University of Piraeus

Department of Digital Systems

MSc. Techno-economic Management & Security of Digital Systems

Master's Thesis

Enhancing AV-avoidance capabilities of a PE crypter

Christos Papadiotis

MTE 1062

Supervisor: Prof. Dr. C. Lambrinoudakis

Piraeus - March 2014

Abstract

Penetration Testers use a wide range of publicly available or custom made tools in their attempt to bypass security controls of the targeted systems during security assessments. A lot of these tools are often flagged by anti-virus products as suspicious or downright malicious. In order to avoid detection a number of solutions have been introduced, with the most popular one involving the use of crypters. A crypter is a piece of software that encrypts an executable object and encapsulates it into seemingly innocuous code, effectively modifying its appearance in a binary level, while at the same time preserves its original functionality. The purpose of this thesis is, based on a reference implementation of a PE crypter, to improve its Anti-Virus avoidance capabilities by using well established obfuscation techniques.

Περίληψη

Κατά τη διενέργεια αξιολογήσεων ασφαλείας οι Penetration Testers στην προσπάθειά τους να παρακάμψουν τα τυχόν μέτρα ασφάλειας, χρησιμοποιούν ένα μεγάλο εύρος εργαλείων, πολλά εκ των οποίων επισημαίνονται από τα αντιβιοτικά προγράμματα ως ύποπτα ή και εμφανώς κακόβουλα. Ανάμεσα στις λύσεις που έχουν προταθεί προκειμένου να αποφεύγεται ο εντοπισμός αυτός, περιλαμβάνονται και οι κρυπτογράφοι (crypters). Με τον όρο κρυπτογράφος, εννοούμε το λογισμικό που, όπως φανερώνει το όνομά του, αναλαμβάνει να κρυπτογραφήσει το εκτελέσιμο-στόχο και να το περιβάλει με φαινομενικά αθώο κώδικα με σκοπό να διατηρήσει ατόφια την αρχική του λειτουργία, αλλάζοντας παράλληλα την εμφάνισή του σε δυαδικό επίπεδο. Ο σκοπός της εργασίας αυτής είναι ,βασιζόμενοι σε μια υλοποίηση αναφοράς ενός κρυπτογράφου για αρχεία τύπου PE (Portable Executable), να βελτιώσουμε τις ικανότητες του αποφυγής αντιβιοτικών, εφαρμόζοντας καθιερωμένες τεχνικές απόκρυψης.

Contents

List of Tables

List of Figures

Acknowledgements

Firstly, I would like to thank my parents (and first teachers) for allowing me to realize my own potential. All the love and support they have provided me over the years were the greatest gifts.

I would like to thank my supervisor, Prof. Dr. Costas Lambrinoudakis for allowing me to pursue a line of research that I am passionate about. I am grateful for his advice, patience and understanding.

I would also like to thank my colleagues John Kolovos, Fanis Drosos and Lefteris Panos for offering useful input and perspective and for tolerating my endless thinking-out-loud rants during the development phase of this thesis.

Last but not least, I'd like to thank my brother Gavriil for being a persistent voice of motivation ("Get over with it already!") and an invaluable proofreader.

1 Introduction

Anti-Virus products being often the first line of defense in a corporate IT environment are an invaluable tool in the arsenal of System and Network Administrations in their constant struggle to keep their perimeter secure.

During security assessments, Penetration Testers adopt the role of an unauthenticated attacker flying under the radar, thus are often faced with the need to deploy tools that are usually flagged by Anti-Viruses as suspicious or malicious. One method to avoid detection of those tools entails using crypters.

A crypter is a piece of software that encrypts an executable (case at hand the suspicious tool) and wraps it with code that appears legit. This way a new executable is produced that has the same functionality as the original file, but is completely different in a binary level.

The starting point of this thesis was "Hyperion" a POC PE crypter developed by Christian Amman, so after a brief reference in the ways that Antivirus products operate and a concise overview of the PE file, we will present the basic concepts behind the original implementation and point out its weaknesses.

We then proceed into analyzing our improvements over the original implementation and presenting the results of the test we conducted, comparing the AV avoidance capabilities of the original with those of our implementation.

Due to the controversial nature of the implementation and in order to prevent its use for malicious reasons, no source code has been included in this document. For requests regarding access to the source code for research purposes, please contact the author.

