BASICS ON CRYPTERS AND BINDERS

Author(s): Deque, Insider.

This article was first posted on GREYSEC.NET. The document contains the thread's initial message. To follow the full discussion, please visit https://greysec.net/showthread.php?tid=6814

Transcribed from Deques book on Crypters. An old acquaintance from our old forums **Hacksociety & Hackcommunity**. Can't find her github these days so reposting this golden gem here. Great for understanding how crypters work. I take no credit for this work. Keep in mind this whitepaper is from 2014, some things have changed; Like PE-method which is protected by DEP *(unless you inject it into your own process)* but still contains a lot of useful information.

Contents

Introduction

If you thought about using this guide to write a crypter or binder, you may do so, but don’t expect any hacker to be impressed. There might be some misguided *skript kiddies* that will be, but writing a piece of software someone else has invented is not an achievement, nor is this hacking in any way.

If you need a crypter for your own purpose then you might be surprised: You don’t need one, once you are capable of writing one. The only purpose of crypters is to get money out of the pockets of *skript kiddies* who don’t know how to create malware in a way that it can’t be detected by antivirus scanners. These *skript kiddies* are dependent on other’s tools and are often enough willing to pay.

Target Audience

This paper is made for everyone who needs an introduction or overview about *cryptersand* binders, what they actually are and how they work. It explains the core concepts and techniques that can be used. You might be interested in malware analysis, or just curious about the terms that are so frequently used in hacking communities. In that case this paper is probably for you. The paper provides source codes with explanations.

The source code listings are usually no fully functional programs *(some links to full sources are provided, though)*. This is not a step-by-step guide on how to build a crypter. The only purpose of the source codes is to help with understanding of the concepts by giving examples. The examples use various programming languages, depending on the purpose and which language seems best to understand while explaining the concepts. I take as a given that you have basic programming knowledge. The reason this paper covers both, crypters and binders, is that a crypter is actually a modified binder (see chapter 2). Techniques used for binding files are valid for crypters too. In case you are only interested in the crypter part, read the binder part as well. You will need it.

Packer

Before we start diving into the inner workings of binders and crypters it is important to know what we are talking about. A frequently asked question in hacking communities is the difference between crypters and packers. This chapter will answer this question and also explain other related terms.

Packer Classification

- **Definition 2.** A packer is a program that packs an executable file by putting it in a software envelope. The packer usually also modifies the executable, but retains the original functionality. The software envelope is also called **stub** and has the purpose to unpack and run the file.
- **Definition 3.** A target is the executable file that shall be operated on by a packer. The target is not modified by the packer at this point.
- **Definition 4.** A packed file is the target after the packer has modified it.

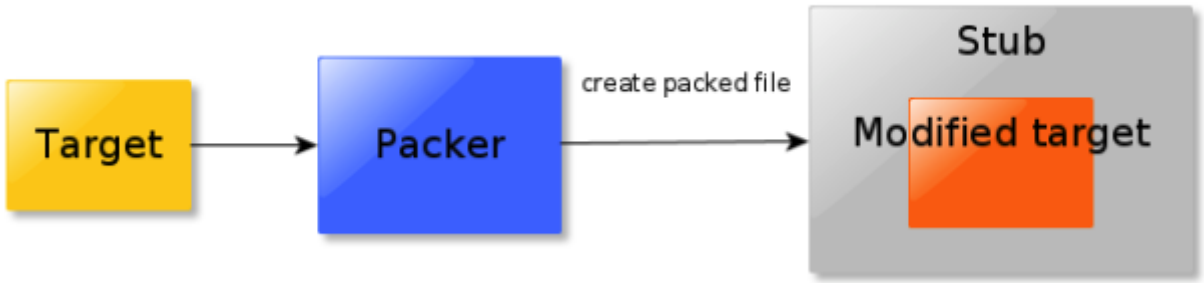

Figure 2.1: The terms target, packer, stub and packed file. Packer modifying the target and putting it in a software envelope, the so called stub. The resulting file is the packed file.

There are four categories of packers: crypters, bundlers, compressors and protectors:

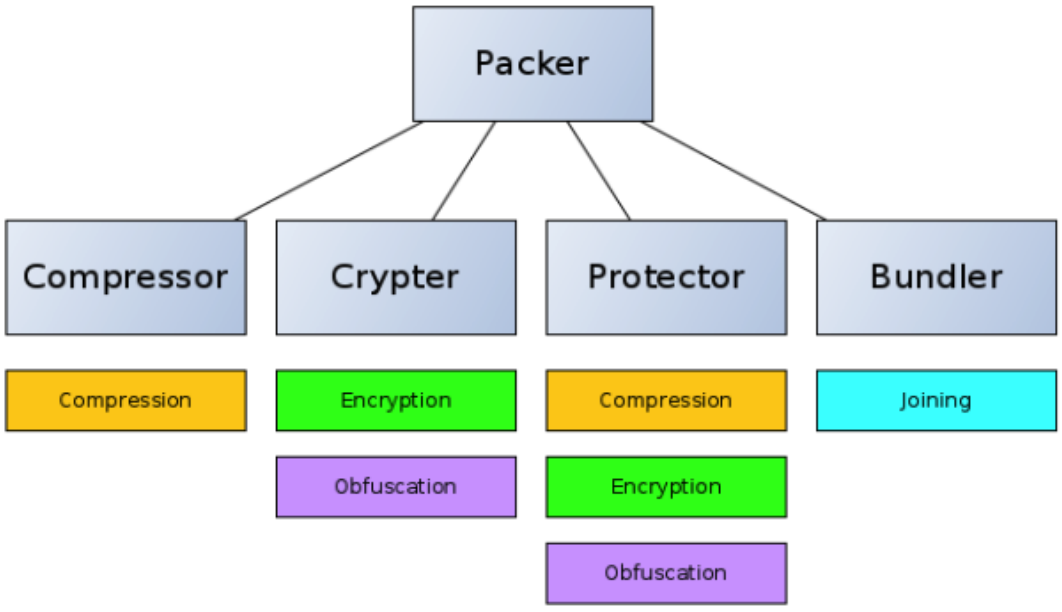

Figure 2.2: Classification of packers.

- **Definition 5.** A compressor is a packer whose only purpose is to shrink the file size of the packed file by using a compression algorithm. A compressor applies no or very little anti-unpacking techniques.
- **Definition 6.** A crypter is a packer that encrypts and often obfuscates the target. A crypter is often used to pack malware with the purpose of making the malware undetected by antivirus software. If a crypter is UD, it means the resulting executable file is undetected by some antivirus scanners. A fully undetected or FUD crypter will produce executables that are undetected by all antivirus scanners. Crypters are grouped into runtime and scantime crypters. More about that in later sections.
  Definition 7. A protector is a combination of a compressor and a crypter. It compresses and encrypts the target before putting it into the software envelope. As well as a crypter, a protector might also apply obfuscation.
- **Definition 8.** A bundler is a packer that joins several files *(executables as well as non-executables)* to a single executable.

