

Spring 2011

An Executable Packer

Neel Bavishi
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects



Part of the [Other Computer Sciences Commons](#), and the [Programming Languages and Compilers Commons](#)

Recommended Citation

Bavishi, Neel, "An Executable Packer" (2011). *Master's Projects*. 185.
DOI: <https://doi.org/10.31979/etd.2e78-fs4r>
https://scholarworks.sjsu.edu/etd_projects/185

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.



An Executable Packer

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Neel Bavishi

May 08, 2011

© 2011

Neel K. Bavishi

ALL RIGHTS RESERVED

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

AN EXECUTABLE PACKER

by

Neel Bavishi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Prof. Soon Tee Teoh, Department of Computer Science Date

Prof. Robert Chun, Department of Computer Science Date

Snehal Patel, Sr. Software Engineer, Yahoo! Date

ABSTRACT

A Packer to defeat Dynamic Un-packers

Neel Bavishi

This thesis addresses the topic of development and advancement of the Packer technology. It aims to prove that with the implementation of advanced code encryption and cryptographic techniques in conjunction with standard packing methods, testing binaries with anti-virus will become increasingly difficult.

Study on this topic reveals that the idea of encoding data has already been established, but it is still not fully incorporated into a technique to pack an executable file. There are some noticeable defects as un-packer tools have also made a great advancement in the field of dynamic analysis. The addition of new capability to recognize emulation environment and taint analysis has lead to execution-time detections of malware.

The plan is to develop a proof of concept that proves that the dynamic un-packers like Renovo can be defeated. The prototype will try to pack and compress the binary file in such a way that it can easily evade the emulation environment created by anti-viruses.

Contents

List of Figures:	2
List of Code Sections:	2
1.0 INTRODUCTION	3
2.0 BACKGROUND	7
2.1 PE File Structure	7
2.1.1 PE File Sections	10
2.2 How does a packer work?	13
2.3 How does a dynamic un-packer work?	13
2.4 Dynamic Un-packer example Renovo	14
3.0 PREVIOUSLY USED TECHNIQUES TO EVADE DYNAMIC UN-PACKERS	17
3.1 Circumventing the emulated environment	17
3.2 Exploiting the time-out	23
3.3 Dual Mapping Physical Pages	23
4.0 PROJECT OVERVIEW	24
4.1 Packer stages	25
5.0 COMPRESSION TECHNIQUE USED	28
5.1 LZ Compress:	29
5.2 Huffman coding	36
5.3 Generic Compressor Class:	41
6.0 PE File Protection:	43
7.0 TESTS	50
8.0 CONCLUSION:	54
9.0 REFERENCES	55

List of Figures:

Figure 1: PE Structure [9]	9
Figure 2: PE Explorer results for Putty.exe	12
Figure 3: Working of a packed Executable [1]	15
Figure 4: Output for program to get the system information	22
Figure 5: Error caused while running executable under virtual environment.....	23
Figure 6: Packer Block Diagram	24
Figure 7: Lempel-Ziv family of Algorithms [7]	29
Figure 8: Buffers used in LZ Compression Algorithm [11]	30
Figure 9: Contents of u-buffer [11]	31
Figure 10: Contents of v-buffer [11]	31
Figure 11: Flowchart of LZCompress.....	33
Figure 12: Adding new section	44

List of Code Sections:

Code Section 1: Small Red Pill test code snippet	19
Code Section 2: Determine host specific process	20
Code Section 3: Obtain System information.....	21
Code Section 4: Implementation of LZ Compress.....	34
Code Section 5: Implementation of find the match function	35
Code Section 6: Huffman Coding build tree function	41
Code Section 7: Our main Compressor Class	41
Code Section 8: Go through all compressors.....	42
Code Section 9: Insert new section in a PE file	45
Code Section 10: code snippet for rerouting procedure in every IJT entry	46
Code Section 11: Simple Encryption for all the sections.....	47
Code Section 12: SoftIce detection.....	48

1. INTRODUCTION

“**PE compression** is a way of shrinking a PE file and merging the packed data with restoration code into a file” [3]. A Packer is a utility which implements compression and some encryption on an executable to make it undetectable by un-packers/virus scanners.

Executing a packed executable necessarily un-wraps genuine code and it is then handed over the control. It has the same effect as that of running the unaffected PE file. The effect it displays is the same as the original executable was running. This means that it is impossible for a normal user to differentiate between compressed and uncompressed PE files.

A condensed executable is a type of self-extracting archive, wherein packed data is wrapped up along with the pertinent restoration routine in a PE file. There are also tools which only decompress an executable without actually running it. For example, programs like ZIP and RAR.

Packed files usually decompress directly into the memory without needing any file system space to execute. However, some de-compressor stubs write such PE to the file system to execute. [3]

Packing a file has its advantages and some disadvantages. We will focus here on some of its disadvantages as it is our aim to protect the hidden malware in a binary.

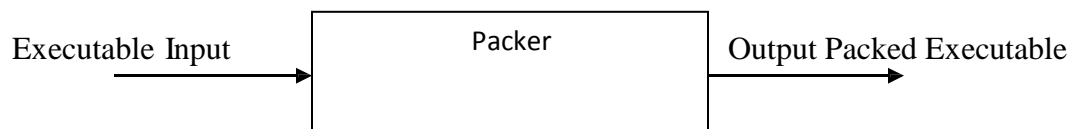
PE file compression often dissuades process of reverse engineering. In some cases, it might do some obfuscation of the executable contents by using methods shrinking and/or added encoding. Executable compression prevents straight disassembly to some extent; it masks the string literal and alters the autographs. This never suggests that the file cannot be reverse engineered; it is just that the procedure is now more expensive. “In addition, it becomes impossible for some utilities to recognize run-time library reliance because only the extractor stub which is statically connected is visible.” [3]

Again, certain of age infection scanners mark all compressed PE files as viruses leading to false alarms. This is due to similarity in some characteristics to those of de-compressor stubs. Most infection scanners usually take out numerous PE compression layers to check for the original file inside, but again certain popular anti-viruses have problem in piercing through such layers.

Therefore, packers have a big say in PE protection.

My Packer tool:

This thesis aims at creating such fully undetectable packer tool to protect executables from anti-virus scanners as well as debuggers. To provide a complete protection to an executable file, this tool has a basic built in flow which goes through various cycles of compression and file protection.



As shown in the above diagram, the input to this packer tool is an executable file and the output is a packed executable.

Packer consists of three different stages:

Compression:

Executable data compression is the first stage in packer tool. In this stage, the data is compressed and the decompressor stub is packed into the packed executable along with the compressed data. This compression happens over two steps as follows:

- 1.) LZCompress: In this step all the repetitive byte sequences are removed
- 2.) Huffman Coding: Prefix code is used to compress data

File Protection:

Executable file protection involves many different stages to make it undetectable.

They include;

- 1.) Modifying PE File Structure: Adding a new security section
- 2.) Modify import table values: Mainly related to modifying the Import table address
- 3.) Static Code redirection: Try to alter flow of a normal program by inserting jump statements.
- 4.) File Encryption: Simple XOR Encryption for all the sections

Anti-Debugging Techniques:

There are mainly three anti-debugging techniques used in this tool:

- 1.) Insert lot of junk code
- 2.) IsDebugger present to detect presence of executable debugger
- 3.) SoftIce detection: detect presence of SoftIce debugger and disassembler

These packer stages will be covered in detail in the following sections.

2. BACKGROUND

A program can be transformed into a packaged executable using *code packing* by condensing and encoding the original code and data into packed data. This can eventually be linked with a *restoration routine*. A *restoration routine* is a code snippet that can be used for recovering original code and data. Along with this kind of recovery, it can also set an execution context to the original code when the packed program is executed.[1]

2.1 PE File Structure

PE file format was introduced by Microsoft as a part of Win32 specifications. However, they hold their base in earlier used COFF format used on VAX or VMS. The term PE which stands for Portable Executable was chosen to intend a common file format across all Windows platforms.

The introduction of 64-bit Windows needed very less modifications in executable format. It is thus known as PE-32+.It just required deletion of one field and spreading of some fields from 32-bit to 64-bit. In almost all the cases, this code works for 32-bit as well as 64-bit systems. “There is magic pixie dust in Windows header file. It creates the differences which are not visible to most C++ code base “ [9].

The EXE and DLL files use the same PE format. The entire distinction between

both the types is just of one semantic. The semantic is only a bit which specifies whether the file is an EXE or as a DLL. DLL extension is also defined by a user. DLLs might have dissimilar extensions. For example, .OCX and .CPL are kinds of DLL. [9]

A standard feature of PE files is the similarity of data structures on disk to those used in the memory. They are equal. Therefore, loading a PE file into memory just maps some ranges of an executable into the address space. “Thus, a data structure like the IMAGE_NT_HEADERS on the disk is similar to that in the memory”. [9]

PE files structures are mapped into memory as a memory-mapped file with multiple units/sections. Windows loader decides what essential sections of the PE file should be mapped. This mapping is unswerving in the sense that upper offsets in the file should relate to upper memory addresses when they are planted into memory. It is not necessary that offset in the item when on the disk, is same as when it is mapped into the memory. However, the information needed during transformation from disk to memory offset is present.