2 Related Work

This sections contains references to related implementations of PE packers that worked as inspiration for our work.

2.1 Hyperion

Hyperion (Ammann, Hyperion: Implementation of a PE-Crypter, 2012) is PE crypter developed by Christian Amman in 2012. Since our work was based on its original implementation a brief analysis of its inner workings can be found in section 6 of this paper.

2.2 Packing Heat!

A presentation (Glinos, 2012) regarding a packer that generates metamorphic executables. Each executable generated by this type of packer both looks different on-disk and behaves differently at runtime. No source code has been released, but the author describes with great detail the methods he followed. The packer was developed in Ruby, using METASM, a Ruby assembly manipulation suite (Guillot).

2.3 PEScrambler

PEScrambler (Harbour, 2008) is a tool to automatically obfuscate win32 binaries. It can relocate portions of code and protect them with anti-disassembly code. It also defeats static program flow analysis by re-routing all function calls through a central dispatcher function.

3 Anti-Virus Modi Operandi

Despite any differences in the actual implementation of malware-detection mechanisms, all antivirus products tend to incorporate the same virus detection techniques (Zeltser, 2001). The following sections briefly describe these methods.

3.1 Signature-based detection

Signature-based detection is the oldest method used for antivirus detection. Modern antivirus products still use it as a first line of defense since it allows a quick detection of known threats. It could involve byte signatures and/or hash signatures.

A byte-signature is basically a specific sequence of bytes that can be found in a file. Usually, this sequence is in hexadecimal form and is kept large enough to avoid false-positive and yet concise enough to avoid memory waste (Kumar, Sharma, & Kumar, 2011). It is chosen because it exist in multiple variants of malware from the same family and it can be any type of data such as code or data contained inside of a data stream of an executable, a XORed .pdf or a Word document. (Zeltser, 2001).

For the creation of hashing signatures a plethora of methods have been proposed and implemented, ranging from simple cryptographic hashing algorithms (MD5, SHA1) that are being applied to known malware, to more advanced such a as context triggered piecewise hashing that can be used to identify modified versions of known malicious

files even if data has been inserted, modified, or deleted in the new files (Kornblum, 2006).

3.2 Heuristics-based detection

Heuristic based detection uses a rule-based approach to diagnose a potentially-malicious file. Antiviruses that perform heuristics-based detection include an analyzer engine that works through its rule-base, while checking a file against criteria that indicate possible malware, assigning score points when it locates a match. If the score meets or exceeds a threshold score, the file is flagged as suspicious (or potentially malicious) (Harley & Lee, 2008) and is handled accordingly.

3.3 Behavioral detection

Antiviruses using behavioral detection methods are based on integrating with the operating system of a host computer and monitoring programs in real-time. They basically attempt to identify malware by looking for suspicious actions. As with heuristics, each of these actions per se might not be enough to classify a program as suspicious, but certain sequences of actions could be indicative of malicious behavior.

4 PE file

Since the implementation of a packer requires a deep understanding of the PE file format and the way that the Windows loader works, in this section we include a short overview of both.

A Portable Executable (PE) file is the standard binary file format for an Executable or DLL under Windows NT, Windows 95, and Win32. The term "Portable Executable" was chosen because the intent was to have a common file format for all flavors of Windows, on all supported CPUs (Pietrek, 2002).

The PE file format is organized as a linear stream of data. It begins with an MS-DOS header, a real-mode program stub, and a PE file signature. Immediately following is a PE file header and optional header. Beyond that, all the section headers appear, followed by all of the section bodies. Closing out the file are a few other regions of miscellaneous information, including relocation information, symbol table information, line number information, and string table data.

When the PE file is run the PE loader examines the DOS MZ header for the offset of the PE header. If found it skips to the PE header. The PE loader checks if the PE header is valid. If so, it goes to the end of the PE header. Immediately following the PE header is the section table, which contains information about each section in the image. The PE header reads the information about the sections and maps those sections into

memory using file mapping. It also gives each section the attributes as specified in the section table (Iczelion, 2002).After the PE file is mapped into memory the PE loader will continue to work on the logic of the file, such as importing code or data from DLLs (Locating all the imported functions and data and making those addresses available to the file being loaded). It then sets the necessary access permissions for each section and passes the execution to the PE file being loaded by jumping to its entry point.

