Software Analysis and De-obfuscation Engine

By
NC. Faiza Khalid (Syndicate Leader)
NC. Komal Babar, NC. Nauvera Rehman, PC. Abdul Wahab

Submitted to the Faculty of Computer Science Department Military College of Signals, National University of Sciences and Technology, Rawalpindi in partial fulfilment for the requirements of a B.E. Degree in Computer Software Engineering
August 2009

ABSTRACT

SADE (Software Analysis and De-obfuscation Engine) is a software analysis toolkit that generically (without finding out the specifics of the compression and encryption scheme used) detects and unpacks a packed (encrypted and compressed) windows executable file (PE32 file) and makes the unpacked code available for analysis. SADE also shows additional information about the executable file (resources, imports, sections etc). The motivation behind the project is that the problem to generically unpack malicious executables has been solved to some extent commercially but the competitive nature of the anti-virus software industry refrain them from publishing a solution. There is hence a lack of publicly available generic unpacking tools that can handle a wide range and variety of packed executable files without knowing the exact packer used to pack it. Furthermore, the growing epidemic of malware has strengthened the need to have more freely available tools to help in analyzing packed executable files. The chief users of the application are security analysts and main area of application is malware analysis. Malware authors use packing techniques to hide their malicious code and security analysts need to uncover the hidden executable code for creating signatures and understanding attacks.

DEDICATION

The entire effort of this project is dedicated to our parents and family who were a constant source of inspiration and whose encouragement, understanding and support made this study possible.

ACKNOWLEDGMENT

This project owes its existence to the Computer Science Department of Military College of Signals (MCS), National University of Sciences and Technology (NUST). We are indebted to both our internal and external advisors, Lec. Aisha Khalid and Dr. Fauzan Mirza, whose assistance made this project possible. They provided invaluable guidance to us in all aspects of our work. Our special thanks to Mr. M. Bilal Siddique (MIS Cell) for his constant support and cooperation in project development.

We are also grateful to the faculty and administration of CS Department MCS for their encouragement and appreciation at all stages.

CONTENTS

LIST OF TABLES - viii
LIST OF FIGURES - x
1. INTRODUCTION - 1
1.1. The Packing Problem - 1
1.2. Magnitude of the Problem - 2
1.3. Incentive for Project - 4
1.4. Scope - 6
1.5. Achievements - 6
2. LITERATURE STUDY - 7
2.1. Microsoft Portable Executable File Format - - - - - - - - - - - - - - 7
2.2. Obfuscation - 13
2.3. De-Obfuscation - 16

2.4. Analysis Techniques - 17
2.5. Unpacking - 21
2.6. Unpacking Techniques - 22
2.7. Identifying Packed Binaries - 25
2.8. Published Unpacking Approaches -26
3. REQUIREMENTS - 32
3.1. The Overall Description - 32
3.2. User Characteristics - 32
3.3. Constraints - 33
3.4. Apportioning of Requirements - 33
3.5. Functional Requirements - 33
3.6. External Interfaces - 35

3.7. Non-functional Requirements - 35
4. SYSTEM DESIGN - 37
4.1. Introduction - 37
4.2. System Overview - 37
4.3. Design Considerations - 38
4.4. Unpacking Methodology - 41
4.5. Architectural Strategies - 41
4.6. System Architecture - 42
4.7. UML Diagrams - 43
4.8. Detailed Subsystem Designs - 52
4.9. Graphical User Interface (GUI) Design- - - - - - - - - - - - - - - - - 56
5. IMPLEMENTATION -63
5.1. Packing Detection - 63
5.2. Generic Unpacking - 67
5.3. Portable Executable File Information Retrieval- - - - - - - - - - - 73
5.4. Entropy and Statistical Information Graphs - - - - - - - - - - - - - 73
6. SOFTWARE TEST PLAN - 74
6.1. Introduction - 76
6.2. Assumptions and Constraints for Test Environment - - - - - - - 77
6.3. Test Methodology - 78
6.4. Test Levels - 86
6.5. Deliverables Matrix -87
6.6. Test Environment -88
6.7. Bug Severity and Priority Definition - - - - - - - - - - - - - - - - - - 88
6.8. Test Personnel - 89
6.9. Test Schedule - 90
6.10. Test Cases - 91
6.11. Test Results Report - 100

7. USER GUIDE -101
7.1. Starting SADE - 101
7.2. Loading the Executable - 101
7.3. Viewing PE File Information - 103
7.4. Viewing Runtime Details - 109
7.5. Viewing Entropy Distribution - 112

7.6. Other Statistics - 113
7.7. History - 114
7.8. Help - 114
8. FUTURE WORK -115
8.1. Disassembly of Code - 115
8.2. Reconstruction of Dumped Executable -115
8.3. Portability with 64-bit Windows Platform -116
ANNEX - A “Project Charter” -117
ANNEX - B “Research Paper” -123
ANNEX - C “Bibliography” -135