Below figure gives the PE structure [9]:

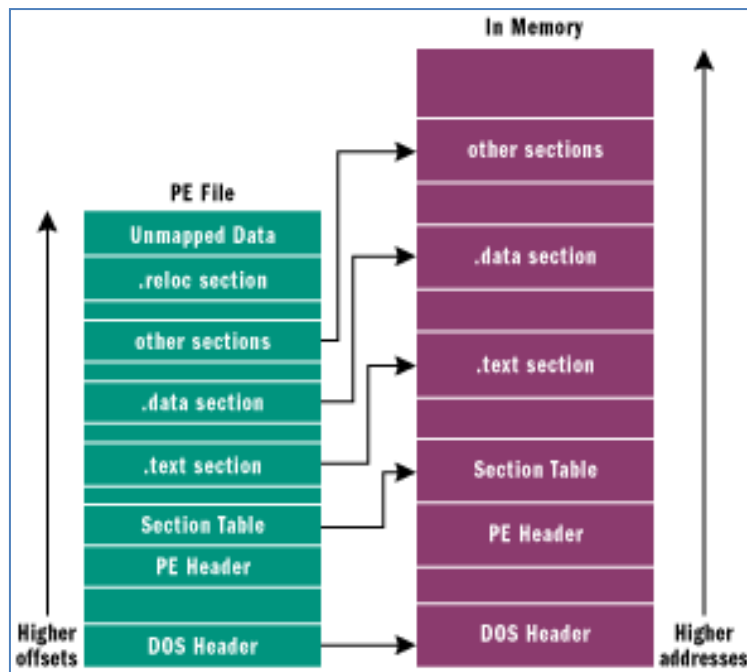


Figure 1: PE Structure [9]

Windows loader loads the executable file into the memory. This in-memory version of the file is called a module. The beginning address of the module is known as an HMODULE. So a point worth noting is that given a HMODULE, you can expect a particular data structure at a given address. This information can help to locate remaining data structures in the memory. This ability can be exploited for API interception as well. [9]

A module in memory is a representation of all the data, resources and code from a portable executable which a process uses. Additional parts of a PE file are possibly read for instance and relocations, but not necessarily mapped in. Some parts are not planted in at all. For eg, end of the file having debug information. PE header

contains a field which informs the system about the memory required to be put aside for mapping the PE file into memory. Unmapped data is appended to the end of the file, after mapping required information.

The PE format is defined in WINNT.H. This header file nearly contains all enumerations, definitions; structures and #defines used with PE files in memory.

2.1.1 PE File Sections

A PE file section contains some kind of code or data. Talking about data, they are of multiple types while code is just code. Data can be of the type read/write program data in form of global variables. In addition, sections contain data which include API import, the exported tables, relocations, resources, etc. Every section contains own set of in-memory traits, such as whether it includes some code, it just contains read-only or read/write data or the data which is shared by all the processes using the portable executable.

In general sense, in a section, all the code/data is somehow related. There are at least two sections in a portable executable file. Both code and data occupy each of them. Usually, a PE file contains at minimum one more kind of data section. Each section of memory has different features which reflect its usage: readable, executable, writable and other more specialized operations.

Each section has a unique title/name. This name should express the utility of the

section. A section “.rdata” represents a read-only data section. Section names are utilized just for human understands, while operating systems have nothing to do with it. A section called “.abcd” is also valid just like a “.text” section. Microsoft formats generally have a period as a prefix to section names, but it is not a necessity. Borland linker names its section as CODE and or DATA. Sections can be created and named; it is the linker which includes them in the executable. A VC++ compiler inserts code or data into a user named section using #pragma statements. For example, the following statement

```
#pragma data_seg( "NEW_CODE" )
```

inserts all VC++ emitted data into a section called NEW_CODE, and not into the .data section [9]. Most programs use the default compiler emitted sections, but occasionally programmers might have a requirement which requires creating new sections to put data in.

The commonly named sections in a PE file are classified as following:

.text: Main Code snippet Main responsible for execution and is mostly read-only.

.data: Code snippet responsible for main data initialization.

.rsrc: Comprises of data associated with Windows Resources.

.rdata: Read-only data.

.reloc: Base relocations.

.debug: Comprises of information responsible for debugging.

.idata: Imported function data.

.tls: Thread Local Storage. Data private to each thread

.CRTData: Set aside for the 'C' Run-Time Library

PE explorer can list all the sections from any DLL or EXE file along with numerous other attributes that are read from PE File Header.

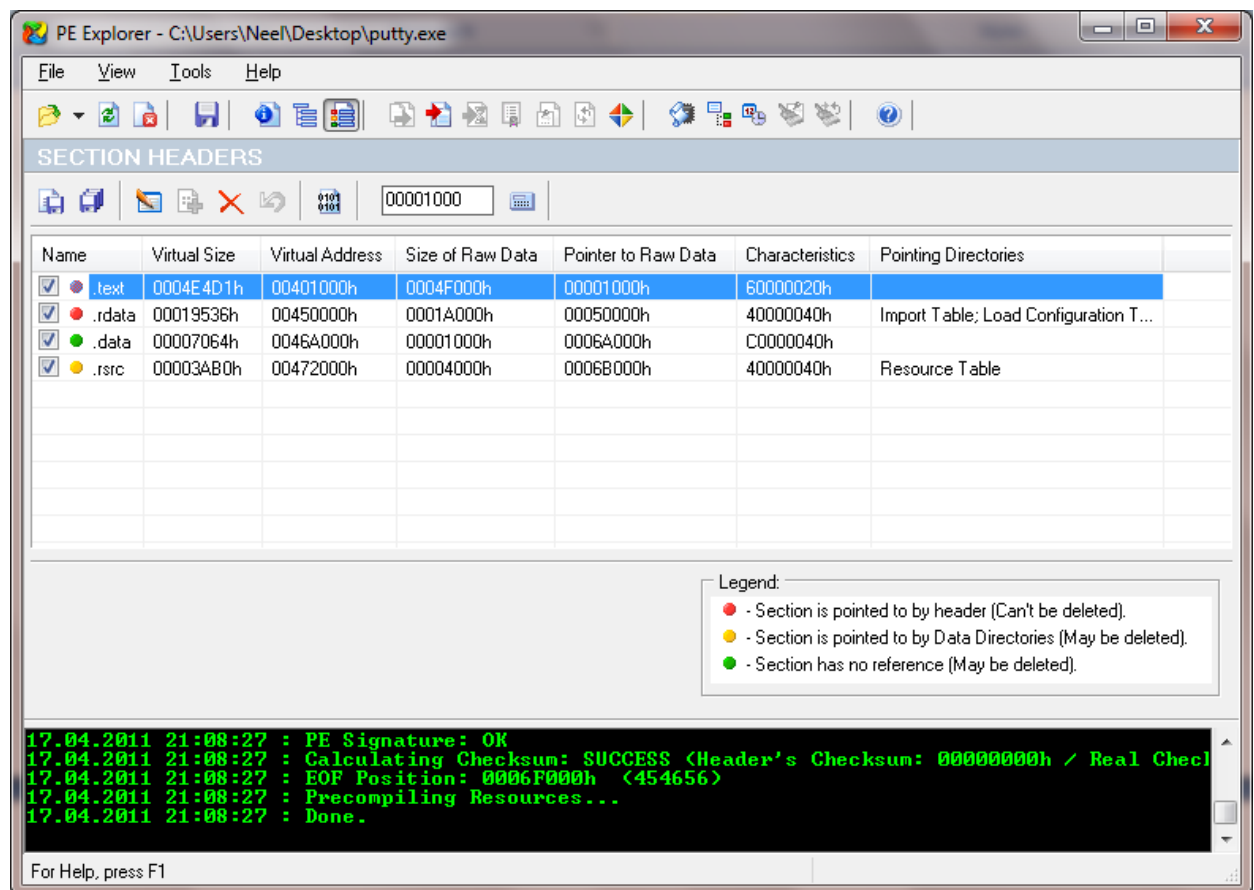


Figure 2: PE Explorer results for Putty.exe

2.2 How does a packer work?

Packing an executable means compressing and encrypting it such that for a user it is impossible to distinguish between a packed executable and an unpacked executable. The packed executable includes packed data as well as the de-compression and decrypting routine.

It works as follows:

Given an arbitrary executable binary, pack the data using compression and encryption routine. Check if the real program code created from the packed data in the file is executed. Extract the whole new-generated code and data with its OEP (Original Entry Point) address. When the packed PE is executed, its bound rebuilding routine performs various alteration actions on the compressed and encrypted data to recuperate the original code and data. After the restoration completion, the execution context for the original program code to execute is prepared. This includes initializing CPU registers and assigning the program counter to the entry point of the newly-generated code region [2].