5 Hyperion

This chapter contains a description of the Hyperion functionality as well as the limitations of its design.

5.1 Description

Hyperion (Ammann, 2012) consists of two, parts a crypter and a container.

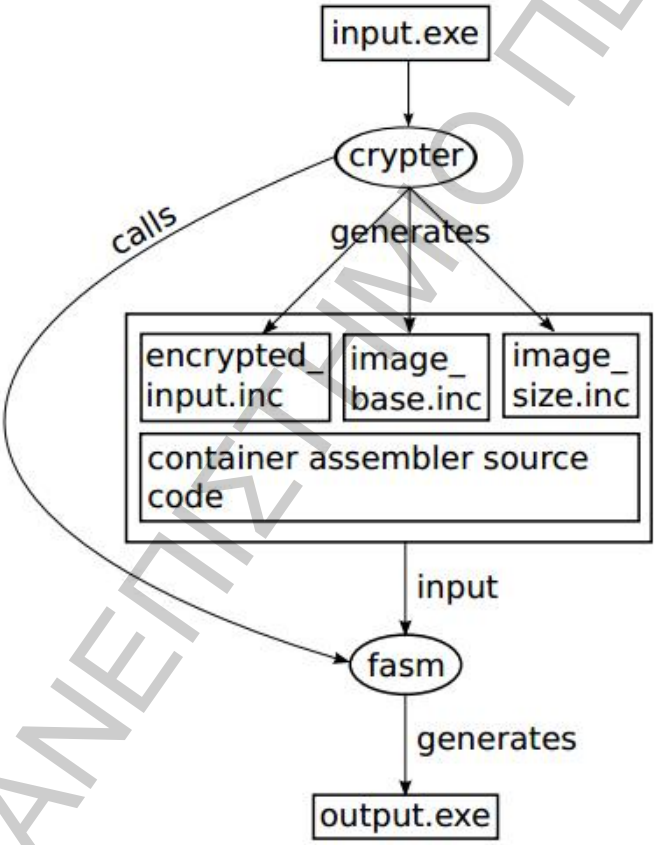

Figure 1: Hyperion Crypter Workflow

The crypter receives a valid executable file as input, it calculates a checksum for it and encrypts both the checksum and the file with AES (with a purposely reduced key range). It then proceeds into creating a new executable file by utilizing FASM (Grysztar, n.d.), an assembler that supports programming in Intel-style assembly language on the IA-32 and x86-64 computer architectures. An assembler source code representation of the container is being used and is being fed into FASM along with the encrypted blob of the input executable as well as other critical pieces of information (image base and size of image of the input executable).

The output executable has the same image base as the input one, it includes the encrypted binary blob in its .data section and contains a .bss section with a virtual size that matches that of the input exe.

At runtime, the container acts as a decrypter and a loader. Since the key for the encrypted input executable is not included in the output executable, the container basically brute forces the key, it calculates a checksum for every decryption output and compares it to the one included. Once decrypted the container's PE header is overwritten with the input file PE header. The section table of the input file is being parsed, and its sections are also copied into the .bss section of the output at their virtual addresses. As it was mentioned earlier, the .bss section has the size of the input's image size and is being positioned in memory before the output's code section, thus ensuring that no portions of the output's code will be overwritten. Furthermore, the import table

of the decrypted input file is processed and the necessary DLL are being loaded and its address table is being populated. Finally, the execution is being passed to the input file with a simple jump to its corresponding entry point.

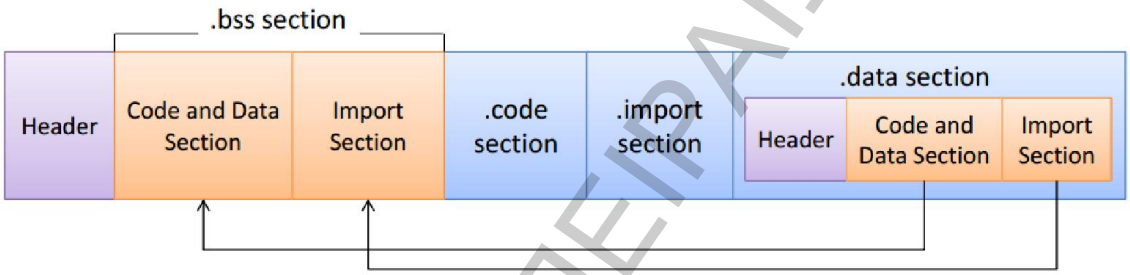

Figure 2: Hyperion decryption procedure

5.2 Limitations

Hyperion is a brilliant implementation that, at the time of its issue, made Anti-Virus evasion a rather trivial task. However due to its Proof-Of-Concept nature it suffers from a number of issues, some of which are outlined by its author himself at the "further work" section of his releasing paper (Ammann, Hyperion: Implementation of a PE-Crypter, 2012).

