bug in Hyperion and either a part, or the complete file is not loaded/decrypted correctly in memory. Therefore, we analyzed the image of the decrypted file and discovered: There is no error in Hyperion, the file is loaded and decrypted successfully. This leads to the following situation:

- As long it's not encrypted using Hyperion, the hello world .NET application runs normally. Therefore, the correct .NET runtime version must be available on our system.
- When the file is encrypted with Hyperion, the .NET runtime environment can not read the CIL header, although it is correctly loaded into memory.

Therefore, we debugged the encrypted file with Immunity [12] debugger, set a breakpoint at *_CorExeMain()* (which is the entry point of the CLR) and stepped through the file. The result

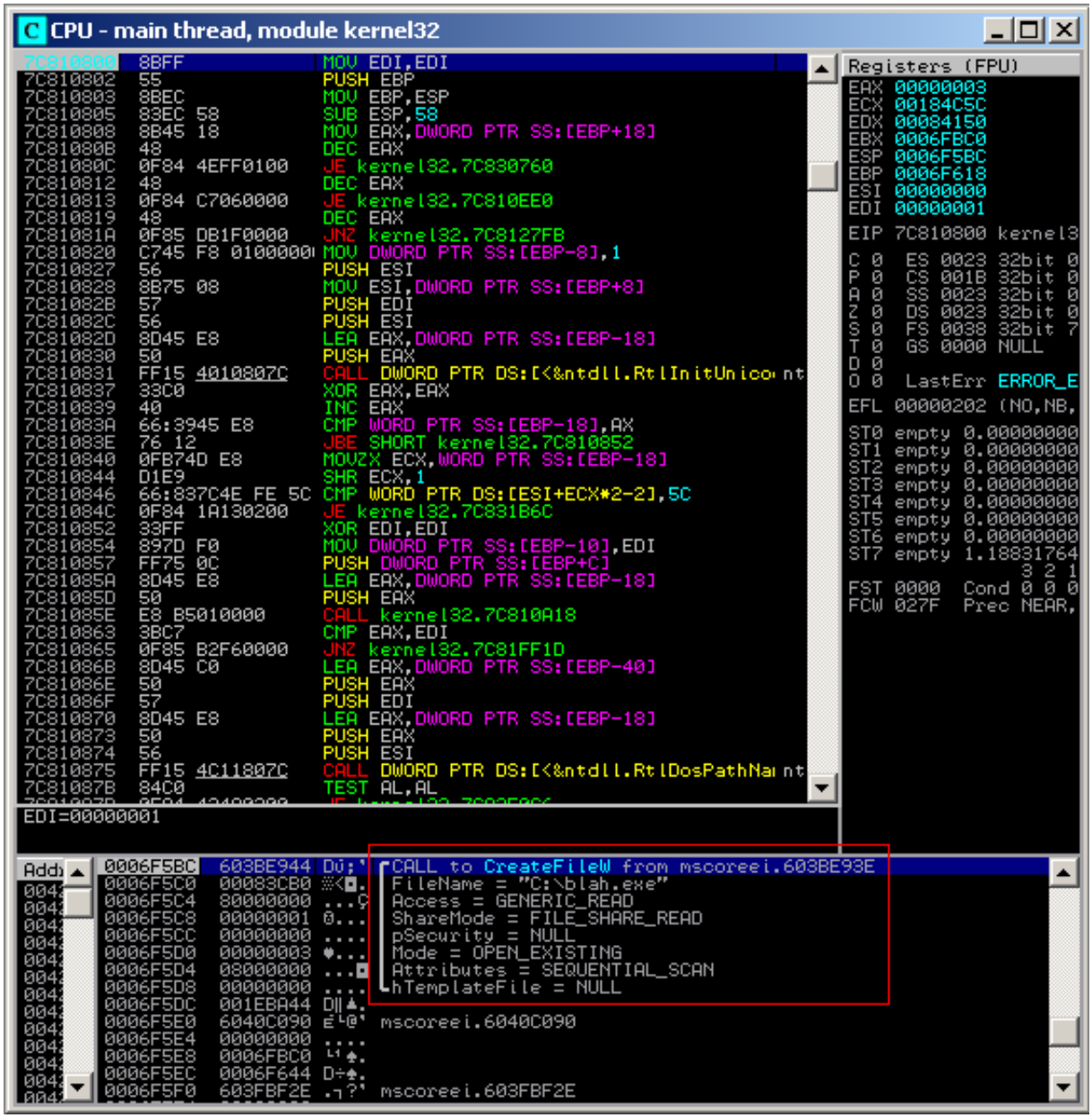

Figure 2: Debugging the .NET Framework

is shown in figure 2 (marked with a red square): The CIL framework opens the *blah.exe* disk image using *CreateFileW()*. This is strange and unnecessary as the complete file was already copied into memory by the PE loader. We continue our investigations, stepping through the code using Immunity and discover the following behaviour:

- *CreateFileW*, *CreateFileMapping* and *MapViewOfFile()* are called to map the disk image into memory.
- Afterwards, the PE header of the file's disk image is parsed and evaluated.