LIST OF TABLES

2.1 - Identifier for 32-bit and 64-bit executable files - 10
3.1 - Functional Requirements - 34
3.2 - External Interfaces - 35
3.3 - Performance Requirements - 35
3.4 - Quality Requirements -36
4.1 - Use Case Description - 45
4.2 - Class Diagram Description - 47
5.1 - Debug Events - 71
6.1 - Project Release Information - 75
6.2 - Project Team Information - 75
6.3 - Packing Detection Results -81
6.4 - Generic Unpacking Results - 82
6.5 - Unknown Packed Samples’ Results - 85
6.6 - Build Test System Specification -86
6.7 - Deliverable Matrix -87
6.8 - Severity List -89
6.9 - Priority List - 89
6.10 - Test Personnel -90
6.11 - Test Schedule -90
6.12 - Module Information - 91
6.13 - Test Case: Packing Detection - 92

6.14 - Test Case: Generic Unpacking - 93
6.15 - Test Case: PE File Information -94
6.16 - Test Case 1: Graphical User Interface -95
6.17 - Test Case 2: Graphical User Interface - 96
6.18 - Test Case 3: Graphical User Interface - 97
6.19 - Test Case: Packed Blocks Graph - 98
6.20 - Test Case: Graph Other Statistics - 99
6.21 - Test Results Report - 100

LIST OF FIGURES

1.1 - Distribution of Packed Malware in 2007 - 3
1.2 - Top Ten Packers by Websense Security Labs - - - - - - - - - - - - - - - - - - - 4
2.1 - PE File on Disk and Mapped in Memory - 8
2.2 - Typical Portable Executable File Layout - 8
4.1 - Architecture of SADE - 42
4.2 - Use Case Diagram of SADE - 44

4.3 - Class Diagram of SADE - 46
4.4 - Sequence Diagram for Scenario where Input File is Packed - - - - - - - - - - - 48
4.5 - Communication Diagram for Scenario where Input File is Packed - - - - - - 49
4.6 - Sequence Diagram for Scenario where Input File is Not Packed - - - - - - - 49
4.7 - Communication Diagram for Scenario where Input File is Not Packed - - - 50
4.8 - Data Flow Diagram of Packing Detection and Unpacking - - - - - - - - - - - - 51
4.9 - Data Flow Diagram of PE File Information Retrieval - - - - - - - - - - - - - - - -52
4.10 - Interface Design Model of SADE - 59
4.11 - GUI of PE File Header Information - 59
4.12 - GUI of PE Sections and Details - 60
4.13 - GUI of Unpacking Details - 60
4.14 - GUI of ASCII and Hex Dump - 61
4.15 - GUI of Entropy Results - 61
4.16 - GUI of Statistical Results - 62
5.1 - Algorithm (High Level View) of SADE - 63
5.2 - Algorithm for Packing Detection - 66
xi
5.3 - Algorithm (High Level View) for Generic Unpacking - - - - - - - - - - - - - - - - 67
6.1 - Notepad.exe (Not Packed) - 82
6.2 - AsPack (Unpacked File by SADE) - 83
6.3 - BeRoExPack (Unpacked File by SADE) -83
6.4 - ExeCryptor (Unpacked File by SADE) - 83
6.5 - PE Compact (Unpacked File by SADE) - 84
6.6 - Expressor (Unpacked File by SADE) - 84
6.7 - XPack (Unpacked File by SADE) - 84
7.1 - Starting SADE - 101
7.2 - ‘Browse’ Button - 102
7.3 - Selecting Input File -102
7.4 - Packing Detected - 103
7.5 - Packing Not Detected - 103
7.6 - View COFF Header - 104
7.7 - View Optional Header - 105
7.8 - View Section Header - 106
7.9 - View Data Directories - 107
7.10 - View Import Address Table (IAT) -108
7.11 - View APIs Called - 108
7.12 - Static ASCII Dump -109
7.13 - Static Hex Dump -109
7.14 - Runtime Details - 110
7.15 - Debug Events - 110
7.16 - Modules Loaded - 111
xii
7.17 - Unpacked File Dump - 112
7.18 - Entropy Distribution - 113
7.19 - Other Statistics - 113
7.20 - History - 114
7.21 - Help Window - 114

CHAPTER 1
INTRODUCTION

In the past it was not uncommon to see malware that used no encryption at all to hinder analysis. Over time malware authors have jumped on the encryption bandwagon as a means of obscuring their activities, whether they seek to protect communications or whether they seek to prevent disclosure of the contents of a binary.