First of all, even if the input executable is being encrypted, the part of the container's code responsible for decryption and loading is static. It is just included as an assembler source code representation by the crypter, it never changes, so it's really easy for the antiviruses to create signatures for it and flag it as malicious every time it appears.

The sparse import table of the output executable, which only contains ExitProcess, GetProcAddress and LoadLibraryA functions from kernel32.dll, is another factor that potentially raises red flags for the Anti-Viruses since it is a rather strong indication that more API functions will be resolved at run time, a behaviour that is in itself suspicious.

The decrypter's assembly code also contains a number functions for log creating, a feature that although useful during development and debugging, adds unnecessary overhead and potentially facilitates the AV signature development.

The key space size can only be changed inside the crypter source code to speed up or slow down at will the brute force key search algorithm. This can be an issue in cases where extensive fine tuning testing is required.

It also must be mentioned that the original implementation is Windows oriented and extra steps have to be taken when cross-compilation is desired.

6 Our contribution

Based upon the original implementation of Hyperion we performed a number of modifications in an attempt to enhance its AV-avoidance capabilities. This following sections describe the steps we took in order to accomplish this.

6.1 Porting Hyperion to Python

The part of the crypter that was originally developed in C, was rewritten in Python. We used well established python libraries such as PyCrypto for the AES encryption and argparse for a basic command-line interface (one that allows users to define key space size and range). We also used miasm & elfesteem in order to analyse input executable and create the output executable.

Miasm (Desclaux, 2012) is an open source reverse engineering framework. It is, in a way, the Python equivalent of the Ruby-written METASM framework and it allows the analysis, modification and (re)generation of binary programs (PE/ELF/CLASS). We used its Elfesteem module to parse & construct valid PEs. Complex information extracting procedures, such as retrieving the image base or the image size value of an executable file, which would normally require a deep knowledge of the PE structure, can be performed in only a couple lines of code.

For the creation of the output executable, we imported the modified (see following sections) ASM code of the container as a list of Python strings. These strings,

once concatenated, are parsed by MIASM and turned it into opcodes. Elfesteem is called again to construct the final output file, using opcodes from the previous step to create the .text section. The encrypted binary blob is being placed in the .data section of the output file.

6.2 Clearing-up ASM code & removing HLA elements

The ASM code in the original implementation of Hyperion contains functions for logging purposes. During the execution of the decryptor a log.txt file is created in the same folder where the decryptor resides. Apart from potentially creating extra opportunities for the Anti-Viruses to create signatures for the binary, it increases the forensics footprint left on the targeted system. All log-related code has been therefore removed from our implementation.

```
;create logfile and write init message into it
createStringLogTxt str1
mov eax,[APITable]
lea ebx,[str1]
stdcall dword [eax+DeleteFile],ebx
createStringStartingHyperionLines str1
lea eax,[str1]
stdcall writeLog,[APITable],eax
test eax,eax
jz main_exiterrornolog
createStringStartingHyperion str1
lea eax,[str1]
stdcall writeLog,[APITable],eax
test eax,eax
jz main_exiterrornolog
createStringStartingHyperionLines str1
lea eax,[str1]
stdcall writeLog,[APITable],eax
test eax,eax
jz main_exiterrornolog
stdcall writeNewLineToLog,[APITable]
test eax,eax
jz main_exiterrornolog
```