This a problem because the disk image is encrypted. Therefore, the CLR can not read the minor/major runtime version in the CLI header and the Hyperion approach fails for .NET executables.

3.2 Partial Encryption of .NET Files

We have seen in the previous section that the CIL framework accesses a file's disk image even though it was previously copied into memory by the PE loader. Therefore, .NET executables which were encrypted with Hyperion fail to load because the CLR can not read the CIL header. We present in this chapter a proof of concept implementation for .NET runtime encryption which keeps the CLI header unencrypted. Our approach consists of two applications:

- Encrypter: Opens a .NET file, and encrypts its CIL code leaving PE header, CIL header and MetaData untouched.
- Loader: Starts the encrypted file with CreateProcess(). The main thread is in a suspended state. The loader decrypts the CIL code and resumes the main thread.

Of course, this implementation is vulnerable to statical analysis by an AV product because the CLI header and the MetaData are not encrypted. Therefore, we derive from this application in section 4 an extension for Hyperion which performs a full encryption of .NET files.

3.2.1 Encrypter

The encrypter gets an input file and parses it's header to allocate and encrypt the CIL code. Therefore, the input file is opened and copied into memory. Afterwards, the file headers are analyzed to find the CIL code.

Each PE file begins with a MZ header which contains a pointer to the coff header. The coff header has a fixed size and is followed by the optional standard and optional windows headers (also with a fixed size). The data directory is located after the optional windows header and it's size is defined by *NumberOfRvaAndSizes* which is an element in the optional windows header. The size of the data directory can be zero. Therefore, it is a good practice to calculate the optional header size using *SizeOfOptionalHeader* and not to rely on the *NumberOfRvaAndSizes* entry. The data directory is an array of *ImageDataDirectory* structures:

```
01. struct ImageDataDirectory {
02.     uint32_t VirtualAddress;
03.     uint32_t Size;
04. };
```

Each entry consists of a relative virtual address (rva) and it's relevant size. If the rva is set zero, the corresponding entry is not used in the PE file. Therefore, the following code can be used to check whether a file contains CIL code:

```
01. ImageCor20Header* rva_cil_header = (ImageCor20Header*)
02.   data_directory [CLR_RUNTIME_HEADER]).VirtualAddress;
03.
```

```
04. if(rva_cil_header==0){
05.    printf("no clr runtime header found");
06.    return false;
07. }
```

The *ImageCor20Header* structure represents the CIL header and has the following layout:

```
01. struct ImageCor20Header
02. {
03.   uint32_t                  cb;
04.   uint16_t                  MajorRuntimeVersion;
05.   uint16_t                  MinorRuntimeVersion;
06.   struct ImageDataDirectory  MetaData;
07.   uint32_t                  Flags;
08.   uint32_t                  EntryPoint;
09.   struct ImageDataDirectory  Resources;
10.   struct ImageDataDirectory  StrongNameSignature;
11.   struct ImageDataDirectory  CodeManagerTable;
12.   struct ImageDataDirectory  VTableFixups;
13.   struct ImageDataDirectory  ExportAddressTableJumps;
14.   struct ImageDataDirectory  ManagedNativeHeader;
15. };
```

For this paper, the important element is the pointer to the MetaData which is implemented with an *ImageDataDirectory* structure. Therefore, when the CIL header is reached, the encrypter has gathered the following information:

- Size and rva of the CIL header
- Size and rva of the MetaData header.

According to the .NET layout in section 2, the CIL code is placed in between the CIL and the MetaData headers. Accordingly, the encrypter calculates the offset of the CIL code and performs a simple XOR encryption.