Binder, Joiner

Definition 9. A binder or joiner are other terms for a bundler (Definition 2.1). The terms binder or joiner are usually only used in the context of bundling malicous targets with non-malicous targets in order to disguise the purpose of the malicious program. Some people also call programs binder that don't have an executable as packed file, but hide files or data within other non-executable files instead. E.g. hiding a file in an image is very common.

These applications are actually *steganography* programs and not binders. If you see a binder that packs executables and images *(or other non-executable files)* you most certainly won't get an image as packed file that will run your executable.

Malware Detection by Antivirus Scanners

The understanding of the detection methods used by antivirus scanners is vital for understanding the countermeasures crypters are taking in order to avoid detection. Therefore the two main detection types, *signature based detection* and *heuristic analysis*, are explained in this chapter.

Signature based detection is still the main countermeasure of antivirus software today. A signature is a regular expression that represents a byte sequence of a certain malware. These signatures are stored in a database and upon scanning a file the contents of the file are matched against the signatures. Signatures have to be unique enough to identify a malicious program and avoid false positives. Different antivirus scanners may use different signatures for the same malware.

The drawback of this approach: Zero-day malware or malware that mutates itself to new variants can not be recognized as malicious because the signature is not in any of the databases. The antivirus scanners always have to be updated in order to work against the new threats. **Imperva** states in a study from 2012: *"For certain antivirus vendors, it may take up to four weeks to detect a new virus from the time of the initial scan."*

For the first antivirus scanners, signatures had to be identified manually. So there was always a person reverse engineering the malware in order to find a signature. Now that there are more than 220,000

new malwares every day, antivirus companies rely on routines that find signatures automatically. This covers most of the malware. Signature based detection is almost useless against polymorphic or metamorphic malware. The reasons will be explained in later sections.

Heuristics find a solution for a problem using incomplete knowledge. The found solution might not be optimal, but good enough. Heuristic analysis is able to recognize new threats based on the knowledge how malicious programs typically behave or look like. A static method for heuristic analysis is disassembling the binaries and analysis of the resulting assembly code for suspicious patterns. If a certain amount of the source matches suspicious patterns, the file is declared as possibly dangerous.

A *dynamic heuristic detection method* is the emulation of the suspicious program. The emulation or simulation happens in a virtual machine and the behavior of the suspicious program is analyzed. Suspicious behavior can be: Overwriting of files, replication, hiding techniques. Malware that is encrypted on disk can be dumped in memory in its decrypted state while it is emulated. Subsequently a signature based detection can be performed as well.

While heuristic analysis is able to detect new malware, the accuracy of the results is very low. False positives happen too often and if a malware uses new methods that are unknown to the antivirus scanner, the malware won't be detected as well. A study in 2012 by **Imperva** revealed: *"The initial detection rate of a newly created virus is less than 5%."* The study used viruses found in honeypots and released at hacking communities. So take into account that malware writers scan their creations and put work into *undetection* before they release them into the wild. Otherwise the detection rate would probably be much higher.

Static vs Dynamic Detection

Static analysis examines the potentially malicious files without running them. The following methods are used:

- String scanning method: Signature based detection where the signature is a sequence of bytes.
- Wildcards method: Signature based detection that allows to skip bytes or byteranges. I.e. a "?" in the signature would skip one character.
- Mismatches method: A certain number of bytes can be of arbitrary value. The position of these bytes doesn't matter.
- Generic detection method: One signature is used to detect all/several variants of a malware or all/several members of the same malware family.
- Bookmarks method: Takes the distance between the start of the virus body and the signature string into account.
- Smart scanning: Is able to skip junk instructions, i.e. NOP instruction. The signature doesn't contain them as well.
- Skeleton detection: The scanner drops all non-essential statements from the malware code. As a result only the skeleton of the virus body is left.
- Static heuristic analysis: See more sections below.
- Virus specific detection: If the standard algorithm of the antivirus scanner is notable to deal with certain malwares, a specific detection routine for this virus is implemented.

Dynamic detection methods run the potentially malicious files and observe their behavior. This is mostly heuristic analysis. An example for a typical dynamic signature of a virus might be:

- Opening an executable, with both read and write permission.

- Reading the portion of the file header containing the executable's start address.
- Writing the same portion of the file header.
- Seeking to the end of the file.
- Appending to the file.

Antivirus software often include so called *behavior blockers*. These programs monitor the behavior of other programs in real-time. If they observe a suspicious behavior, the observed program is blocked and the user is asked, what to do with it. In contrast to real-time observation is the emulation in a virtual machine.

Inner Workings of a Binder

The general functionality of a binder is illustrated in figure 4.1. The binder takes two files and a stub *(the software envelope)* and embeds the files to the stub. This way the code of the stub is invoked if the packed file is executed. The stubs purpose is to extract and run the embedded files.

How a binder can bundle the files depends on the possible file formats of the targets and the file format of the packed file. Most binders have the PE file format with the file ending *.exe* as the packed file. The following techniques will cover those binders. Binders that work with other file formats will be mentioned in the last section.

Portable Executable

Definition 10. A Portable Executable or, short, PE, is a Windows file format for 32-and 64-bit architecture. It includes EXE, DLL, SYS, FON and other file types. The PE is especially interesting in the context of crypters and binders, because most of them are made for use with Windows EXE files and have an EXE as packed file. Knowledge about the PE file format and how a PE is loaded into memory is necessary for advanced implementations.

Using the Overlay of the PE

This is also referred to as EOF method, where EOF means end of file. The overlay of a PE is the data that is appended to the file, but not mapped by the PE format. An overlay can easily be detected, but is also used by legal programs in order to store data. The following listing of **Python** code shows how simple it is to write one file to the overlay of another.

```
with open("infile.exe", "rb") as in_file:
    with open("outfile.exe", "a+b") as out_file:
          out_file.write(in_file.read())
```