A runtime packer is a software “envelope” used by malware authors to hide the functionality of an executable file. A number of packers are available publicly and are used by legitimate software to reduce the size of their executable files and to protect the intellectual property that is distributed with their code. *Packers encode programs to which they are applied. Upon execution, the packer will decode and decompress the original program in memory and execute it.* If a key value is modified in the encoding routine, the binary file that is produced looks completely different, especially to security software that relies on detection by known signature [1].

1.1. The Packing Problem

Packing is a data hiding technique that replaces a binary (code and data) sequence with a data block containing the binary sequence in encrypted and compressed form and a decryption routine that, at runtime, recovers the original binary sequence from the data block. The result of packing is a program that dynamically generates code in memory and then executes it. There are a large number of tools available for this purpose commonly known as executable packers. Packing describes the process of encrypting a program and adding a runtime decryption routine to it, such that the behaviour of the original program is preserved. By randomly choosing encryption keys, it is possible to create a multitude of instances from one original program. The encryption completely changes the binary signature of a program, and malware authors commonly use packing to evade string-based malware detectors. The malicious code resides in the executable file in an encrypted form, and is not

exposed until the moment the executable is run. Thus, a scan string algorithm will fail to detect the malware by reading the file, unless it is updated with a new scan string tailored towards this specific packed instance of the malware [2].

Packing resembles encryption because it also compresses the code, greatly reducing the file size of the executable program. This feature is important for programs that are to be distributed over the Internet. A packed binary appears to be an undecipherable series of symbols, and only the decryption routine (or only a part of it in the case of polymorphic code) is visible on the disk. After the decryption routine has been executed, the original program becomes visible in memory. Once in memory, the program can be edited dynamically. The memory dumping process is complicated by anti-memory dumping code, and the fact that the PE (Portable Executable) headers and Import Address Tables need restoration to their original state [3].

Packers were first written in order to provide a mechanism to shrink executables so they take less space to store and less time to transfer over slow channels. Later on, their usage started taking another scope when malware authors used them to conceal their parasites. There are a few reasons behind this close correlation between malware authors and packers. Packers always offered a sanctuary for malware authors where they managed to disguise the code and data which their malware contained. Likewise, in some cases, packers provided them with different looking binaries each time they repacked their code (a technique commonly used against check summing). Malware authors can unleash a new malware by simply repacking a known virus without any significant work on its coding aside from minor changes.

1.2. Magnitude of the Problem

Undoubtedly, the single most challenging problem Anti Virus (AV) vendors currently face is the problem of packed malware. Traditionally, AV vendors dealt with packers by providing their engines with un-packing routines to normalise the executables and generate the unpacked version of the file. This was achieved through two different ways. By either releasing updated engines, or releasing new signatures that contained the unpacking routines which will be interpreted by the engines at the users' end. AV vendors who have followed the former method suffered from the slow cycles in which their engines had to go through before being fully tested and ready to

be released. While the latter method gave some AV vendors an edge by providing faster updates, they still had to invest more resources into analysing the different packers that got released restlessly. A further extension for the problem is the fact that some software vendors, for no obvious reasons decide to publish their software packed with rather very suspicious packers.

In 2007, Avert Labs started experiencing the birth of at least one packer or a variant of a packer on a daily basis. Figure 1.1 shows the distribution of packed malware experienced by Avert Labs in 2007. It shows that the rules of the game have changed and it is no longer a few packers that dominated the scene. Instead, there are many variants of unknown or patched packers. This relationship is projected to worsen simply because mass producing those packers is much less costly than coding their unpackers [4].

Figure 1.1 - Distribution of packed malware in 2007

According to the research of WSLabs (Websense Security Labs), more than 80% malicious codes are disguised by a certain packer. Figure 1.2 shows the top ten packers (of all times) from the tracker of WSLabs [5]. The essential problems raised by packers are that they have made it easier to produce “new” malware. Using packers brings more difficulties to analysts and researcher. Also, the packed files are smaller in size and hence easier to propagate.

Figure 1.2 - Top ten packers by Websense Security Labs

1.3. Incentive for Project

One way that malware writers have been using to disguise their malicious software is executable compression and encryption. They use tools that take a Windows PE-file (or another platform executable file) and compress it in such a way that the compressed file only decompresses itself in memory at runtime. The PE-files compression presents two problems for antivirus engines. First, to detect a known malware, the file has to be decompressed before the point where signature matching can occur can be reached. Secondly, it is necessary to decompress the file so that strong code analysis heuristics can be applied. There are some already available methods that can be used to decompress these files to the point at which they can be analyzed further. They have both advantages as well as disadvantages. There is definitely scope for enhancement.