Figure 3: Logging code in main.asm

Although FASM is a low-level assembler, it does support a number of High Level Assembly statements which can be found in the original implementation of Hyperion. Since MIASM does not support most of those statements, they have been

removed from our implementation. That being the case, certain features that the HLA offers, such as Win32-specific APIs, memory allocation and management were no longer available, and portions of the code had to be rewritten in order to compensate for the fact.

```
;function call

stdcall loadImportTable, [APITable], [loaded_file]



;function declaration

proc loadImportTable, APITable:DWORD, image_base:DWORD

local str1[256]:BYTE, import_table:DWORD,\
      null_directory_entry[sizeof.IMAGE_IMPORT_DESCRIPTOR]:BYTE

        pushad
        ;find import table in data directory
        mov edx,[image_base]
        mov eax,[edx+IMAGE_DOS_HEADER.e_lfanew]
        add eax,edx
        add eax,4
        ;image file header now in eax
        add eax,sizeof.IMAGE_FILE_HEADER
        lea eax,[eax+IMAGE_OPTIONAL_HEADER32.DataDirectory]
        ;first data directory entry now in eax
        add eax,sizeof.IMAGE_DATA_DIRECTORY
        ;import data directory entry now in eax
        mov eax,[eax+IMAGE_DATA_DIRECTORY.VirtualAddress]
        add eax,edx
        ;pointer to import table now in eax
        mov [import_table],eax
```

Figure 4: Function call and function declaration with HLA elements

```
;function call

    push eax ;returned by loadFile
    push dword ptr [ebp+0x8]
    call loadImportTable



;function declaration

loadImportTable:
     push ebp
     mov ebp,esp
     sub esp,0x18
     mov edx,[ebp+0x0C]
     mov edi,[ebp+0x08]
     mov eax,[edx+0x3C]
     add eax,edx
     add eax,4
     add eax,0x14
     lea eax,[eax+0x60]
     add eax,8
     mov eax,[eax]
     add eax,edx
```

Figure 5 Function call and function declaration after removing most HLA elements

6.3 Using PEB technique to load Windows functions

As mentioned earlier, the original implementation of Hyperion imports GetProcAddress and LoadLibraryA functions from kernel32.dll and uses them to import all other functions needed during runtime. In our version we chose to implement an alternative method, used commonly by shellcode writers to locate and import the necessary functions. The method entails manually locating KERNEL32.dll and then parsing its Export Address Table (EAT) to find necessary functions.

This basically renders the original import table useless and allows it to be populated with a number of dummy Windows API functions, (which could later even appear to be invoked by junk code added inline to the original code). This technique can up to a certain point obscure the actual activities of our program and potentially hinder automatic analysis, since a number of antiviruses appear to reach to conclusions regarding the maliciousness of newly discovered programs, by amongst other checks, examining the Imports Table.

KERNEL32.dll is always automatically mapped into a process's address space regardless of the executable's import table. In order to locate it we can take the following steps: In Windows every created thread have their own TEB block Structure (Nowak, 2008). TEB (Thread Environment Block) is a memory block containing system variables. Its base address is stored in the FS segment register. At offset 0x30 we can find the PEB (Process Environment Block), a structure that contains all User-Mode parameters

associated by system with current process. The PEB includes a pointer to LoaderData, which contains linked lists with information on loaded modules (SkyLined & Cipher, 2009). From these lists the location of kernel32.dll can be determined. Depending on the version of Windows, kernel32.dll is the second or the third entry in the "InitializationOrder" list, thus by walking the list and checking the length of the name of each module we can find kernel32.dll (The Unicode string "kernel32.dll" has a terminating 0 as the 12th character, so scanning for a 0 as the 24th byte in the name allows us to find kernel32.dll correctly.)

LDR_MODULE

```
typedef struct _LDR_MODULE {

  LIST_ENTRY              InLoadOrderModuleList;
  LIST_ENTRY              InMemoryOrderModuleList;
  LIST_ENTRY              InInitializationOrderModuleList;
  PVOID                   BaseAddress;
  PVOID                   EntryPoint;
  ULONG                   SizeOfImage;
  UNICODE_STRING          FullDllName;
  UNICODE_STRING          BaseDllName;
  ULONG                   Flags;
  SHORT                   LoadCount;
  SHORT                   TlsIndex;
  LIST_ENTRY              HashTableEntry;
  ULONG                   TimeDateStamp;

} LDR_MODULE, *PLDR_MODULE;
```