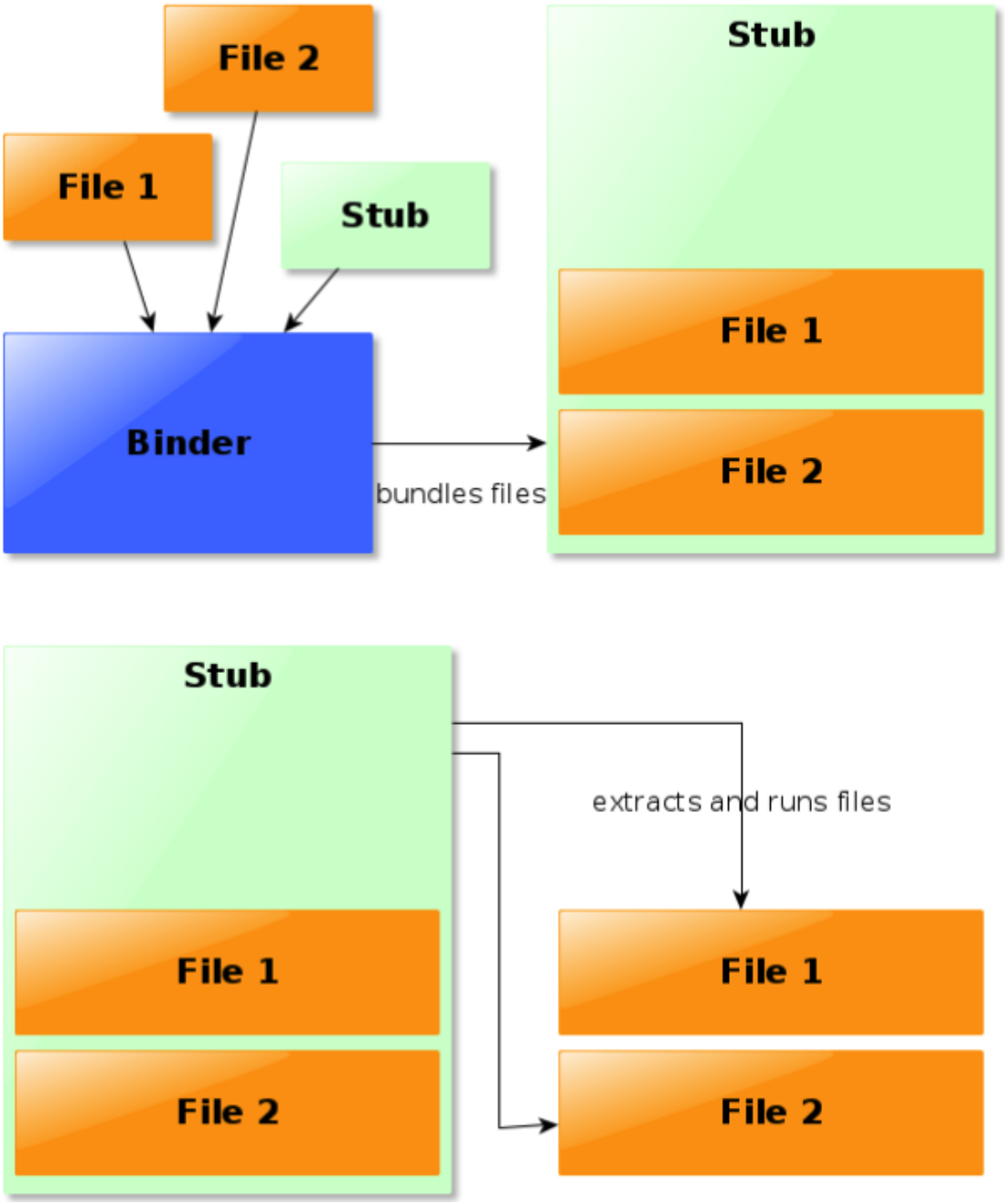

Figure 4.1: A binder bundling the files and the stub to a single executable. The stub extracts the files from its own body in order to run them separately.

But there is more you need. Appending a file to another doesn't execute the appended file. The appended file will just be ignored. So you actually have to append your file to a program that executes its own overlay. This is the software envelope or stub mentioned in definition 2.1

A stub can either write the files to disk and execute them afterwards or run the programs in memory only by injecting them into a process. The basics for running a program in memory will be covered in section 5.5. For now we will stay with writing the file to disk for simplicity. Since you

want to bind two or more files, you will have to write all files that shall be binded to the stub's overlay.

To distinguish the beginning and end of each file, some binders use markers. These are signatures the stub scans for. Beware that these signatures make a perfect signature for antivirus scanners as well. Here is a source sample in **AutoIt** of a file binder that uses markers to join two executables:

```
Func bind()
    $path1=GUICtrlRead($input1)
    $path2=GUICtrlRead($input2)
    $path3=GUICtrlRead($input3)
    If FileExists($path1) And FileExists($path2) Then
          $marker=StringToBinary("SOLIXIOUS")
          $output=FileOpen($path3,1+2+8+16)
          $inputx1=FileOpen($path1,0+16)
          $inputx2=FileOpen($path2,0+16)
          $data1=FileRead($inputx1)
          $data2=FileRead($inputx2)
          $stub=FileOpen(@ScriptDir & "\Stub.exe",0+16)
          $stubdata=FileRead($stub)
          FileWrite($output,$stubdata)
          FileWrite($output,$marker)
          $data1=encrypt(BinaryToString($data1))
          $data2=encrypt(BinaryToString($data2))
          FileWrite($output,StringToBinary($data1))
          FileWrite($output,$marker)
          FileWrite($output,StringToBinary($data2))
          FileClose($output)
          FileClose($inputx1)
          FileClose($inputx2)
          MsgBox(64,"Successful","Files Binded Successfully!")
          GUICtrlSetData($input1,"")
          GUICtrlSetData($input2,"")
          GUICtrlSetData($input3,"")
    EndIf
EndFunc
```

Listing 4.1: Excerpt of Hack Community Binder by **Solixious**.

The corresponding stub parses for the marker to see where the files start and end, reads both files, writes them to disk and runs them:

```
$path=@ScriptFullPath
$file=FileOpen($path,0+16)
$data=FileRead($file)
$marker="SOLIXIOUS"
$strdata=BinaryToString($data)
$ardata=StringSplit($strdata,$marker,1)
HotKeySet("+!c","quit")
```

```
If @error Then
    MsgBox(0,"No Delimiters","No delimiters Found!")
Else
    If $ardata[0]=3 Then
          $ardata[2]=decrypt(BinaryToString($ardata[2]))
          $ardata[3]=decrypt(BinaryToString($ardata[3]))
          $file1=FileOpen(@TempDir & "\GraphicsAccelerator.exe",2+16)
          FileWrite($file1,$ardata[2])
          FileClose($file1)
          $file2=FileOpen(@TempDir & "\GraphicsService.exe",2+16)
          FileWrite($file1,$ardata[3])
          FileClose($file2)
          ShellExecute(@TempDir & "\GraphicsAccelerator.exe")
          ShellExecute(@TempDir & "\GraphicsService.exe")
          While 1
              ;
          WEnd
    EndIf
EndIf
```

Listing 4.2: Excerpt of Hack Community Binder stub by **Solixious**.