Packed malicious programs (malware) pose a striking problem in malware analysis, detection, and forensics. Such programs consist of a decompression or decryption routine that extracts the garbled payload from memory and then executes it. This unpacking routine can be invoked once, in which case the whole payload is extracted to memory in a single step, or multiple times, when parts of the payload are extracted to memory at different times. For a security analyst, this means that the program has to be executed in a contained yet accurate environment before an analysis of the payload can be performed. For a malware detector, this means that the scanning for malicious code has to be postponed until after the start of execution, i.e., when the

program has unpacked its payload. Malware writers have learned that binary packers are effective at bypassing signature-based detectors and at keeping the malware undetected for longer. The numerous packers currently available generate many variants from the same executable. The percentage of new malware that is packed is on the rise, from 29% in 2003 to 35% in 2005 up to 80% in 2007. This situation is further complicated by the ease of obtaining and modifying the source code for various packers (e.g., UPX). Alterations to the source code can introduce changes in the compression or encryption algorithm, create multiple layers of encryption, or add protection against reverse engineering. Currently, new packers are created from existing ones at a rate of 10–15 per month. As a result, malware writers have a large selection of tools to pack their malware, to the point that more than 50% of malware samples are simply repacked versions of existing malware [6].

A preliminary requirement in the executable analysis is the capability to robustly parse and analyze executable files. Dealing with the full spectrum of executables found in the wild is quite demanding. While normal files are typically well structured, malicious files can be quite difficult to analyze, often due to deliberate malformations intended to foil static analysis. The consequence of successfully applying packing techniques is that static analysis of the file will view the obfuscated block as noninstruction data or omit its analysis entirely, thereby hiding the code's true intentions.

The capacity of information security practitioners to implement models of detection and methods of recovery against malware are often thwarted by instances of packed malware, such as encrypted and polymorphic viruses [7].

Time must be invested to learn the mechanism by which a given instance of malware unpacks its compile-time obfuscated code (usually the malicious component) so that it can be extracted and studied. Some Computer Emergency Response Teams (CERTs) report that as many as 160 new viruses arrive each day, out of many hundreds of sample submissions. Given this volume, the process of unpacking alone (before any analysis is performed) can be overwhelming. Further, resources can be wasted in determining whether a new malware sample contains unpack-execute behaviour, or when two or more new samples found turn out to be the same malware with well-differentiated unpacking methods [8]. Generic unpacking is the solution to the threat of diverse packing techniques. Automating the unpacking process in a generic way so that it is not hampered by continually evolving packing schemes can go a long way to save the time and effort of security analysts and help the overall malware detection process.

1.4. Scope

Development of a toolkit for executable analysis that detects whether any packing transformation has been applied on the input executable and then upon detecting packing, unpacks the executable code and data and makes it available for analysis along with retrieving useful executable file information.

The chief users of the toolkit are security analysts. Malware writers use executable encryption and compression to hide their malicious code and security analysts need to uncover the hidden data and code for creating signatures and understanding attacks.

1.5. Achievements

Our **Research Paper** with the title "Generic Unpacking Techniques" has been presented at and published by *IEEE-International Conference on Computer, Control & Communication* (IC4), 2009. The research paper is available at IEEE digital library on IEEE-explorer.

This Project SADE scored **3rd position** in 8th All Pakistan Software Project Exhibition and Competition COMPPEC, 2009 organized by NUST at College of Electrical & Mechanical Engineering.
7

CHAPTER 2

LITERATURE STUDY

This chapter provides background knowledge about the project. Pre-requisite information pertaining to the project has been covered as well as a comprehensive study of already existing packing techniques and other related work in the field of information security.

2.1. Microsoft Portable Executable File Format

Microsoft introduced the Portable Executable File format, more commonly known as the PE format, as part of the original Win32 specifications. The term "Portable Executable" was chosen because the intent was to have a common file format for all flavours of Windows, on all supported CPUs. A very handy aspect of PE files is that the data structures on disk are the same data structures used in memory. Loading an executable into memory is primarily a matter of mapping certain ranges of a PE file into the address space. The key point is that if you know how to find something in a PE file, you can find the same information when the file is loaded in memory.
PE files are not just mapped into memory as a single memory-mapped file. Instead, the Windows loader looks at the PE file and decides what portions of the file to map in. This mapping is consistent in that higher offsets in the file correspond to higher memory addresses when mapped into memory. The offset of an item in the disk file may differ from its offset once loaded into memory. However, all the information is present to allow anyone to make the translation from disk offset to memory offset, as shown in figure 2.1. The central location where the PE format is described is WINNT.H. Within this header file, nearly every structure definition, enumeration, and #define needed to work with PE files or the equivalent structures in memory can be found [9]. Figure 2.2 shows the detailed layout of a typical PE file. First section in figure 2.2 is the MS-DOS 2.0 section which is placed for backward compatibility with MS-DOS. Next there is the PE header and then the section headers. Finally, there are the PE file directories which include the import pages containing the import information, export information, base relocations and resource information.
8