2.3 How does a dynamic un-packer work?

The primary motive of a un-packer is to un-wrap the hidden exe and the algorithm used to hide it without executing it on a host system.

It works as follows:

Irrespective of packing procedures used or number of covered layers applied, the genuine program and the data available in the memory runs, and in addition the instruction pointer jumps to the original entry point of the re-obtained code which was written in the memory at runtime [1]. Using this disadvantage of such fundamental nature of packed executable, the un-packer uses a technique that on runtime extracts the coveted real code and the original entry point from the wrapped up executable by checking if the current instruction is formed at runtime [1]. In this approach, the instruction pointer makes a jump to the monitored memory region that was written to after the program started. When program loads in the memory, it generates a memory map which is initialized as clean. Whenever, a memory write operation is performed, for example `mov ead,[edx]` and `push edi`, we blot the respective terminus of the memory location as dirty, meaning it is recently generated [1]. Now, the address pointed by the memory pointer is the OEP.

2.4 Dynamic Un-packer example Renovo

RENOVO resides on TEMU [13] platform, a dynamic investigation emulation software from Bit Blaze. Executable is first executed in an emulation environment. This emulation isolates the extraction engine from the harmful program code. Hence, malicious code's interference with the extraction engine is difficult and does not affect analysis of the results. For our analysis, we need to know which processes should be observed. TEMU provides mechanism to reason about this OS-level

semantics. Theoretically, a kernel module is inserted into the emulated environment to obtain necessary process data. Hence, the module will be notified whenever a process is created or destroyed, or a module (.dll or .exe) is loaded into the process. In 32-bit systems, the physical address of the page table for the current process is stored in CR3 register which makes it unique for each process [1]. After the program starts to execute, identifying the loaded module leads to knowledge of memory region it occupies and the states within the region are cleaned. To know whether program has a hidden executable a timeout mechanism is executed.

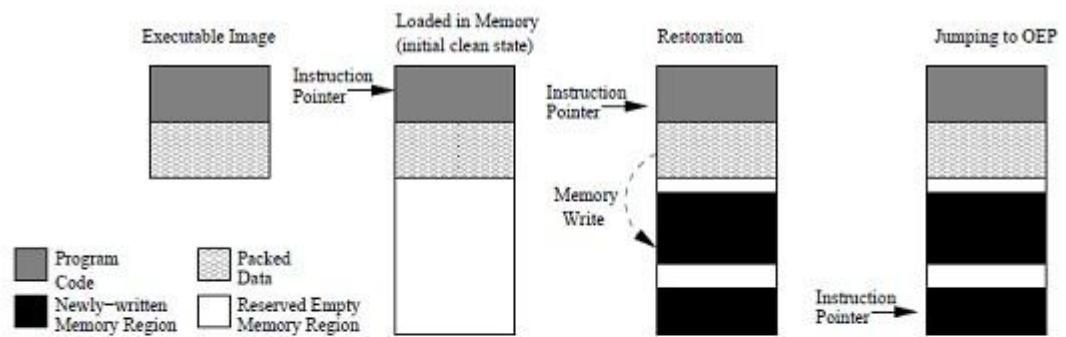


Figure 1: How a packed executable works.

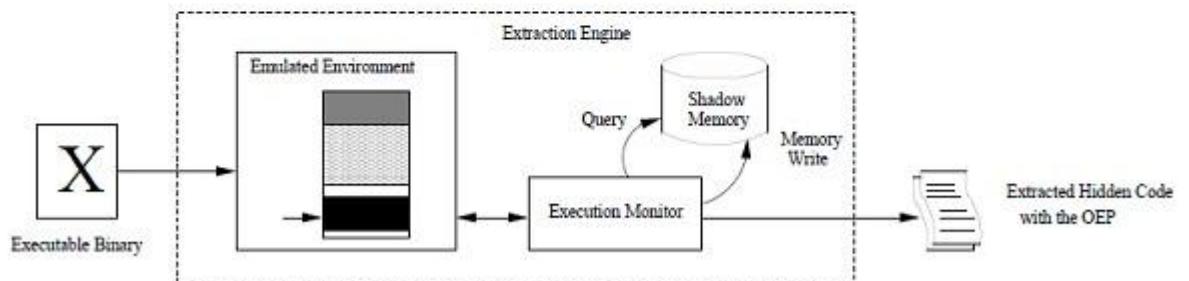


Figure 3: Working of a packed Executable [1]

When checking newly generated instructions, every instruction is not checked. For performance, optimization, every fundamental block in the monitored process is verified. A fundamental block is defined as sequential instructions with a unique entry and exit. Thus, a fundamental block is nothing but a neighboring code region. The address is recorded at the entry of the block. Eventually, at the exit of the block, memory locations, which are marked as dirty within the region covering this block, are verified whether they exist or not. If they exist then this block entry is the unique point of entry and the pages with dirty memory bytes are dumped.

3. PREVIOUSLY USED TECHNIQUES TO EVADE DYNAMIC UN-PACKERS

Various techniques are proposed to evade dynamic analysis by un-packers:

Important among those are explained as follows:

3.1 Circumventing the emulated environment

As the binaries are executed in emulated environment in un-packer, one obvious evasion technique that comes to mind is to detect the presence of emulated environment and stay inactive. The malicious code measures time elapsed for certain instructions, as the emulation of such instructions incurs high overhead. This code may also verify the instructions results such as sidt, because they produce diverse results under physical and emulated environments. This can be shown by performing the RedPill test as shown below.

3.1.1 Red Pill Test

Red Pill test [6] discovered by Joanna Rutkowska in Nov 2004, is a way to detect VMM (Virtual Machine Manager) using a processor instruction. “The important part of this test is the SIDT instruction which contains the contents of the IDTR in the destination operand. This operand represents memory location. The interesting characteristic of SIDT instruction is that though it is executed in the user mode, the

contents of the IDTR are returned. IDTR is used internally by operating system”.
[6]

On any system, there is only one IDT register, but there could be multiple OS running concurrently, like guest and host. Therefore, the guest's IDTR should be relocated in a safe place by VMM (Virtual Memory Manager) to keep it safe from host one. Unfortunately, it is not possible for VMM to know when the process running in guest OS executes SIDT instruction because of its limited privileges. Thus, the relocated address of IDT is obtained by the process. The relocation address of IDT on VMWare is detected as 0xffXXXXXX, whereas on Virtual PC it is 0xe8XXXXXX [4].

Short background on SIDT:

SIDT stands for “Store Interrupt Descriptor Table Register”. Its Opcode is 0F 01/1.

General use: SIDT m -> Store IDTR to m.

“The destination operand indicates a 6-byte memory location. If the operand-size attribute is 32 bits, the 16-bit limit field of the register is stored in the lower 2 bytes of the memory location and the 32-bit base address is stored in the upper 4 bytes. If the operand-size attribute is 16 bits, the limit is stored in the lower 2 bytes and the 24-bit base address is stored in the third, fourth, and fifth byte, with the sixth byte filled with 0s.” [19]

```

#include <stdio.h>
int main ( ) {
    unsigned char m[2+4], rpill[ ] =
"\x0f\x01\x0d\x00\x00\x00\x00\xc3";
    // rpill[] contains SIDT instruction.
    *((unsigned*)&rpill[3]) = (unsigned)m;
    ((void(*)())&rpill)();

    printf ("idt base: %#x\n", *((unsigned*)&m[2]));
    if (m[5]>0xd0) printf ("Inside VM!\n", m[5]);
    else printf ("Not in VM!\n");
    return 0;
}

```

Code Section 1: Small Red Pill test code snippet

Host with no VMM: Not in Matrix. (Not in VM)

Host with VMM, but no VMM is running: Not in VM

Host with VMM, VMM running: In VM (Supposedly in a VM)

Guest in the above host: In VM.

Flaw: This test is inadequate because, in multicore CPUs the process execution takes place in different processors every time. The main problem caused is that every time the IDT address will change, same problem will be faced while checking LDT and GDT tables.

3.1.2 Determine the Host Specific process

For example, **vmusrvc** for Virtual Private Client from Microsoft is a process which runs when we can identify while running our application in VPC. Similarly, there is a process called **VBox Service** for Virtual Box from Sun.

```
private static bool DetectVPC()
{
    if (Process.GetProcessesByName("vpcmap").Length >= 1 &
        Process.GetProcessesByName("vmusrvc").Length >= 1)
        return true;

    else

        if (Process.GetProcessesByName("vmusrvc").Length >= 1)
            return true;

        return false;
}
```

Code Section 2: Determine host specific process

Flaw: The software emulators like QEMU being a full system emulator; it is difficult to identify any such processes being executed to identify the execution unit.