Another possibility is to write the number of appended files and their sizes to a certain offset of the stub. For example if you append two files to the stub, the first is 643000 bytes and the second 700000 bytes, you could append the following to the stub:

```
2|643000|700000|<bytes_of_file1><bytes_of_file2>
```

Since the stub size is known, the offset where the file information is written to can be *hardcoded*. For a full source sample of an overlay binder see **Solixious**' file binder at web.archive.org.

Embedding Files into the Resource Section

A PE file consists of several sections. The resource section is one of them. It is used to store data, pictures, strings and similar resources the file might need. This technique is also referred to as resource method. You need to use the Windows API or a PE library that is capable of editing PE files to add something to the resource section. The following source code demonstrates how the resource section can be accessed by astub with the Windows API.

```
HRSRC hrsrc = NULL;
HGLOBAL hGlbl = NULL;
BYTE *pExeResource = NULL;
HANDLE hFile = INVALID_HANDLE_VALUE;
DWORD size = 7168; //hardcoding the size of the exe resource (in bytes)

hrsrc = FindResource(hInstance, (LPCWSTR)IDR_FOO, RT_RCDATA);
if(hrsrc == NULL)
    return FALSE;
```

```
hGlbl = LoadResource(hInstance, hrsrc);
if(hGlbl == NULL)
    return FALSE;

pExeResource = (BYTE *) LockResource(hGlbl);
if(pExeResource == NULL)
    return FALSE;

hFile = CreateFile(L"\\Voila.exe", GENERIC_WRITE|GENERIC_READ, 0, NULL,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

if(hFile != INVALID_HANDLE_VALUE)
{
    DWORD bytesWritten = 0;
    WriteFile(hFile, pExeResource, size, &bytesWritten, NULL);
    CloseHandle(hFile);
}


intret = CreateProcess(L"\\Voila.exe", NULL, NULL, NULL, FALSE, 0, NULL,
NULL, NULL, &pi);
```

Listing 4.3: C++ source code of a stub using the resource section.

Other PE binding Techniques

Most of the other methods of embedding files in a PE also make use of the Windows API. You need a basic understanding of the PE format in order to use these techniques or to come up with new ones. You might want to spare your time by looking into a PE tutorial. Some other techniques are:

- Adding a new section and storing the file it.
- Appending the data to the last section after *resizing* that section.
- Injecting the data into the source of the stub and compiling the stub on the fly.
- Using slack spaces to store the file.
- Creating artificial slack spaces, i.e. by moving a section, and storing the file there.

It is up to your imagination what else can be done to bind files.

Binding Techniques for Other File Formats

PE files are not the only files that are executable. I.e. You also have *runnable* **Java** archives *(file ending .jar)*, ELF *(Executable and Linkable Format, Linux)*, Mach-O *(Mac OS)*, COM *(MS-DOS)*, **Batch** files or other source files that can be executed. I will ignore the old formats like COM for now as they aren't used much any more. I will also ignore the operating system specific file formats for other operating systems than Windows, because most malware and especially crypters and binders are written for Windows. But if you understood the principles, applying this to other file formats is just a matter of reading format specifications and using common sense. Binding source files is only interesting in certain circumstances. Usually there are

no binders created to do that as you can bind sources manually in a text editor. But sometimes viruses use source files or *bytecode* files to spread. That's another topic, though.

Java archive files however need an entirely different approach. A *runnable* Java archive is just a ZIP file with a MANIFEST that denotes where the main method can be found. This binder saves the files to be binded as entries in a Java archive and defines the main method in the MANIFEST as the one in the stub, which is a .class file. On execution the stub extracts the files as temporary files and runs them.

Inner Workings of a Crypter

Now that you know how a binder works, the crypter is only one step away from it. The only difference is that a crypter only puts one file in the software envelope and applies an encryption and possibly obfuscation to the target *(see figure 5.1)*. As the goal of a crypter is mostly to *undetect* a file from antivirus software *(see antivirus detection in chapter 3)*, more techniques have been created to make the resulting files stay undetected as long as possible. But let's start with the simple things.

Encryption

As explained in chapter 3, malware is mostly detected by pattern matching of byte sequences. A crypter applies an encryption to the target, which makes it impossible to do any signature based detection on the encrypted part. The only part of the packed file that may contain signatures that can be matched is the stub. This is the reason that the stub will be recognized as malware after it was used to carry several malware files, although the stub doesn't contain malicious code.

Because of that a lot of work is put into the modification of stubs to make them undetected again *(see section 5.7)*. A detected stub renders the crypter useless. The encryption however doesn't need to be a secure one to work. Actually it can't be secure as the stub has to be able to apply a decryption. So the stub needs to know the key. There are several methods to deal with the key:

1. The key is *hardcoded* to the stub, a different key can be applied by the crypter to the stub for every packed file.
2. The key is generated in the stubs code.
3. The crypter applies a random key which is *bruteforced* by the stub *(an example is the crypter Hyperion)*.

An example for method 1 is Solixious ' **HC-Crypterin AutoIt**. The target is encrypted with AES-256 and the keyword SOLIXIOUS. The crypter also appends the string SOLIXIOUS as marker to the stub and afterwards the encrypted file.