Figure 2.1 – PE File on disk and mapped in memory

Figure 2.2 - Typical Portable Executable File Layout

9

2.1.1. PE File Sections

A PE file section represents code or data of some sort. While code is just code, there are multiple types of data. Besides read/write program data (such as global variables), other types of data in sections include API import and export tables, resources, and relocations. Each section has its own set of in-memory attributes, including whether the section contains code, whether it's read-only or read/write, and whether the data in the section is shared between all processes using the executable. Each section has a distinct name. This name is intended to convey the purpose of the section. For example, a section called .rdata indicates a read-only data section.
Sections have two alignment values, one within the disk file and the other in memory. The PE file header specifies both of these values, which can differ. Each section starts at an offset that's some multiple of the alignment value. For instance, in the PE file, a typical alignment would be 0x200. Thus, every section begins at a file offset

that's a multiple of 0x200. Once mapped into memory, sections always start on at least a page boundary. That is, when a PE section is mapped into memory, the first byte of each section corresponds to a memory page. On x86 CPUs, pages are 4KB aligned, while on the IA-64, they're 8KB aligned.

2.1.2. Relative Virtual Addresses (RVA)

PE files can load just about anywhere in the process address space. For this reason, it's important to have some way of specifying addresses that are independent of where the executable file loads. To avoid having hardcoded memory addresses in PE files, RVAs are used. An RVA is simply an offset in memory, relative to where the PE file was loaded. To convert an RVA to an actual address, add the RVA to the actual load address to find the actual memory address. The actual memory address is called a Virtual Address (VA).

2.1.3. The MS-DOS Header

Every PE file begins with a small MS-DOS executable. The need for this stub executable arose in the early days of Windows, before a significant number of consumers were running it. When executed on a machine without Windows, the

program could at least print out a message saying that Windows was required to run the executable. The first bytes of a PE file begin with the traditional MS-DOS header, called an IMAGE_DOS_HEADER. The only two values of any importance are e_magic and e_lfanew. The e_lfanew field contains the file offset of the PE header. The e_magic field needs to be set to the value 0x5A4D. There's a #define for this value, named IMAGE_DOS_SIGNATURE. In ASCII representation, 0x5A4D is MZ, the initials of Mark Zbikowski, one of the original architects of MS-DOS.

2.1.4. PE Signature

After the MS-DOS stub, at the file offset specified at offset 0x3c, is a 4-byte signature that identifies the file as a PE format image file. This signature is “PE\0\0” (the letters “P” and “E” followed by two null bytes). This can be used to validate an input executable as a correct windows executable.

2.1.5. Common Object File Format File Header and Optional Header

At the beginning of an object file, or immediately after the signature of an image file, is a standard Common Object File Format (COFF) file header. Every image file has an optional header that provides information to the loader. The size of the optional header is not fixed. The SizeOfOptionalHeader field in the COFF header must be used to validate that a probe into the file for a particular data directory does not go beyond SizeOfOptionalHeader. The IMAGE_NT_HEADERS structure is the primary location where specifics of the PE file are stored. Its offset is given by the e_lfanew field in the IMAGE_DOS_HEADER at the beginning of the file. There are actually two versions of the IMAGE_NT_HEADER structure, one for 32-bit executables and the other for 64-bit versions. The optional header magic number determines whether an image is a PE32 (32 bit) or PE32+ (64 bit) executable, as shown in table 2.1.

Table 2.1 – Identifier for 32-bit and 64-bit executable files

Magic Number	PE Format
0x10b	PE32
0x20b	PE32+

This information can be used to validate that the input portable executable file is 32-bit for which the software has been tailored.

2.1.6. Optional Header Data Directories

Each data directory gives the address and size of a table or string that Windows uses. These data directory entries are all loaded into memory so that the system can use them at run time.

2.1.6.1. The Section Table

Each row of the section table is, in effect, a section header. This table immediately follows the optional header. This positioning is required because the file header does not contain a direct pointer to the section table. Instead, the location of the section table is determined by calculating the location of the first byte after the headers. The number of entries in the section table is given by the NumberOfSections field in the file header. Entries in the section table are numbered starting from one. The code and data memory section entries are in the order chosen by the linker. In an image file, the virtual addresses for sections must be assigned by the linker so that they are in ascending order and adjacent, and they must be a multiple of the SectionAlignment value in the optional header.

2.1.6.2. The .debug Section and Debug Directory

The **.**debug section is used in object files to contain compiler-generated debug information and in image files to contain all of the debug information that is generated. This section describes the packaging of debug information in object and image files. Image files contain an optional debug directory that indicates what form of debug information is present and where it is. This directory consists of an array of debug directory entries whose location and size are indicated in the image optional header. The debug directory can be in a discardable .debug section or it can be included in any other section in the image file, or not be in a section at all. The debug directory can be used to find out the name of the environment or tool that generated the executable and the version of the tool. It can also tell the time of creation of the executable file.