3.1.3 Read Bios information

This technique was used by a security group in a company named Sysinternals which was later brought by Microsoft. This is one of the easiest and most effective techniques used to identify whether a binary is executing in an emulated environment. If it is executed in a virtual machine, it will give an error and stop executing.

```
SYSTEM_INFO siSysInfo;

    // Copy the hardware information to the SYSTEM_INFO
    structure.

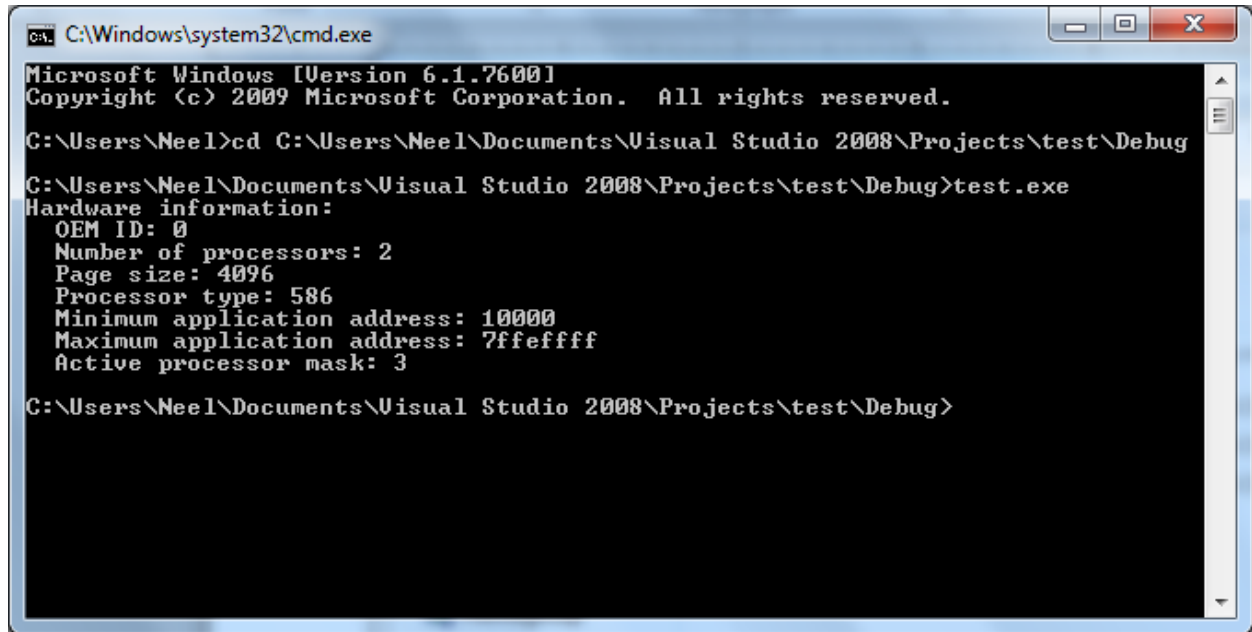
    GetSystemInfo(&siSysInfo);

    // Display the contents of the SYSTEM_INFO structure.

    printf("Hardware information: \n");
    printf(" OEM ID: %u\n", siSysInfo.dwOemId);
    printf(" Number of processors: %u\n",
    siSysInfo.dwNumberOfProcessors);
    printf(" Page size: %u\n", siSysInfo.dwPageSize);
    printf(" Processor type: %u\n", siSysInfo.dwProcessorType);
    printf(" Minimum application address: %lx\n",
    siSysInfo.lpMinimumApplicationAddress);
    printf(" Maximum application address: %lx\n",
    siSysInfo.lpMaximumApplicationAddress);
    printf(" Active processor mask: %u\n",
    siSysInfo.dwActiveProcessorMask);
```

Code Section 3: Obtain System information

The above code snippet runs correctly on real system. It prints the hardware information that is read from RMBIOS by making system calls. Basically, it copies hardware information into system_info structure.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Neel>cd C:\Users\Neel\Documents\Visual Studio 2008\Projects\test\Debug
C:\Users\Neel\Documents\Visual Studio 2008\Projects\test\Debug>test.exe
Hardware information:
  OEM ID: 0
  Number of processors: 2
  Page size: 4096
  Processor type: 586
  Minimum application address: 10000
  Maximum application address: 7ffeffff
  Active processor mask: 3