One important aspect is still missing: The input file is mapped into memory at a certain image base. The files PE headers contain pointers which are relative virtual addresses. An rva has to be transformed into the corresponding raw address before it's data can be read or written. It is therefore good practise to write a small converter function:

```
01. unsigned char* rvaToOffset(unsigned char* base,
02.   struct SectionHeader* sections,
03.   uint16_t sections_size, uint32_t rva)
04. {
05.   unsigned char* ret = 0;
06.   for(int i=0;i<sections_size;i++){
07.     if(rva >= sections[i].VirtualAddress &&
08.     rva < sections[i].VirtualAddress + sections[i].VirtualSize){
09.           ret = base + sections[i].PointerToRawData +
10.       (rva - sections[i].VirtualAddress);
```

```
11.     }
12.   }
13.   return ret;
14. }
```

The method *rvaToOffset()* receives the following parameters: The input files address in memory, a pointer to the section headers, the total amount of section headers and the rva which has to be transformed into a raw address. *RvaToOffset()* searches the section, in which the rva parameter value is located. It adds the sections raw address to the files address in memory and returns the corresponding value.

3.2.2 Loader

We have seen in the previous section, how the CIL code of a .NET file can be encrypted. The loader starts the encrypted file using *CreateProcess()*. The file's main thread is suspended. Afterwards, it decrypts the CIL code in memory and resumes the main thread:

```
01. STARTUPINFOA startupinfo;
02. memset(&startupinfo, 0, sizeof(startupinfo));
03. startupinfo.cb = sizeof(STARTUPINFOA);
04. PROCESS_INFORMATION process_information;
05. memset(&process_information, 0, sizeof(process_information));
06. CreateProcessA(application_name_and_path, 0, 0, 0,
07.  false, NORMAL_PRIORITY_CLASS | CREATE_SUSPENDED, 0, 0, &startupinfo,
08.  &process_information);
```

CreateProcess() relies on two structures: *STARTUPINFO* and *PROCESS_INFORMATION*. In the listing above, both are initialized with zero bytes. Afterwards, *CreateProcess()* is called and starts the target file. The file's process information is stored in the *process_information* structure. Finally, the CIL code is decrypted using the following algorithm:

- *VirtualProtectEx()* is called for making the process memory writable.
- The process is copied into a buffer using *ReadProcessMemory()* and the (previously populated) *process_information* structure.
- The CIL code is allocated and decrypted.
- The buffer is copied back to the process image base using *WriteProcessMemory()*.
- The original memory page attributes are restored using *VirtualProtectEx()*.

Finally, the main thread is resumed using *ResumeThread(process_information.hThread);*.

4 Further Work

This paper describes the .NET file layout. Afterwards, it presents an algorithm which can be implemented in Hyperion to perform a runtime encryption of .NET files. The approach fails because the .NET runtime environment relies on the disk image which is still encrypted and therefore can not be read. We prove this claim with a partial runtime encryption method which leaves the CIL header untouched.

Although the proof of concept implemetation can be used for runtime encryption of .NET files, it suffers from the following disadvantages:

- It uses a simple XOR encryption algorithm while Hyperion is capable of AES-128.
- It encrypts only the CIL code while the header and MetaData are still vulnerable to statical analysis by an AV product.

In further work, full .NET runtime encryption will be added to Hyperion in the following manner: Hyperion has to hook *CreateFileW*, *CreateFileMapping* and *MapViewOfFile()* in *kernel32.dll*. The hooks check whether the CIL runtime environment tries to open the files image on disk. In this case, a pointer to a previously decrypted file image is returned. This allows the CLR to access the CIL header. This disables the problem described in figure 1 and allows a full encryption of .NET files.

5 Acknowledgement

We would like to express our gratitude to Chris Cowling, Ian Qvist and Thomas Pedersen for supporting this work. Furthermore, we would like to thank the whole Nullsecurity team for the fruitfull discussions and (most important;) a great party on Berlin-Sides 2012.

License

References

[1] Christian Ammann. Hyperion: Implementation of a PE-Crypter. http://nullsecurity.net/papers/nullsec-pe-crypter.pdf.

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

[3] 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.

[4] Microsoft Cooperation. The .NET Framework. http://www.microsoft.com/net.

[5] Microsoft Cooperation. C# Programming Guide. http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx.

[6] Microsoft Cooperation. J-Sharp. http://msdn.microsoft.com/en-us/vstudio/bb188593.

[7] Microsoft Cooperation. Visual Basic. http://msdn.microsoft.com/en-us/vstudio/hh388568.aspx.

[8] y0da. Lord PE. http://www.woodmann.com/collaborative/tools/index.php/LordPE.

[9] Microsoft Cooperation. Standard ECMA-335 - Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[10] Microsoft Cooperation. Shared Source Common Language Infrastructure. http://www.microsoft.com/en-us/download/details.aspx?id=4917.

[11] Daniel Pistelli. The .NET File Format. http://www.codeproject.com/Articles/12585/The-NET-File-Format.

[12] Immunity Inc. Immunity Debugger. https://www.immunityinc.com/products-immdbg.shtml.