2.1.6.3. The Imports Section and Import Directory Table

All image files that import symbols, including virtually all executable (EXE) files, have an .idata section. The import information begins with the import directory table, which describes the remainder of the import information. The import directory table contains address information that is used to resolve fix up references to the entry points within a DLL image. The import directory table consists of an array of import directory entries, one entry for each DLL to which the image refers. The last directory entry is empty (filled with null values), which indicates the end of the directory table [10].

2.1.7. Dynamically Linked Libraries (DLLs)

Dynamically linked libraries (DLLs) are a key feature in a Windows. The idea is that a program can be broken into more than one executable file, where each executable is responsible for one feature or area of program functionality. The benefit is that overall program memory consumption is reduced because executables are not loaded until the features they implement are required. Additionally, individual components can be replaced or upgraded to modify or improve a certain aspect of the program. From the operating system's standpoint, DLLs can dramatically reduce overall system memory consumption because the system can detect that a certain executable has been loaded into more than one address space and just map it into each address space instead of reloading it into a new memory location. DLLs are different from build-time static libraries (.lib files) as static libraries are permanently linked into an executable. With static libraries, the code in the .libfile is statically linked right into the executable while it is built, just as if the code in the .libfile was part of the original program source code. When the executable is loaded the operating system has no way of knowing that parts of it came from a library. If another executable gets loaded that is also statically linked to the same library, the library code will essentially be loaded into memory twice, because the operating system will have no idea that the two executables contain parts that are identical. Windows programs have two different methods of loading and attaching to DLLs in runtime. *Static linking* refers to a process where an executable contains a reference to another executable within its import table. This is the typical linking method that is employed by most application programs, because it is the most convenient to use. Static linking is implementing by having each module list the modules it uses and the functions it calls within each module (this is called the *import table)*. When the loader loads such an executable, it

also loads all modules that are used by the current module and resolves all external references so that the executable holds valid pointers to all external functions it plans on calling. *Runtime linking* refers to a different process whereby an executable can decide to load another executable in runtime and call a function from that executable. The principal difference between these two methods is that with dynamic linking the program must manually load the right module in runtime and find the right function to call by searching through the target executable's headers. Runtime linking is more flexible, but is also more difficult to implement from the programmer's perspective. From a reversing standpoint, static linking is easier to deal with because it openly exposes which functions are called from which modules [11].

2.2. Obfuscation

One of the most prevalent features of modern malware is obfuscation. Obfuscation is the process of modifying something so as to hide its true function. In the case of malware, obfuscation is used to make automated analysis of the malware nearly impossible and to aggravate manual analysis to the maximum extent possible. The term obfuscation refers to techniques that preserve the program's semantics and functionality while at the same time making it more difficult for the analyst to extract and comprehend the program's structures. In the context of disassembly, obfuscation refers to transformations of the binary such that the parsing of instructions becomes difficult. Besides obfuscation techniques to increase the difficulty of the disassembly process, the code itself can be obfuscated to make it difficult to extract the control flow of a program or to perform data flow analysis. The basic idea for such obfuscation techniques is that they can be automatically applied, but not easily undone, even if the transformation approach is known. Finally, the code that is analyzed by a static analyzer may not necessarily be the code that is actually run. In particular, this is true for self-modifying programs that use polymorphic and metamorphic techniques and packed executables that unpack themselves during run-time.

Sophisticated techniques for protecting code against reverse engineering are called code obfuscation. An obfuscator transforms a program into an obfuscated program that displays the same observable behaviour but is illegible. The quality of an obfuscator is measured by its potency, resilience and cost. Potency is the amount of subjective complexity added to a program, thus making it harder for humans to

comprehend the functionality. Resilience describes the robustness of the obfuscation against automated de-obfuscation methods and cost refers to the magnitude of additional time and space consumption caused by the transformation. Some of the common obfuscation transformations are dead code insertion, code reordering, instruction substitution and packing. These different methods of obfuscation are briefly described in the subsequent sections.

2.2.1. Dead Code Insertion

Dead code insertion is the simplest of the obfuscation techniques. It means to insert instructions or sequences of instructions without changing the machine state at random points in the program. Examples of dead code on x86 architectures are the NOP (No Operation or No Operation Performed) instruction or statements such as mov eax, eax that have no effect on the code execution yet change the binary signature of the program. Dead code insertion changes the binary footprint of a piece of malicious software, yielding false negatives in traditional anti-virus products. NOPs are often involved when cracking software that checks for serial numbers, specific hardware or software requirements etc. This is accomplished by altering functions and subroutines to by-pass security checks and simply returning the expected value being checked for.