C:\Users\Neel\Documents\Visual Studio 2008\Projects\test\Debug>
```

Figure 4: Output for program to get the system information

Now, when the exe is executed in an emulated environment like QEMU or VPC it gives an error straight away as no hardware information can be obtained. The error is as shown below:

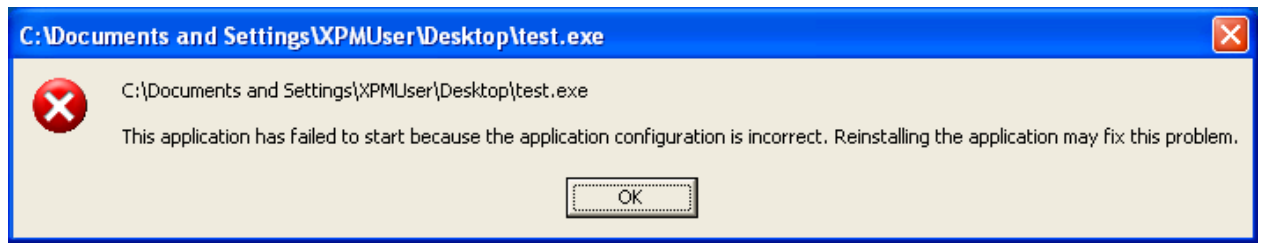


Figure 5: Error caused while running executable under virtual environment

3.2 Exploiting the time-out

Time-out [4] is an interesting problem discussed in Security Application Conference in Washington DC, in 2006. As we know, determining whether a PE contains hidden code or not is an un-decidable problem, for which a time-out mechanism is usually employed. The malicious programs use this technique regularly to exploit such feature to remain inactive for long period leading to incorrect results by the un-packer. To counter this exploitation, un-packers use an improved metric which determines the termination of the extraction procedure by counting the number of different instructions from the binary that execute. This means that these malicious codes cannot avoid detection by merely looping around.

[1]

3.3 Dual Mapping Physical Pages

Another approach is dual-mapping. [5] This approach was first used by H Miller.” According to this approach, physical pages are mapped to two distinct virtual address regions. The first region is provided for editable mapping to write during

unpacking process whereas the second region is provided for executable mapping which dynamically executes the unpacked code. Thus, this approach is effective in evading automated un-packers, which solely depend on perceiving the virtual addresses code execution that it has been written to.” [5]

4. PROJECT OVERVIEW

The aim of this thesis project is to make a tool which protects the executable or PE in question from being recognized by anti-virus scanners as threats. Fig 6 gives the block diagram of this tool.

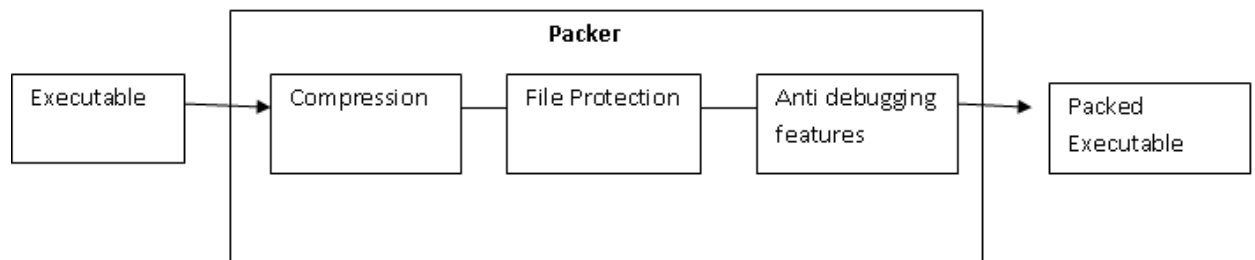


Figure 6: Packer Block Diagram

This tool accepts a PE (portable executable) file as input. In the packer, it goes through various transformations as shown in the block diagram to output a packed executable.

4.1 Packer stages

Packer has 3 built in stages in form of compression, file protection and anti-debugging.

Compression:

This tool has two built in compressor which have base in open source LZIB compression library:

1.) LZCompress: LZCompress is a lossless type of data compression. It is most effective when there are repetitions in byte sequences. It has its base in LZ77 algorithm from Lempel Ziv algorithms family. In this type of compression algorithm, there are two types of buffer, one which contains processed bytes, u-buffer and other which contains bytes to processed, v-buffer. While filling in the v-buffer we check for the similar byte sequences in u-buffer. If a sequence is obtained, we save the location in the v-buffer instead of holding the entire buffer content. This helps to reduce a lot of size. It is explained in detail in next section.

2.) Huffman Coding: It is a form of prefix code. Each value is represented by a sequences of codes, either 0 or 1. The values which are repeated frequently, for example vowels in a sentence, are given shorter codes. This again reduces the executable file size. Its implementation is covered in the next section.

File Protection:

File protection is a very important stage of this packer tool. An executable/PE file goes through various Static protection procedures and dynamic redirection process to prevent against reverse engineering. Some of them can be listed as follows:

- 1.) Modify PE File Structure
- 2.) Static Code Redirection
- 3.) PE File Encryption
- 4.) Modify Import Table
- 5.) Dynamic Code redirection

Anti-Debugging Techniques:

In this tool there three main anti-debugging techniques used

- 1.) Insertion of Junk Code
- 2.) Is-Debugger present
- 3.) SoftIce detection

Apart from this, static code redirection and dynamic code direction causes lot of problems in code debugging due to changes in normal execution routines.

This part is covered in detail in Section 6.

5. COMPRESSION TECHNIQUE USED

Data compression is an important stage of this packer tool. Compression is possible due to redundancy in data. It is basically the technique to encode data such that it requires less storage space or transmission time than it would take without being compressed.

There are mainly 2 types of data compression techniques:

1. **Lossless Compression:** Used in spreadsheets, text, executable program Compression.
2. **Lossy less Compression:** Compression of images, movies and sounds.

As, this tool is about executable compression, it uses Lossless Compression.

In this tool two different types of compression techniques are used. One is the variant of Lempel Ziv Algorithms family [18] and other uses Huffman coding [17]. They have their roots in the two algorithms proposed by Jacob Ziv and Abraham Lempel. These Compression algorithms are mainly divided in two main groups: LZ77 and LZ78 [16]. The clear difference between the two groups is that LZ77 do not need an explicit dictionary where LZ78 does need it.

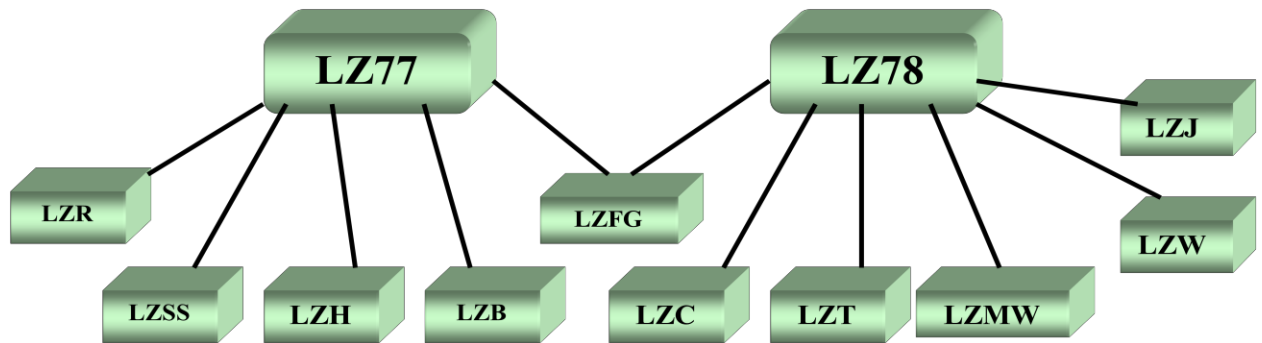


Figure 7: Lempel-Ziv family of Algorithms [7]

There are two compressors built in this tool. They are executed one after the other.

The two of them are:

1. LZ Compress
2. Huffman Coding

Small executable files go through only one step where as larger ones need the execution of second step.

5.1 LZ Compress:

In LZ Compression algorithm, the program stores W previous bytes and scans the next few bytes such that they are repetition of the previous stored data. In case of the match found,

then the length of bits and location are recorded and stored. This requires fewer bytes and hence less amount of memory space which leads to compression. [14]

This algorithm uses two buffers. First buffer provides encoded strings that were previously encoded. This “previously encoded buffer” is denoted as “u”. The second buffer provides the “to be compressed” strings. This “to be encoded buffer” is denoted as “v”. Last symbol in the v- buffer is excluded because an extension symbol should always be kept un-encoded.

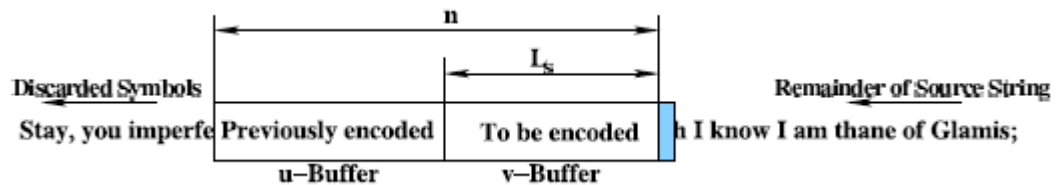


Figure 8: Buffers used in LZ Compression Algorithm [11]

Flowchart (Fig.11) shows the vital parts of this algorithm. In the beginniing, the u- buffer is initialized with a value. Now, v-buffer is parsed to find the longest possible match. During the first iteration, unless the string in contention to be condensed starts with spaces, the parser cannot search a match.

A code word of the form $\langle p, |u|, q \rangle$ is formed. p indicates the position from where the match starts in the u-buffer, and $|u|$ is the extension character. The extension character q is the subsequent character read in this string and has to be encrypted after if finds a match. Sometimes, the matches can reach into the v-buffer.

Suppose a String for compression:

“Fair is foul, and foul is fair: Hover through the fog and filthy air.” [11]

E.g. assuming the u-buffer already contains:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F	a	i	r	␣	i	s	␣	f	o	u	l	,	␣	a

Figure 9: Contents of u-buffer [11]

And the v-buffer contains:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
n	d	␣	f	o	u	l	␣	i	s	␣	f	a	i	r

Figure 10: Contents of v-buffer [11]

u-Buffer															v-Buffer															< p, μ , σ >
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
f	a	i	r	␣	i	s	␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r	␣	<0, 0, n>
a	i	r	␣	i	s	␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r	␣		<0, 0, d>
i	r	␣	i	s	␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r	␣			<6, 4, ␣>
␣	f	o	u	l	,	␣	a	n	d	f	o	u	l	␣	i	s	␣	f	a	i	r	␣								...

Table 1: Contents of u- and v-Buffer during Compression [11]

Algorithm:

Basic Algorithm for LZ Compress for the above example is as follows:

Step1: Parse the buffer to find the longest match → the next coming character is n which does not appear in the u-buffer.

Step2: This makes code word to be $\langle 0, 0, n \rangle$, n being the next incoming character scanned.

Step3: Now, shift the u-buffer and v-buffer by 1 character as shown in the table.

Step4: Parse the buffer to find the longest match → the next coming character is d which does not appear in the u-buffer.

Step5: This makes code word to be $\langle 0, 0, d \rangle$, d being the next incoming character scanned.

Step6: Again, shift the u-buffer and v-buffer by 1 character as shown in the table.

Step7: While the third iteration executes, the lengthiest match is detected in form of the string “foul”.

Step8: Hence, the code word obtained is $\langle 6, 4, t \rangle$.

Step9: Both u- and v- buffers are moved by 5 characters towards left, that is length of the string foul and the extension characters.

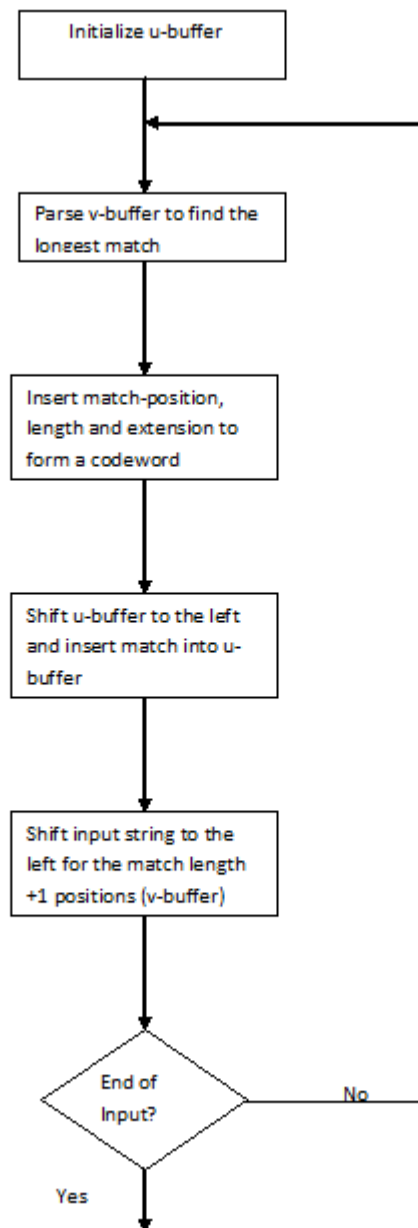


Figure 11: Flowchart of LZCompress

In this algorithm, the input bytes are stored in an array. The bytes processed already are stored from `a[0]` to `a[Q-1]` i.e our u-buffer and the bytes that are to be processed are stored from `a[Q]` to `a[Q+U-1]` which is same as stored in `b[0]` to `b[U-1]` i.e. our v-buffer. Therefore, the bytes in `b[0]`, `b[1]`, `b[2]` . . `b[U-1]` are compared with previous bytes to find a match.