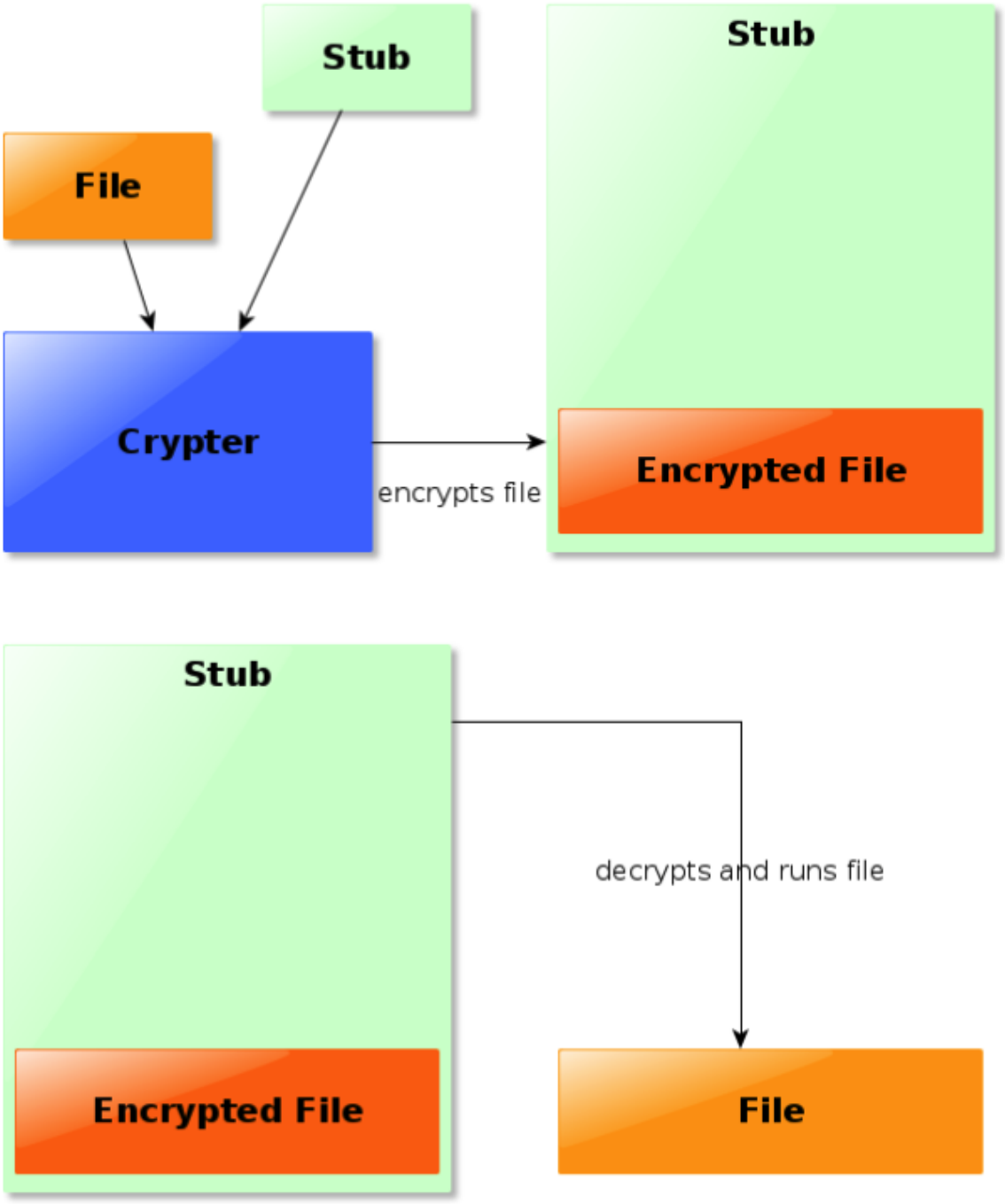

Figure 5.1: The crypter encrypts the target and embeds it into the stub. The stub *decrypts* the embedded file and runs it from disk or by injecting it into another process.

```
Func encryptfile()
    $path1=GUICtrlRead($hInp1)
    $path2=GUICtrlRead($hInp2)
    If FileExists($path1) Then
          $marker=StringToBinary("SOLIXIOUS")
          $output=FileOpen($path2,2+8+16)
          $inputx1=FileOpen($path1,0+16)
          $data1=FileRead($inputx1)
          $stub=FileOpen(@ScriptDir & "\Stub.exe",0+16)
          $stubdata=FileRead($stub)
          FileWrite($output,$stubdata)
          FileWrite($output,$marker)
          $data1=encrypt(BinaryToString($data1))
          FileWrite($output,StringToBinary($data1))
          FileClose($output)
          FileClose($inputx1)
          GUICtrlSetData($hInp1,"")
          GUICtrlSetData($hInp2,"")
    EndIf
EndFunc


Func encrypt($data)
    $data=_Crypt_EncryptData($data,"SOLIXIOUS",$CALG_AES_256)
    Return $data
EndFunc
```

Listing 5.1: Excerpt of **HC-Crypter** by **Solixious.**

Antivirus companies sometimes apply unpacking techniques for known packers in order to identify the embedded files within. Method 3 is a countermeasure to that. The packed file has a lot of time to crack the own encryption whereas antivirus scanners need to scan so many files that speed is a real issue. They don't have the time for *bruteforcing*.

But as mentioned in section 3.1.2 the file can also be dumped in memory while it is initial decrypted state. Since the stub is usually detected by antivirus programs, countermeasures have been developed that shall make the stub undetected as long as possible. This is the topic of the next section.

Oligomorphic, Polymorphic and Metamorphic

Let's take a step back to viruses and how they avoid detection. Don't confuse the term virus and the term malware. Not every malicious program is a virus.

- **Definition 11.** A Virus is computer program that replicates itself, usually by infecting a host file.

As shown in chapter 3, antivirus scanners mostly rely on signature based detection routines to identify a malicious program. The first countermeasure of viruses was the encryption of the virus body, so that only the code of the decrypter was still visible. Unless the repair code of the antivirus scanner is able to decrypt the body of the virus, only the decrypter code can be used for signature based detection. This makes the detection accuracy worse, but in many cases the decrypter has still enough code to find a signature for it.

At some point virus writers got the idea to store several decrypting routines in the virus body and with each replication of the virus, another decrypter would be used. So in case one decrypter was detected by an antivirus scanner, the others could still be fine. This is called **oligomorphic code**.

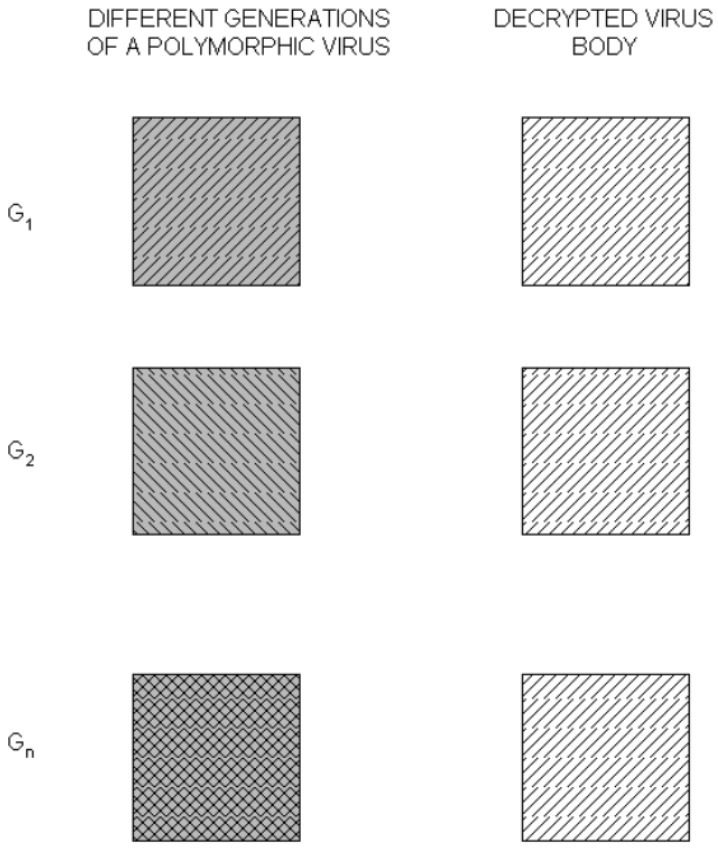

Figure 5.2: Generations of a polymorphic virus *(picture by [12])*. The encrypted body and the decrypter code change, but the decrypted virus body stays the same.