Figure 6: LDR Module

The following assembly code snippet implements the locating procedure described above.

```
my_get_apis:
;=============================================================================
                ;get kernel32.dll address
                ;-----------------------------------------------------------
                push ebp
                mov ebp,esp
                xor ecx,ecx
                mov esi,[fs:ecx+0x30]        ;locate PEB
                mov esi,[esi+0x0C]           ;locate LDR
                mov esi,[esi+0x1C]           ;locate InitializationOrder list.

next_module:
                mov ebx,[esi+0x08]           ;ebx carries module's base address
                mov edi,[esi+0x20]           ;edi carries module's name
                mov esi,[esi]
                cmp [edi+12*2],cl            ;compare 24th byte of module name with 0
                jne next_module              ;if not 0, check the next module, else
                                             ;base address of kernel32.dll in ebx
```

Figure 7: Locating kernel32.dll

Having located kernel32.dll we proceed in locating the necessary functions. A well-established method (also used by shellcoders) is to populate our code with precomputated hashes of those functions and iterate through the export table of kernel32.dll during runtime to find functions that their name produce the same hashes (The Last Stage of Delirium Research Group, 2002).

Instead of using one of the publicly available hashing algorithms, included in published shellcode implementations, we chose to develop our own in an attempt to evade signature-based anti-viruses which could potentially identify known implementations as frequently associated with malicious code and therefore suspicious. Our hashing algorithm, as well as the hashes it produces, given the names of the functions we need to locate can be seen in the following figure:

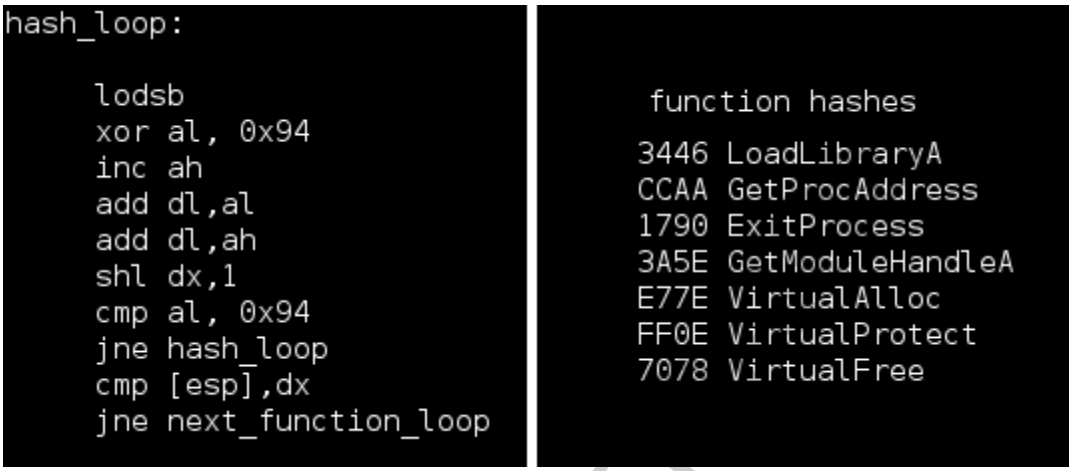

Figure 8: Hashing loop and produced hashes

6.4 Adding polymorphism

One of the biggest issues in the original implementation of Hyperion, one that allowed antiviruses to flag it in a short period of time as malicious, was the static nature of it code. The portions of code responsible for memory decrypting and loading the target executable are hardcoded in the original C code as a FASM template and are basically 100% the same in every executable produced by running Hyperion. We attempted to rectify the issue by using two different methods, taking advantage of the versatility that the MIASM framework offers to the developer.

When constructing a PE file, the elfesteem module of the MIASM framework is creating assembly code blocks , making use of the ASM labels placed by the developer to transfer the flow of execution, as well as labels that itself sets in previous steps of the building process. The framework uses an optimization algorithm to place these blocks in order, based on the calling graph of the final executable amongst other factors (which code block is calling which).

By modifying the portion of the framework's code responsible for the ordering, we managed to make this selection process random. This practically means that every time our implementations creates an new output executable, even if the input executable is the same, the assembly code blocks will be ordered in a different way, creating every time a seemingly different executable file.

As a further step towards antivirus evasion, we used the inherent redundancy found in the x86 instruction: It is possible for certain assembly language instructions to be encoded in machine code in various different ways (Kankowski, 2009). We will use the ADD instruction as an example, to explain this property.