```

after = before = 0;
for(i = 0; i<Q; i++) a[i] = ' '; /* initialize a[] as empty */
b = a+Q; /* b points to unprocessed U data bytes */
m = 0;
for(;;) {
    while(m<U && (c = getc(ifp))!=EOF) {
        b[m++] = c;
        before++;
    }
    if(m==0) break; /* all data is processed */
    s = find(&r);
    if(debug) dbprint(r, s);
    writelz(s, r, ofp); /* write one pair of bytes */
    after += 2;
    m -= s;
    for(i = 0; i<Q+U-s; i++) a[i] = a[i+s]; /* shift a[] left s
places.*/
}

```

Code Section 4: Implementation of LZ Compress

In this program, “`a[]`” stores the latest 4096 processed bytes read from the file in question and plus 16 unprocessed bytes. The 16 unprocessed bytes are referred to as “`b[]`” in the above program segment. The algorithm tries to find a copy of bytes in array `b` starting from `b[0]` to as many as possible bytes in previously processed data `a[]`. If at least two matching bytes are found, then “`s`” indicates the number of bytes matched and “`r`” gives the position for match occurrence in `a[]` so that, the first byte matched is `a[r]`. If no match

is found, at least b[0] and b[1] are tied together in a[], s and m are set to the value of 1 and b[0] respectively, which indicates an input character. [14]

Find function that tries to find actual match:

```
int find(int *y)
{
    int i, p, q;
    *y = 0;
    q = 1;
    for(i = 0; i<Q; i++) { /* search for pattern at b[0] in
a[p] */
        for(k = 0; p<m && a[p+i]==b[p]; p++);
        if(p>q) { q = p; *y = i; } /* save longest match
found */
    }
    if(q==1) *r = b[0]; /* if q=1 then y is the first
character */
    return q;
}
```

Code Section 5: Implementation of find the match function

The actual searching happens in the find function. The for loop "for(i = 0; i<Q; i++). . ." initializes the search process in each position in the processed data. The actual comparison happens in the statement "for(p = 0; p<m && a[p+i]==b[p]; p++);" This loop just repeats until the end of where the match is found in the two arrays that are in contention. On completion, "p" indicates the number of places that match and "i" gives the location of the match. If the match found is longer than the best one found before, p and i are taken as the new values of s and r. At the end of

for loop, the length and location of the longest match in s and r respectively are obtained.

5.2 Huffman coding

Huffman coding [17] discovered by Ken Huffman, is a form of prefix coding, which is knowingly or unknowingly are used in common. One example of prefix coding used in phone is Huffman coding. The order of keys pressed may be a sequence of any key number combination -- and each order pressed represents a different definite phone number.

Suppose that you are in a workplace environment with all the employees having their allotted phone numbers in the office. For internal communication in most of the companies you don't need to dial full number. It is just last four digits and a prefix digit „9“. This digit is known as the prefix digit in Huffman coding. Each element specified has a unique code created by numbers, and because each name begins with a unique code, there will be no ambiguity that each code when you enter will be exactly what you wanted. [12]

A Huffman code is a form of prefix code dealing with bits. Here, codes are made up of a sequence of bits that may be 0 or 1 in place of a series of decimal numbers from 0 to 9. Each code represents a series of alphabets. This is the main use of Huffman coding in Deflation algorithm.

In Huffman algorithm, firstly all the alphabets are assembled. Each alphabet is then assigned a “weight” – Weight is the frequency of letters in the data to be packed. Such weights may be decided earlier, or stated from parsing the data, or some permutation-combination of both. Two elements are chosen at a time in any case and the one with minimum weight is selected first. The two elements are made to be leaf nodes of a node with two branches. Let us see an example with weights given as below:

```
A 16
B 32
C 32
D 8
E 8
```

D and E are picked first as they have lowest weight. A node is branched into these two elements -- one being the '0' branch and the other being '1'.

```
( )
0 / \ 1
D   E
```

In this situation, complete code for any element cannot be known, but it is at least clear that D and E have equal codes, other than the last binary digit where D ends in 0 whereas E in 1.

The joined node D-and-E is positioned back into the pool of elements which are not combined, with the weight obtained from the sum of its leaf nodes: for example, $8 + 8 = 16$ in this case. Now, the two nodes with lowest weight taken are A, and D-and-E combined, and they form a large node.

```

    ()
  0 / \ 1
  () A
0 / \ 1
D  E

```

Again, the node A-D-E is re-added to the original set of elements. But this time around, all outstanding elements have the same value of 32. So there is confusion in which two to select first for the combination. But it is actually not important in Huffman algorithm.

Finally we get a complete Huffman tree wherein we can reach any element from root selecting proper 0 or 1 branch. Thus, each element traversal can be done with the order of 0's and 1's. This is known as Huffman code for those elements, that symbolizes the pathway through the tree.

Now, it can be visualized that such a tree, and mere a set of codes, provide a way for executable compression. During compression of ordinary text, probably 50% of the ASCII characters could be omitted from the tree completely. Commonly utilized

characters, like all the vowels or some letters like “T” will perhaps get quite smaller codes and the long codes will be used the least.

It is also fairly simple to pass encrypted data along with the tree and can be coded by slightly altering the algorithm which generates the tree.

So how is Deflate different from classic Huffman coding? In the case of classic, multiple trees could be generated using a single set of elements and weights, whereas the Deflate variation uses two supplementary rules: elements with the shorter codes are positioned on the left side and the longer codes on right side. If codes have the same length, then the first in the element set are positioned on the left.

Thus, if these two restrictions are applied on the trees, there is a unique tree generated for every set of elements and their respective code lengths. These code lengths will help in reconstruction of the tree.

```

void En_Decode::BuildHufTree()
{
    int NodeCounter = 256;
    int i;

    for (i = 0; i < NodeCounter; i++)
    {
        OurTree[i].parent = -1;
        OurTree[i].right = -1;
        OurTree[i].left = -1;

        while (1)

            int MinFreq0 = -1;
            int MinFreq1 = -1;

            for (i = 0; i < NodeCounter; i++)
            {
                if (i != MinFreq0)
                {
                    if (OurTree[i].freq > 0 && OurTree[i].parent == -1)
                    {
                        if (MinFreq0 == -1 || OurTree[i].freq <
OurTree[MinFreq0].freq)
                        {
                            if (MinFreq1 == -1 || OurTree[i].freq <
OurTree[MinFreq1].freq)
                                MinFreq1 = MinFreq0;
                            MinFreq0 = i;
                        }
                        else if (MinFreq1 == -1 || OurTree[i].freq <
OurTree[MinFreq1].freq)
                            MinFreq1 = i;
                    }
                }
            }
            if (MinFreq1 == -1)
            {
                NumOfRootNode = MinFreq0;
                break;
            }

            //Combine two nodes to form a parent node
            OurTree[MinFreq0].parent = NodeCounter;
            OurTree[MinFreq1].parent = NodeCounter;
            OurTree[NodeCounter].freq = OurTree[MinFreq0].freq +
OurTree[MinFreq1].freq;
            OurTree[NodeCounter].right = MinFreq0;
            OurTree[NodeCounter].left =
MinFreq1; OurTree[NodeCounter].parent
= -1; NodeCounter++;
        }
    }
}

```

Code Section 6: Huffman Coding build tree function

5.3 Generic Compressor Class:

```
class GenericCompressor : public Compressor
{
public:
    GenericCompressor(char *pszName, int idrCompressExe, int
idrDecompressDll);
    ~GenericCompressor();

    // Size of the Compressed program
    virtual DWORD MeasureSize(char *pszFn) = 0;
    // Compress function calls different methods to compress the data. This is
    done by calling deflate functionalities of ZLIB.
    virtual BYTE *Compress(char *pszFnIn, DWORD *pcb) = 0;
    // Clears all the hashmaps and temporary files
    virtual void Cleanup() = 0;
    // gets decompression dll that should go with the compressed data to
    decompress correctly. This is done by inflate functionality of ZLIB.
    virtual BYTE *GetDecompressDll(DWORD *pcb) = 0;
    // Copies the name and icon of the original file.
    virtual char *GetName() = 0;
};
```

Code Section 7: Our main Compressor Class

The exe compression process goes through both the Compression Strategies and the resulting exe is the obtained which has the minimum size.