2.2.2. Code Reordering

A sequence of binary code can be broken into several pieces and put together in a random order by connecting subsequent instructions in the original code through unconditional jumps. As long as addresses used in the code are rewritten during the process, the program semantics are not affected even though the resulting binary executable is new.

2.2.3. Instruction Substitution

In large instruction sets such as those of the x86 processor family, various instructions can be used to perform equivalent operations. Substituting an instruction with its equivalent would not change the outcome of the program but could change its signature [12].

2.2.4. Packing

Packing is the method that an executable uses to obfuscate an executable or to reduce its size. Packers are typically implemented with a small decoder stub which is used to unpack or obfuscate a binary in question. Once the decoding process is complete, the decoder stub transfers control back to the original code of the program. Execution then proceeds similar to that of a normal executable. Packing involves compressing an executable file but leaving it in an executable state. An infected executable can thereby be changed by the packing process such that its signature becomes completely different while remaining executable.

2.2.4.1. Packer Tools

Tools used to obfuscate compiled binary programs are generically referred to as *packers.* This term stems from the fact that one technique for obfuscating i.e. disguising a binary program is simply to compress the program, as compressed data tends to look far more haphazard, and certainly does not resemble machine language. For the program to stay executable on the target computer, it must remain a valid executable for the target platform. The standard approach taken by most packers is to embed an unpacking stub in to the packed program and to modify the program entry point to point to the unpacking stub.

When the packed program executes, the operating system reads the new entry point and initiates execution of the packed program at the unpacking stub. The purpose of the unpacking stub is to restore the packed program to its original state and then to transfer control to the restored program. Packers vary significantly in their degree of sophistication. The most basic packers simply perform compression of a binary’s code and data sections. More sophisticated packers not only compress, but also perform some degree of encryption of the binary’s sections. Finally, many packers will take steps to obfuscate a binary’s import table by compressing or encrypting the list of functions and libraries that the binary depends upon [13].

Programs obfuscated by packing consist of a decryption routine (an instruction sequence that generates code and data), a trigger instruction that transfers control to the generated code, an unpacked area (the memory area where the generated code resides) and a packed area (the memory area from where the packed original binary is read). PE packers typically take the existing sections of the image file to be

packed, compress them, and store them in a new section within the packed executable. Then they add the unpacking stub, possibly some more data needed during the unpacking process, and new headers to correctly describe the packed file. This typically includes creating a new section that will contain the unpacked data, with a raw size of zero and the virtual size set to at least the size of the unpacked data, adding new sections to contain the packed data and the unpacking stub, and setting the entry point to the entry point of the unpacking stub. At the same time, PE packers typically remove most of the original import data as well and keep or add only a few imports, as a bare minimum only for the LoadLibraryA and the GetProcAddress from kernel32.dll. At load time, the unpacking stub is executed which unpacks the packed code into the empty section reserved for it. Then, the stub typically resolves the original imports, using the LoadLibraryA API call to have Windows load dynamic link libraries into the process's address space and return handles to them, and using the GetProcAddress API call to obtain the virtual addresses of symbols these libraries export. These virtual addresses are then written to the unpacked executable's import address table. Finally, control is transferred to the unpacked code's entry point, typically dubbed original entry point (OEP) in this context, the unpacking process is complete, and the original code should be able to execute as if nothing had happened [14].

2.3. De-Obfuscation

De-obfuscation is the opposite process of obfuscation. There are two basic ways to deal with obfuscation. The first way is to simply ignore it, in which case your only real option for understanding the nature of a piece of malware is to observe its behaviour in a carefully instrumented environment such as a virtual computer. The second way to deal with obfuscation is to take steps to remove the obfuscation and reveal the original "de-obfuscated" program, which can then be analyzed using traditional tools such as disassemblers and debuggers. Of course, malware authors understand that analysts will attempt to break through any obfuscation, and as a result they design their malware with features designed to make de-obfuscation tricky. De-obfuscation can never be made truly unattainable since the malware must eventually run on its target CPU; it will always be possible to view the sequence of instructions that the malware executes using some combination of hardware and software tools. In all likelihood, the malware author's goal is simply to make analysis amply complicated

that a window of opportunity is opened for the malware in which it can run without discovery.

2.4. Analysis Techniques

Malware analysis is the process of determining the purpose and functionality of a given malware sample (such as a virus, worm or Trojan horse). This process is a necessary step to be able to develop effective detection techniques for malicious code. In addition, it is an important prerequisite for the development of removal tools that can thoroughly delete malware from an infected machine. Traditionally, malware analysis has been a manual process that is tedious and time intensive. Unfortunately, the number of samples that need to be analyzed by security vendors on a daily basis is constantly increasing. This clearly reveals the need for tools that automate and simplify parts of the analysis process. Analyzing unknown executables is not a new problem. Consequently, many solutions already exist. These solutions can be divided into two broad categories: *static analysis* and *dynamic analysis* techniques.