- **Definition 12.** An oligomorphic virus changes its decrypter with each generation intoa new, but semantically equivalent form. The number of possible forms is small.

These viruses can still be detected by signature based methods, but the antivirus scanner needs one signature for all decrypters used. This is impractical, but possible. So the virus writers came up with a way that renders signature based detection useless.

- **Definition 13.** A polymorphic virus mutates its decrypter with each generation, so that the number of possible mutations is very high, i.e. several million.

The virus body of a polymorphic virus still stays the same in each generation, but is encrypted differently (see picture 5.2). Polymorphic viruses are challenging to write and many of such viruses that where released to the wild contained a lot of bugs.

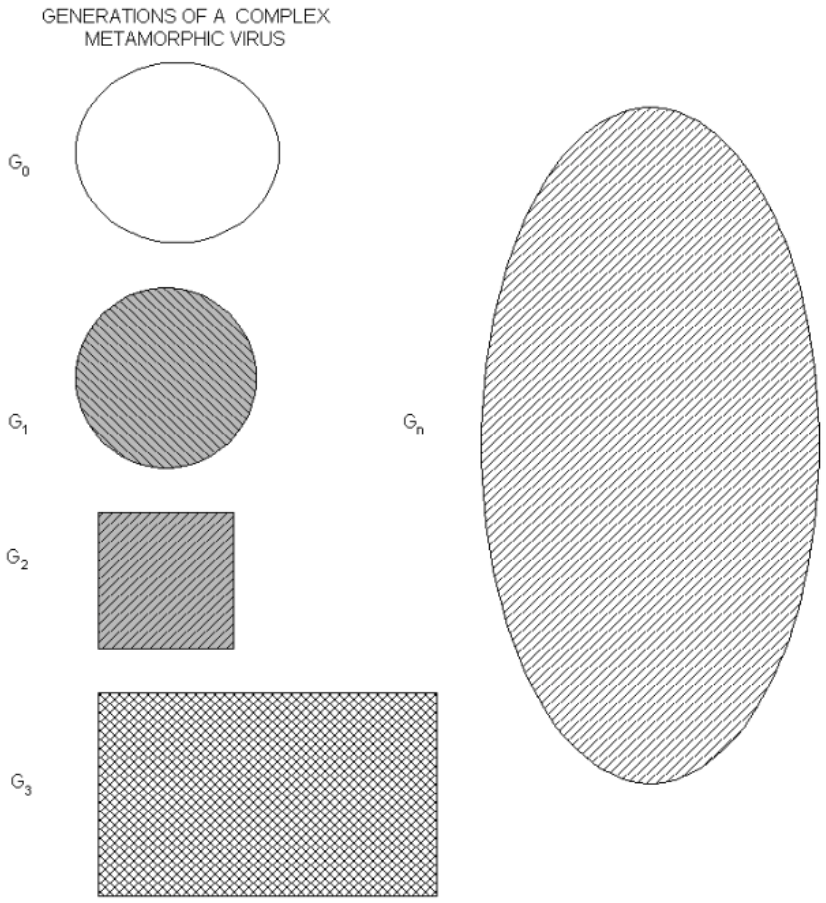

Figure 5.3: The different shapes representing the mutations of a metamorphic virus.

Which made it possible in some cases to find a signature or a generic decryption routine nevertheless. Otherwise dynamic decryption (emulating the virus and dumping the decrypted virus body in memory) and a subsequent signature based detection is needed to identify these viruses. Even more challenging (for both sides) are metamorphic viruses.

- **Definition 14.** A metamorphic virus mutates its body. It has no decrypter and no constant body.

"Metamorphic viruses use several metamorphic transformations, including Instruction reordering, data reordering, in-lining and outlining, register renaming, code permutation, code expansion, code shrinking, Subroutine interleaving, and garbage code insertion."

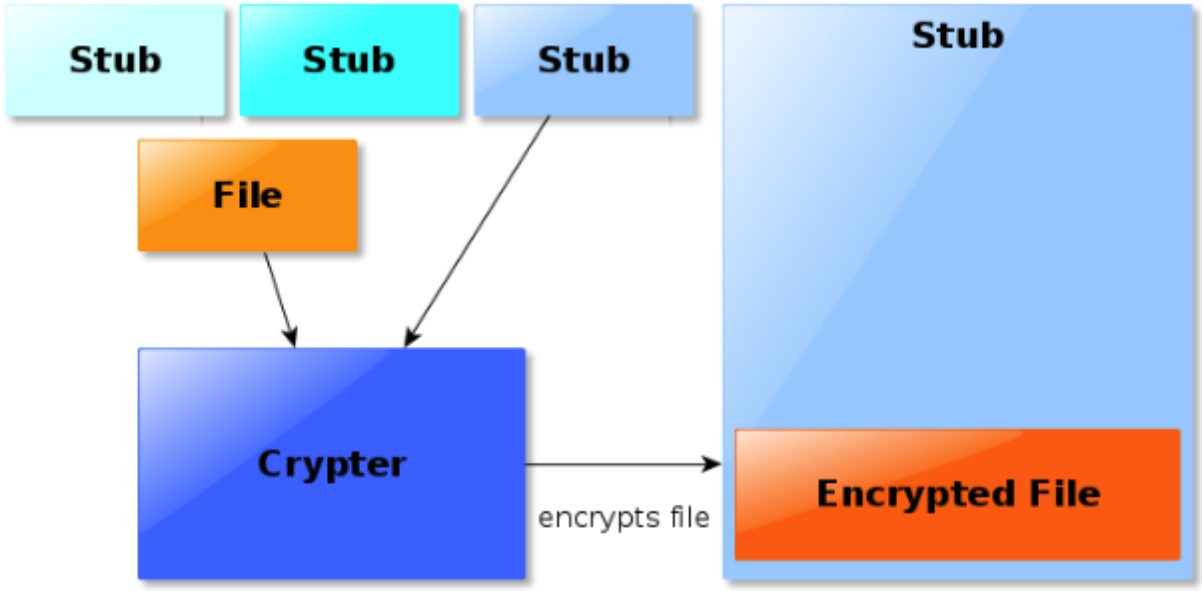

Figure 5.4: Crypter with oligomorphic stub generation. The crypter chooses random lyone of several available stubs.