```

DWORD cbMin = MAXDWORD;
Compressor *pCompressorMin = NULL;
for (int i = 0; i < kcCompressors; i++) {
    Compressor *pCompressor = gapCompressor[i];
    DWORD cb = pCompressor->MeasureSize(szFnIn);
    if (cb < cbMin) {
        if (pCompressorMin
!= NULL)
            pCompressorMin->Cleanup();
        cbMin = cb;
        pCompressorMin = pCompressor;
    } else {
        pCompressor->Cleanup();
    }
}

```

Code Section 8: Go through all compressors

6. PE File Protection:

PE file protection has been implemented in the following ways in this thesis project.

6.1 Altering Executable structure:

In an executable, the PE header contains information describing the assets and general features. During the step of PE file protection; this PE header information is modified. The change mainly includes number of sections, Origin Point Address, Image Size and data's real virtual addresses and their magnitudes.

Addition of a novel section to the executable with the security stuff is a common method used in packing process. This extra added section is carries the essential knowledge to be used for the un-wrapping process. This might include the actual executable file headers and assemblies detached or changed during procedure for putting in the security. Following fig. describes how such protected PE file is structured.

PE Headers Dos, NT, Section Headers
Code Section
Data Section
Other Sections
Security Section Contains code stub, Information useful for unpacking dll

Figure 12: Adding new section

```

int Info::insert(Section *_section) // insert in sorted linked-list
{
    Section *new_section = new Section(_section);
    new_section->next=NULL;
    if (sort==byVirtualOffset) new_section->psort=&new_section->voffset;
    else if (sort==byRawOffset) new_section->psort=&new_section->roffset;
    if (head==NULL) tail=head=new_section;
    else if (*head->psort>*new_section->psort) {
        new_section->next=head;
        head=new_section;
    } else {
        Section *previous=head, *current=head->next;
        while (current!=NULL) {
            if (*current->psort>*new_section->psort)
                break;
            previous=current;
            current=current->next;
        }
        new_section->next=current;
        previous->next=new_section;
        if (new_section->next==NULL) tail=new_section;
    }
    return 0;
}

```

Code Section 9: Insert new section in a PE file

6.2 Modifying Import Table:

This is mainly done by altering the table containing Import Addresses (IAT). It mainly delivers information about the DLL imports and its purposes, which is used by the executable during runtime. The security is implemented by altering the address of Import table as well as changing the structure of table itself. The newly generated table is dependent upon the un-wrapping DLL. This un-wrapping DLL along with some code from the added protective section performs the un-wrapping operation.

6.3 Static Code Rerouting:

This rerouting procedure is a significant step towards a completely secure PE file. This method aims to reroute some JMP or CALL statements in the actual executable code towards the IJT which is contingent on the un-wrapping DLL. The static code redirection processes includes stripping the executable code, then choosing some JMP or CALL statements and then modify their aimed localities to matching IJT entry [15]. The un-wrapping DLL is used to reload the apt Interception Jump Table Entry code snippet so that the execution flow is redirected towards the original location.

Following code illustrates the code used for rerouting procedure in every IJT entry.

```
PUSHFD
PUSHAD
MOV     EAX, imm_Redirect
PUSH    imm_ImgBase
PUSH    imm_Entryndx
CALL    EAX
POP     EAX
POP     EAX
POPAD
POPFD
JMP     [TrueRVA]
```

Code Section 10: code snippet for rerouting procedure in every IJT entry

6.4 File Encryption:

This procedure should encode some parts of the executable file so that static or dynamic disassembling and code reverse engineering can be prevented. The defense procedure will encode the code sections, data directories, actual IAT and IJT and bury the key anywhere in

the executable, or the key is derived from certain sections of the executable using some mathematical calculations/algorithms. Adding several progressions of security on the executable file defies reverse engineering automation and makes it difficult for the disassembling software to strip the secured code. Encoding of the code is done primarily putting untrue algorithm stream in case of direct disassembly. The encoding of IJT makes the process of reversing the executable difficult as the procedure should be dynamic now.

```
void encrypt(char *ptr, int size)
{
    for (int i=0; i<size; i++)
        ptr[i]=ptr[i]^key;
}

// key is generated dynamically
```

Code Section 11: Simple Encryption for all the sections

6.5 Anti-Debug Methods:

1.) IsDebuggerPresent Windows Api: It will return none zero value whenever the current process is running in the context of a debugger.

2.) SoftIce Detection:

```

if(CreateFile( "\\\\.\\NTICE", GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL) !=INVALID_HANDLE_VALUE)
{
    There is SoftICE NT on your system;
}

OR

if(CreateFile( "\\\\.\\SICE", GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL) !=INVALID_HANDLE_VALUE)
{
    There is SoftICE98 on your svstem;
}

```

Code Section 12: SoftIce detection

3.) Insert Junk Code:

In this anti-debug technique, the tool inserts lot of junk code such that even if the PE file is opened in some unknown debugger, junk code will make sure that the reverse engineering gets frustrating.

6.6 Dynamic Code redirection:

We have seen Static code rerouting before. It is nice way to protect an exe file. The problem with it is that we need to keep attached un-packer dll at all times. This creates lot of overhead for the PE file and degrades its performance. To prevent this degrade, we implement Dynamic Code Redirection.

The Dynamic Code Redirection should offer an algorithm that, by some means, reduces the execution overhead without affecting the security of the PE file. This redirection should treat every IJT Entry as a separate unit and observe the amount of implementations of every unit. During the same interval, it should monitor the global amount of implementations of

all the units in the run time. These counters will act as a key component of the algorithm in balancing swiftness, performance efficiency and protection of the application. [15]

7. TESTS

As a test example, an executable binder.exe is taken. It is checked on a website known as www.virustotal.com, where in there are about 41 different anti-viruses who check your file.

Out of those 41, 38 of the viruses detect the existing threat.



The screenshot shows a Firefox browser window displaying the VirusTotal scan results for a file named **1337_EXE_Binder.exe**. The submission date is 2010-02-28 11:36:29 (UTC), the current status is finished, and the result is 38 / 41 (92.7%). A warning icon indicates the file was not reviewed and the safety score is -.

Below the summary, a table lists the results from 41 different antivirus engines. The table has four columns: Antivirus, Version, Last Update, and Result.

Antivirus	Version	Last Update	Result
a-squared	4.5.0.50	2010.02.28	Virus.Win32.VB.FK2!IK
AhnLab-V3	5.0.0.2	2010.02.27	Win-Trojan/Binder.376832.B
AntiVir	8.2.1.176	2010.02.26	TR/VB.Downloader.Gen
Antiy-AVL	2.0.3.7	2010.02.26	HackTool/Win32.Binder.gen
Authentium	5.2.0.5	2010.02.27	W32/Dropper.IZH
Avaast	4.8.1351.0	2010.02.27	Win32:VB-FKY
AVG	9.0.0.730	2010.02.27	Dropper.Generic.SNE
BitDefender	7.2	2010.02.28	Virtool.5098
CAT-QuickHeal	10.00	2010.02.27	HackTool.Binder.r (Not a Virus)
ClimAV	0.96.0.0-git	2010.02.28	Trojan.Dropper-2732
Comodo	4091	2010.02.28	TrojWare.Win32.HackTool.Binder.r
DrWeb	5.0.1.12222	2010.02.28	Trojan.MulDrop.14034
eSafe	7.0.17.0	2010.02.25	Win32.Binder.r
eTrust-Vet	35.2.7331	2010.02.26	-
F-Prot	4.5.1.85	2010.02.27	W32/Dropper.IZH
F-Secure	9.0.15370.0	2010.02.27	Virtool.5098
Fortinet	4.0.14.0	2010.02.28	Dropper.SC!tr
GData	19	2010.02.28	Virtool.5098
Ikarus	T3.1.1.80.0	2010.02.28	Virus.Win32.VB.FK2
Jiangmin	13.0.900	2010.02.28	TrojanDropper.Agent.ehb
K7AntiVirus	7.10.984	2010.02.26	HackTool.Win32.Binder

Now, the file is zipped in a .rar format and again the test is done on Binder.rar. The results show that 32/41 anti-viruses are still able to detect the threat.



File name: 1337_EXE_Binder.rar
 Submission date: 2009-07-12 15:48:52 (UTC)
 Current status: finished
 Result: 32 /41 (78.0%)

not reviewed
 Safety score: -

[Compact](#) [Print results](#)

Antivirus	Version	Last Update	Result
a-squared	4.5.0.18	2009.07.12	-
AhnLab-V3	5.0.0.2	2009.07.11	-
AntiVir	7.9.0.204	2009.07.11	TR/Crypt.ZPACK.Gen
Antiy-AVL	2.0.3.1	2009.07.10	HackTool/Win32.Binder.gen
Authentium	5.1.2.4	2009.07.11	W32/Dropper.IZH
Avast	4.8.1335.0	2009.07.11	Win32:VB-FKY
AVG	8.5.0.387	2009.07.12	Dropper.Generic.SNE
BitDefender	7.2	2009.07.12	Virtool.5098
CAT-QuickHeal	10.00	2009.07.10	-
ClamAV	0.94.1	2009.07.11	Trojan.Dropper-2732
Comodo	1626	2009.07.12	TrojWare.Win32.HackTool.Binder.r
DrWeb	5.0.0.12182	2009.07.12	Trojan.MulDrop.14034
eSafe	7.0.17.0	2009.07.12	Win32.Binder.r
eTrust-Vet	31.6.6608	2009.07.10	-
F-Prot	4.4.4.56	2009.07.11	W32/Dropper.IZH
F-Secure	8.0.14470.0	2009.07.12	HackTool.Win32.Binder.r
Fortinet	3.120.0.0	2009.07.12	Dropper.SC!tr
GData	19	2009.07.12	Virtool.5098
Ikarus	T3.1.1.64.0	2009.07.12	Virus.Win32.VB.FK2
Jiangmin	11.0.706	2009.07.12	TrojanDropper.Agent.ehb
K7AntiVirus	7.10.790	2009.07.11	HackTool.Win32.Binder
Kaspersky	7.0.0.125	2009.07.12	HackTool.Win32.Binder.r

Finally, the file is packed by the proposed Binder and it goes through encryption stage as shown below.