Table 1: ASM redundancy example

opcode	first opcode	mod	r	r/m	instruction
03 C3	ADD reg , reg/mem32	11	000 (eax)	011 (ebx)	ADD eax , ebx
01 D8	ADD reg/mem32, reg	11	011 (ebx)	000 (eax)	ADD eax , ebx

In the IA-32 architecture, the ADD instruction consists of 1 primary opcode byte and an addressing-form specifier byte (the ModR/M byte). The ModR/M byte contains three fields of information, the mod field, the r field and the r/m field. The mod filed value in our example (11B) indicates that the r/m field refers to a register (a general-purpose in case of an ADD instruction). Since the ADD instructions requires a second operand, this information can be found in the r field.

According to the Intel Architectures Software Developer's Manual (Intel, 2014), the opcode 03 corresponds to ADD r, r/m (so the first operand will be taken from r (=000 = eax), and second will be taken from r/m (=011 = ebx). At the same time, the 01 opcode corresponds to ADD r/m, r. (so the first operand will be from r/m (=000 = eax), and

second will be from reg (=011 = ebx).Combining all the above, we can infer that both 03 C3 and 01 D8 can be disassembled as ADD eax, ebx.

The assembler in MIASM framework provides all possible opcode versions for an instruction.

```
>>> print x86_mn.asm('add eax,ebx')
['\x01\xd8', '\x03\xc3']
>>>
>>>
>>> print x86_mn.asm('add eax,1')
['\x83\xc0\x01', '\x05\x01\x00\x00\x00', '\x81\xc0\x01\x00\x00\x00']
>>>
>>>
>>> for a in x86_mn.asm('add dword ptr [esi],edi'):
...  print repr(a)
...
'\x01>'
'\x01<&'
'\x01<f'
'\x01<\xa6'
'\x01<\xe6'
'\x01~\x00'
'\x01|&\x00'
'\x01|f\x00'
'\x01|\xa6\x00'
'\x01|\xe6\x00'
'\x01\xbe\x00\x00\x00\x00'
'\x01<5\x00\x00\x00\x00'
'\x01\xbc&\x00\x00\x00\x00'
'\x01\xbcf\x00\x00\x00\x00'
'\x01\xbc\xa6\x00\x00\x00\x00'
'\x01\xbc\xe6\x00\x00\x00\x00'
```

Figure 9: ASM redundancy

Although during building of an executable file the elfesteem module is configured by default to choose the first item of the provided options list, by modifying its code, we were able to force the framework to randomly choose between the available options. This effectively means, that the code of each output file created will be significantly different.

7 Testing

In order to test the effectiveness of our methods a testing environment was created. We built 15 different Virtual Machines running Windows, using VMware Workstation that shared a folder with the host machine. In each of the VMs a different antivirus was installed and was allowed to update itself. A VM running Kali Linux was then added to our lab, which was configured to handle shell connections.

5 different binaries were used for our testing.

- A Windows reverse tcp shell executable, created with Metasploit, (Offensive Security, 2014) configured to connect to our Kali running machine (1.exe)
- 1.exe packed with the original version of Hyperion (vanilla_1.exe)
- 1.exe packed with our version (alpha_1.exe)
- A totally harmless "Hello World" exe packed with the original version of Hyperion (vanilla_HW.exe)
- The same Hello World exe packed with our version (alpha_HW.exe)

The testing results can be found in the following table:

Table 2: Test results

	1.exe	vanilla_HW.exe	vanilla_1.exe	alpha_HW.exe	alpha_1.exe
Avast	flagged	flagged	flagged	flagged	flagged
AVG	flagged	OK	flagged	OK	OK
Avira	flagged	flagged	flagged	flagged	flagged
Bitdefender	flagged	flagged	flagged	flagged	flagged
ESET NOD32	flagged	flagged	flagged	OK	OK
Kaspersky	flagged	OK	flagged	OK	OK
Malwarebytes	flagged	OK	OK	OK	OK
McAffee	flagged	OK	OK	OK	OK
Microsoft S.E.	flagged	OK	flagged	OK	OK
Norton	flagged	flagged	flagged	OK	OK
Panda	OK	flagged	flagged	OK	OK
Sophos	flagged	OK	OK	OK	OK
Symantec	flagged	OK	OK	OK	OK
Trendmicro	flagged	OK	OK	OK	OK
Webroot	OK	OK	OK	OK	OK

It is evident from the results that only three of the tested antivirus were able to identify our sample as malicious. The rest allowed its execution and a command shell was presented to us in our Kali running machine, offering us unfettered access to the targeted machines in the context of the user that executed the sample.