Metamorphic viruses are the most advanced and difficult ones by now. There are not many virus writers capable of writing *metamorphics*. Some virus writers make their task easier by mutating a high-level language representation of the virus and compiling it to machine code. But these viruses need a compiler on the target machine to evolve. How is this related to crypters? Crypters can use the same techniques and are often advertised with having a polymorphic engine. More about that follows in the next section.

Unique Stub Generation

- **Definition 15.** Unique stub generation is a feature of a crypter, which is able to generate a high number of different stubs.

As the stub is the part that is detected by antivirus scanners, the generation of new stubs helps to

prevent newly created packed files to be detected as well. Stub generators apply the same techniques as described in the last chapter. They can have a set of stubs they might choose from *(similar to oligomorphic code, see figure 5.4)*, but they can as well generate a very high number of stubs by using a stub template that the stub generator changes randomly. Similar to polymorphic viruses each packed file will have a different stub code and encryption for the embedded target *(see figure 5.5)*.

Please note the difference to actual polymorphic code. Crypter advertisers who claim to use polymorphic code, often just have a unique stub generator *(USG)*. But the generated stubs themselves are not polymorphic. So if a crypter user packs a malware once and spreads it, it will always have the same appearance. If it is detected by an antivirus scanner, all of the other instances of that malware will be detected too. While it is possible that crypters actually create polymorphic stubs, most of them don't.

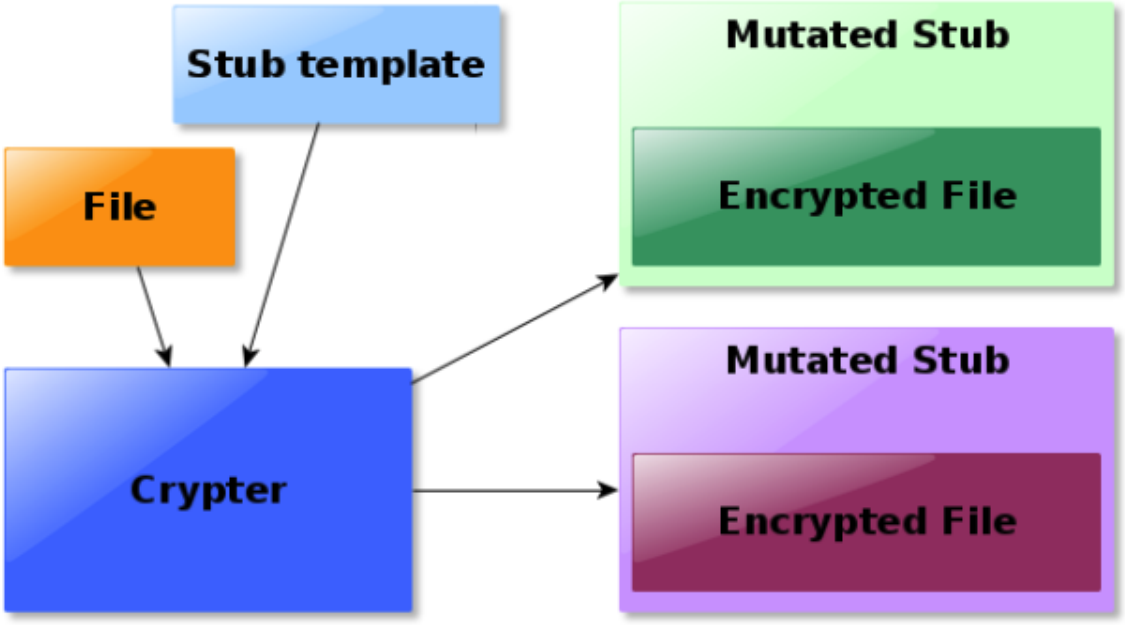

Figure 5.5: Crypter with polymorphic stub generation. Packed files always look different as the stub and the encryption are mutated.

Obfuscation

- **Definition 16.** Obfuscation changes source code to semantically equivalent source code that is hard for any human to understand.

The purpose of obfuscation is to prevent reverse engineering of the code. Some crypters not only apply encryption, but also obfuscation techniques. Obfuscation can be realized by:

1. Reordering of instructions *(for some instructions the order of execution doesn't change the programs semantics)*.
2. Renaming of variables or subroutines.
3. Adding conditional statements or jumps *(i.e. conditional statements might always yield true)*.
4. Adding unreachable code *(the code will never be executed in the program)*.
5. Adding rubbish code *(code that does only trivial calculations which are not necessary for the program)*.
6. Encryption of strings used in the program.
7. Using equivalent formulas and transformations *(i.e. you can calculate 10 by the formula x= 20 / 2 or x = 7 + 3)*.
8. Replacing instructions by subroutine calls.
9. Splitting data structures *(i.e. splitting an array into two arrays)*.
10. *Anti debugging* techniques *(the program will terminate or behave differently if it is run in a debugger)*.

Obfuscation can be done manually or by using an **Obfuscator** *(a program that obfuscates code automatically)*. An example is **yGuard** that obfuscates Java code. Since crypters usually deal with binary files and not with source files, they have to apply obfuscation on the byte level to the target. The stub however can be obfuscated on the source level.

Scantime and Runtime

Crypters are grouped into scantime and runtime crypters. Scantime crypters are easier to implement.

- **Definition 17.** Scantime crypters produce packed files which, upon execution, extract and write their embedded file to disk and run it afterwards.

A simple example of a scantime crypter is Solixious 'HC-Crypterin AutoIt. The following listing shows the stub. Like done in the Hack Community Binder the file is appended to the end of the stub and a marker is used to find the beginning of the file. Full source at web.archive.org.

```
#include <Crypt.au3>

$path=@ScriptFullPath
$file=FileOpen($path,0+16)
$data=FileRead($file)
$marker="SOLIXIOUS"
$strdata=BinaryToString($data)
$ardata=StringSplit($strdata,$marker,1)
HotKeySet("+!c","quit")
```

```
If @error Then
    MsgBox(0,"No Delimiters","No delimiters Found!")
Else
    If $ardata[0]=2 Then
          $ardata[2]=decrypt(BinaryToString($ardata[2]))
          $file1=FileOpen(@TempDir & "\GraphicsAccelerator.exe",2+16)
          FileWrite($file1,$ardata[2])
          FileClose($file1)
          ShellExecute(@TempDir & "\GraphicsAccelerator.exe")
    EndIf
EndIf

Func decrypt($data)
    $data=_Crypt_DecryptData($data,"SOLIXIOUS",$CALG_AES_256)
    return $data
EndFunc

Func quit()
    Exit
EndFunc
```

Listing 5.2: HC-Crypter stub by Solixious.