```
C:\Windows\system32\cmd.exe
376832 bytes compressed to 267776 bytes, 28.94% savings.
Crypter/Packer 1.0 by Neel

Original File Info
NumberOfSections for the file... 0x0002
Original EntryPoint of the file... 0x00001404
ImageBase for the Original...0x00400000
ImportTable RVA ...0x000014bc
ImportTable Size ... 0x00000050

File Sections Info
+-----+-----+-----+-----+-----+-----+
!   Name   ! VirtualSize ! Vir.Offset ! RawSize  ! RawOffset !   Flags   !
+-----+-----+-----+-----+-----+-----+
! .text    ! 0x00000740  ! 0x00001000 ! 0x00000800 ! 0x00000400 ! 0x60000020 !
! .rsrc    ! 0x00040864  ! 0x00002000 ! 0x00040a00 ! 0x00000c00 ! 0x40000040 !
+-----+-----+-----+-----+-----+-----+

Output file: bind.exe

Writing the crypted file [267776/267776 bytes]
Writing the stub ...
Done!
```

Results obtained when the file is checked on www.virustotal.com are quite positive this time around. Only 16/41 anti-virus tools are able to catch the existing threat. The results are shown below.

Firefox | Mark Stamp's CV | L-Z Data Compression | W Portable Executable Automatic Prote... | VirusTotal - Free Online Virus, Malwar... | +

http://www.virustotal.com/file-scan/report.html?id=642ca127b592e91f7a8af2836d42d92a3845162b994612e3afad43c9f790e207-1303492888

File name: bind.exe
Submission date: 2011-04-22 17:21:28 (UTC)
Current status: finished
Result: 16/41 (39.0%)

not reviewed
Safety score: -

[Compact](#) [Print results](#)

Antivirus	Version	Last Update	Result
AhnLab-V3	2011.04.23.00	2011.04.22	-
AntiVir	7.11.6.251	2011.04.22	TR/VB.Downloader.Gen
Antiy-AVL	2.0.3.7	2011.04.22	-
Avast	4.8.1351.0	2011.04.22	Win32:VB-FKY
Avast5	5.0.677.0	2011.04.22	Win32:VB-FKY
AVG	10.0.0.1190	2011.04.22	Generic22.JOF
BitDefender	7.2	2011.04.22	MemScan:Virtool.5098
CAT-QuickHeal	11.00	2011.04.21	-
ClamAV	0.97.0.0	2011.04.21	-
Commtouch	5.3.2.6	2011.04.22	-
Comodo	8436	2011.04.22	-
DrWeb	5.0.2.03300	2011.04.22	-
eSafe	7.0.17.0	2011.04.22	-
eTrust-Vet	36.1.8285	2011.04.22	-
F-Prot	4.6.2.117	2011.04.22	-
F-Secure	9.0.16440.0	2011.04.22	-
Fortinet	4.2.257.0	2011.04.22	-
GData	22	2011.04.22	MemScan:Virtool.5098
Ikarus	T3.1.1.103.0	2011.04.22	Backdoor.Win32.PoisonIvy.ag
Jiangmin	13.0.900	2011.04.22	-
K7AntiVirus	9.97.4451	2011.04.21	-
Kaspersky	7.0.0.125	2011.04.22	HackTool.Win32.Binder.r

Secure Search | McAfee

Firefox | Mark Stamp's CV | L-Z Data Compression | W Portable Executable Automatic Prote... | VirusTotal - Free Online Virus, Malwar... | +

http://www.virustotal.com/file-scan/report.html?id=642ca127b592e91f7a8af2836d42d92a3845162b994612e3afad43c9f790e207-1303492888

First seen: 2011-04-22 17:21:28
Last seen: 2011-04-22 17:21:28

TrID:
Win32 EXE Yoda's Crypter (67.9%)
Win32 Executable Generic (21.8%)
Generic Win/DOS Executable (5.1%)
DOS Executable Generic (5.1%)
Autodesk FLIC Image File (extensions: flc, fl1, cel) (0.0%)

sigcheck:
publisher.....: n/a
copyright.....: n/a
product.....: n/a
description...: n/a
original name: n/a
internal name: n/a
file version.: n/a
comments.....: n/a
signers.....: -
signing date.: -
verified.....: Unsigned

packers (F-Prot): embedded, MSLZ

PEInfo: PE structure information

```
[ [ basic data ] ]
entrypointaddress: 0x44000
timedatestamp....: 0x37618821 (Fri Jun 11 22:05:21 1999)
machinetype.....: 0x14c (I386)

[ [ 4 section(s) ] ]
name, viradd, virsiz, rawdsiz, ntropy, md5
.text_, 0x1000, 0x740, 0x800, 6.56, 8d8ce8eb7ad78d11bbbd64ad84431a04
.rsrc_, 0x2000, 0x40864, 0x40A00, 7.99, 4cace3932b3e2bda7847cf86e1a22c7d
.idata, 0x43000, 0x200, 0x200, 0.91, b19fc6c308364cc1e3500464ebb5615
.OB_, 0x44000, 0x200, 0x200, 2.53, 11d0f44a2ab22daf050968cb76ff4dfb

[ [ 1 import(s) ] ]
_KERNEL32.DLL: LoadLibraryA, GetProcAddress
```

ExifTool:

Secure Search | McAfee

8. CONCLUSION:

The technology to pack a portable executable file has gone through efficient and rich development through the use of various code packing cryptographic approaches. However, it is important to realize the progress made by un-packers too which render most of the current means incompetent. Multi core processors and dynamic analysis have made unpacking very effective and powerful. Hence, it is important to realize the integration of new encryption and compression methods to the current equipment. This thesis proposes a new system for greatly improving the packers. If the processed executable file is able to cut out some import information, implement dynamic redirection while debugging or implement the time-out mechanism, it will be possible to break any kind of dynamic scrutiny. This technique differs from the original packing process in that it doesn't measure the emulation. This thesis will provide a way to integrate a new approach to compress and encrypt with the currently used technology.

9. REFERENCES

- [1] Kang, M. G., Yin, H., & Poosankam, P. (n.d.). *Renovo*. Retrieved from Bit blaze Berkley: <http://bitblaze.cs.berkeley.edu/papers/renovo.pdf>
- [2] Molnár, L., Reiser, J., & Oberhumer, M. (1996, April). *UPX, the ultimate packer for executables*. Retrieved from sourcefourge.net: <http://upx.sourceforge.net/>
- [3] *Executable Compression*. (n.d.). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Executable_compression
- [4] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. *PolyUnpack: Automating the hidden-code extraction of unpack-executing malware*. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.
- [5] Miller, H. (2008, September 20). *Dual Mapping*. Retrieved from Uninformed: <http://www.uninformed.org/?v=all&a=44>
- [6] Rutkowska, J. (2004, November). *Redpill test*. Retrieved from Invisible things: <http://www.invisiblethings.org/papers/redpill.html>

[7] IQExplorer. (n.d.). *A Review Of Data Compression Technique*:

<http://www.slideshare.net/nayakslideshare/data-compression-technique>

[8] Armstrong, B. (n.d.). *Virtual PC Guy's blog*. Retrieved from MSDN:

http://blogs.msdn.com/virtual_pc_guy/archive/2005/01/24/359650.aspx

[9] Pietrek, M. (2002). *An In-Depth Look into the Win32 Portable Executable File* .

MSDN Magazine: <http://msdn.microsoft.com/en-us/magazine/cc301805.aspx>

[10] Silurian. (n.d.). *Anatomy Of PE Files*:

<http://www.silurian.com/inspect/peformat.htm>

[11] Esau, K. (2001). *Data Compression*. In *Distributed Systems Engineering*:

<http://www.conan.de/docs/compression.pdf>

[12] Feldspar, A. (1997). *An Explanation of Deflate Algorithm*:

<http://zlib.net/feldspar.html>

[13] Dawn Song, D. B. (2008). *TEMU: The BitBlaze Dynamic Analysis Component*.

Information Systems Security. Hydrebad: <http://bitblaze.cs.berkeley.edu/temu.html>

[14] LZ Compression. (n.d.): <http://www2.hawaii.edu/~wes/ICS212/Notes/LZ.html>

[15] *PE File*

protection: http://en.wikipedia.org/wiki/Portable_Executable_Automatic_Protection

[16] Lempel Ziv Algorithms: http://en.wikipedia.org/wiki/LZ77_and_LZ78

[17] Huffman, K. (2010). *Huffman Algorithm*:

<http://www.huffmancoding.com/my-family/my-uncle/huffman-algorithm>

[18] Zeeh, C. (2003). *The Lempel Ziv Algorithm. Famous Algorithms*:

<http://tuxtina.de/files/seminar/LempelZiv.pdf>

[19] *SIDT Instruction*: <http://pix.cs.olemiss.edu/>