8 Conclusion and Further Work

Antivirus evasion is no trivial task. Antivirus vendors constantly devise new methods for flagging and rooting out malicious code, in a constant arms race against malware writers. The security professional that has to emulate the actions of hypothetical attackers, is involuntarily being made part of the same equation, often finding herself in need to evade defenses in order to stealthily infiltrate her targets virtual perimeter.

Based on our test results, our PE crypter implementation can be a valuable aid towards achieving this goal. The fact that certain AV were able to flag it does point out the need for further calibrating. The introduction of anti-emulation techniques are expected to improve the undetection rates, such as those that allow the crypter to detected that is being executed by an antivirus in an emulated environment and exit out gracefully without even decrypting its payload.

Since the original goal was to evade automated analysis, no steps were taken to prevent or hinder manual dissection of the crypter by a human analyst. However, as further work, the addition of Anti-Debugging, Anti-Disassembly and Anti-VM techniques could be definitely proposed.

9 References

Ammann, C. (2012, May 8). Hyperion: Implementation of a PE-Crypter. Retrieved from http://www.nullsecurity.net/papers/nullsec-pe-crypter.pdf

Ammann, C. (2012). *Presentation of the Hyperion runtime encrypter for portable executables at Berlinsides 2012.* Retrieved from http://www.nullsecurity.net/papers/nullsec-bsides-slides.pdf

Desclaux, F. (2012). Miasm: Framework de reverse engineering. *Symposium sur la sécurité des technologies de l'information et des communications.* Rennes. Retrieved from https://www.sstic.org/media/SSTIC2012/SSTIC-actes/miasm_framework_de_reverse_engineering/SSTIC2012-Article-miasm_framework_de_reverse_engineering-desclaux_1.pdf

Glinos, D. (2012). Packing Heat! Retrieved from http://census-labs.com/media/packing-heat.pdf

Grysztar, T. (n.d.). *flat assembler 1.71 Programmer's Manual.* Retrieved from http://flatassembler.net/docs.php?article=manual

Guillot, Y. (n.d.). Metasm, the Ruby assembly manipulation suite.

Hanel, A. (2001, January). *An Intro to Creating Anti-Virus Signatures*. Retrieved from http://hooked-on-mnemonics.blogspot.gr/2011/01/intro-to-creating-anti-virus-signatures.html

Harbour, N. (2008, August). Advanced Software Armouring and Polymorphic Kung-Fu. Retrieved from https://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-harbour.pdf

Harley, D., & Lee, A. (2008, January). Heuristic Analysis - Detecting Unknown Viruses. ESET. Retrieved from http://www.eset.com/us/resources/white-papers/Heuristic_Analysis.pdf

Iczelion. (2002). Overview of PE file format. Retrieved from http://win32assembly.programminghorizon.com/pe-tut1.html

Intel. (2014). Intel Architectures Software Developers Manual. Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

Kankowski, P. (2009). Redundancy of x86 Machine Code. Retrieved from http://www.strchr.com/machine_code_redundancy

Kath, R. (1994). The Portable Executable File Format from Top to Bottom.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Forensic Research Workshop.* Lafayette: Digital Forensic Research Workshop.

Kumar, A., Sharma, V., & Kumar, S. (2011). A Comparative Analysis of Various Exact String-Matching Algorithms for Virus Signature Detection. *First International Conference on Emerging Trends in Soft Computing and ICT(SCICT-2011)* (pp. 162-166). Bilaspur, India.

Nowak, T. (2008). *ntinternals.net*. Retrieved from http://undocumented.ntinternals.net/

Offensive Security. (2014). Metasploit Unleashed. Retrieved from http://www.offensive-security.com/metasploit-unleashed/Binary_Payloads

Pietrek, M. (2002, February). An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*. Retrieved from http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

SkyLined & Cipher. (2009). *Shellcode/kernel32*. Retrieved from http://skypher.com/wiki/index.php?title=Hacking/Shellcode/kernel32

The Last Stage of Delirium Research Group. (2002, December 12). Win32 Assembly Components. Retrieved from http://lsd-pl.net/projects/winasm.zip

Zeltser, L. (2001, October). *How antivirus software works: Virus detection techniques*. Retrieved from http://searchsecurity.techtarget.com/tip/How-antivirus-software-works-Virus-detection-techniques