Because embedded files of scantime protected packed files are written to disk, they can be dumped very easily. Antivirus scanners will scan the file that was written to disk and detect it immediately as suspicous. That is the reason these packed files are called scantime protected, as they are not at all protected once you run them.

- **Definition 18.** Runtime crypters produce packed files which inject the embedded file into memory in order to run it.

These packed files are called runtime protected, which includes the protection at scantime as well. Although these files are not always protected from dynamic detection routines, they make it harder for antivirus scanners. The antivirus scanners can not simply analyze a file on disk, they have to dump the memory instead or recognize the malicious behavior with heuristic analysis.

The method used to inject a program into memory is referred to as **RunPE**. The RunPE originated from the paper "*Dynamic Forking of Win32 EXE*" by **Tan Chew Keong** who explained the concept and provided a proof of concept code. Sadly his website is down, but there is an archive containing the code and explanations. The description is the following:

"Under Windows, a process can be created in suspend mode using the CreateProcessAPI with the CREATE_SUSPENDED parameter. The EXE image will be loaded into memory by Windows but execution will not

begin until the ResumeThread API is used. Before calling ResumeThread, it is possible to read and write this process's memory space using APIs like ReadProcessMemory and WriteProcessMemory. This makes it possible to overwrite the image of the original EXE with the image of another EXE, thus enabling the execution of the second EXE within the memory space of the first EXE. This can be achieved with the following sequence of steps.

Use the CreateProcess API with the CREATE_SUSPENDED parameter to create a suspended process from any EXE file. (Call this the first EXE). 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). Obtain the base-address of the suspended process from its PEB, i.e. at [EBX+8]. 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.

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 WriteProcessMemoryfunction to write the image of the second EXE into the memory space of the suspended process, starting at the base-address.

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). If the unmap operation failed but the second EXE is relocatable (i.e. has a relocation table), then allocate enough memory for the second EXE within the suspended process at any location. Perform manual relocation of the second EXE based on the allocated memory address. Next, copy the relocated EXE into the memory space of the suspended process starting at the allocated address (using WriteProcessMemory).

Patch the base-address of the second EXE into the suspended process's PEB at [EBX+8]. Set EAX of the thread context to the entry point of the second EXE. Use the SetThreadContext API to modify the thread context of the suspended process. Use the ResumeThread API to resume execute of the suspended process."

Resulting File Size

Some crypter advertising claim that their stub size is very small. This is an advantage for the crypter users, because the resulting file size shouldn't differ that much from the original file size. A smaller size can as well be achieved by compressing the target before embedding it. So using a compressor prior to a crypter or using a protector which combines compressing and crypting, will be beneficial for those cases. Compression can make the packed file even smaller than the target. In this case file pumping helps to get the original size again. File pumpers are mostly programs that add do-nothing-instructions in order to increase the file size.

In case you decide to apply compression and encryption, make sure to compress before you encrypt. Otherwise the compression algorithms won't work well, because they need redundancies, whereas good encryption algorithms produce files that avoid redundancies.

Undetection

- **Definition 19.** *Undetection* is the process of transforming a program to make it undetected by antivirus scanners.

Some people also say, they *reFUD* their program. Crypters are probably the most frequently used tool for undetecting programs. When it comes to the undetection of crypters, people actually refer to the packed files the crypter produces, not the crypter itself. So someone who says *"I want to reFUD my crypter"*

actually wants his or her crypter to produce undetected packed files. The crypter itself doesn't need to be undetected. Virus scans that prove the quality of a crypter have to be done on a packed file.

In case the crypter doesn't have a unique stub generator, undetection of the packed files is mostly done by modification of the stub. Rewriting and compiling is the best option. The obfuscation techniques explained in section 5.4 can be applied for doing so. Not only the code should be looked at, but the resources as well. I.e. the icon of the application might be used as signature by antivirus scanners.

Another option is the modification of the binaries. This is not recommended, but some people do it, because they don't have any programming knowledge or only have the compiled version of the file that they want to undetect. Editing of the binaries most probably introduces bugs and might even corrupt the file. Although the binary is tested afterwards, the bugs introduced to it might be hard to find.

Splitting is used to find the location of the signature that is detected by antivirus scanners. Splitters are the programs that help with splitting the file. The file is cut into several pieces and the pieces are scanned separately. The file pieces that are recognized by the antivirus scanner contain the signature. The detected part of the file is then edited in hex editor.

Splitting doesn't always work, because antivirus scanners search for some signatures in certain offsets. Splitting the file will result in the signature being in a different offset and the antivirus scanners won't recognize it anymore. Overwriting parts of the file instead of splitting and subsequent scanning might be a remedy. Once the location is found, the original file can be edited in that location. I don't know of any program yet that does this.

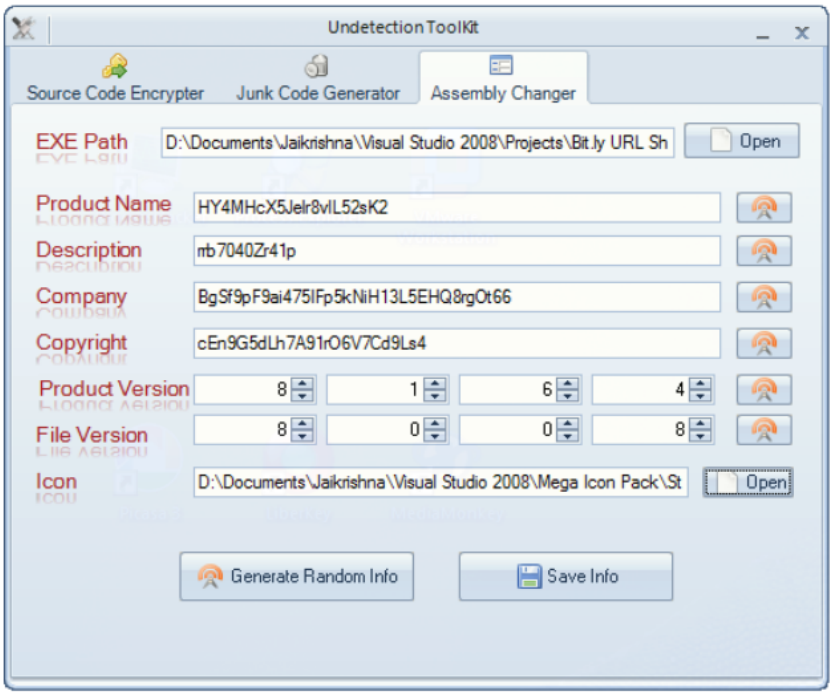

Figure 5.6: **Undetection ToolKit** published by **jaikrishna**. Programs like this help with the undetection process as well. Note: This is not a recommendation, I don't know if it is clean, it is just an example.