2.4.1 Static Analysis

Static analysis is the process of analyzing a program's code without actually executing it. In this process, a binary is usually disassembled which denotes the process of transforming the binary code into corresponding assembler instructions. Then, both control flow and data flow analysis techniques can be employed to draw conclusions about the functionality of the program. A number of static binary analysis techniques have been introduced to analyze different types of malware. Static analysis has the advantage that it can cover the complete program code and is usually faster than its dynamic counterpart. However, a general problem with static analysis is that many interesting questions that one can ask about a program and its properties are un-decidable in the general case. Of course, there exists a rich body of work on static analysis techniques that demonstrate that many problems can be approximated well in practice, often because difficult to handle situations occur rarely in real-world software. Unfortunately, the situation is different when dealing with malware. Because malicious code is written directly by the adversary, it can be crafted deliberately so that it is hard to analyze. In particular, the attacker can make use of binary obfuscation techniques to thwart both the disassembly and code

analysis steps of static analysis approaches. There are many tools out there to do basic static analysis such as PEiD [15].

Static analysis can be used to gather a variety of information about an executable e.g., high-level information such as its file size, a cryptographic hash, its file format, imported shared libraries, the compiler used to generate it, or even just a list of human-readable strings that are contained in the file, or, low-level information gathered by disassembling or decompiling the specimen. Information about the file format, shared library or compiler can aid disassembly or de-compilation. Cryptographic hashes can be used to identify a specimen. Packer signatures or its entropy may be used to determine whether it might be runtime packed.

Static analysis has several advantages over dynamic approaches. As static methods do not involve executing a potentially malicious program, there is a lesser risk of damaging the system that analysis is performed on. Given availability of the right tools, it is also possible to perform the analysis on a platform that differs from the platform that the specimen is designed to run on, further mitigating the risk of damaging the analysis platform (e.g., by accidentally executing it). Furthermore, static analysis typically covers the whole program and not just those code paths that are executed for a set of inputs, like dynamic analysis. There are however some disadvantages, too. Determining a sample’s behaviour through low-level static analysis, like disassembly, is typically very time-consuming and requires a lot of knowledge and skill. Static analysis also has trouble dealing with self-modifying code and packed binaries as these generate new code during execution, behaviour that is hard to capture without executing a specimen.

2.4.2. Dynamic Analysis

In contrast to static techniques, dynamic techniques analyze the code during runtime. While these techniques are non-exhaustive, they have the significant advantage that only those instructions are analyzed that the code actually executes. Thus, dynamic analysis is immune to obfuscation attempts and has no problems with self-modifying programs. When using dynamic analysis techniques, the question arises in which environment the sample should be executed. Of course, running malware directly on the analyst’s computer, which is probably connected to the Internet, could be disastrous as the malicious code could easily escape and infect other machines. Furthermore, the use of a dedicated standalone machine that is

reinstalled after each dynamic test run is not an efficient solution because of the overhead that is involved.

As Dynamic analysis is a way of analysing an unknown program by executing it and observing its behaviour, careful consideration must be given to securing the analysis environment, so as not to risk damaging the systems on which it is run. The simplest solution to containing hostile code is the so-called sacrificial lamb, which is a real machine with no or limited network access, which can be disposed of or wiped clean and reinstalled after an analysis run. There also exist hardware and software solutions to automate the task of restoring a real machine to an untouched state.

Dynamic analysis can be performed at different levels of abstraction. In the simplest case, a security researcher can record the initial system state, execute the program

to be analysed and examine the system state after execution and make note of all changes. Additionally, the researcher can monitor the system's inputs and outputs during execution, e.g., network activity. More fine-grained dynamic analysis involves tracing a program's behaviour, which, again, can be performed at various levels of abstraction. System call tracing captures the interaction of a program with the operating system, on transitions from user mode to kernel mode code. System call tracing can also quickly generate a lot of data that might be hard for a human analyst to process. To perform analysis at a higher level of abstraction on other operating systems, library call tracing is another method that can be employed for dynamic analysis. For programs written in a high-level language, this method provides a more natural view of a program's inner workings than system call tracing, if the program uses the high-level language's standard libraries (which is common practice).
Dynamic analysis has several advantages. At high levels of abstraction, e.g., library call tracing, it can quickly give a researcher an overview of what a certain program does. It is largely immune to obfuscation techniques that target static analysis methods, like self-modifying code, including runtime-packing or encryption, or antidisassembly tricks. But there are also drawbacks. Dynamic analysis typically only covers one possible execution path through a program [14].

2.4.2.1. Virtual Machines

Running the executable in a virtual machine (that is, a virtualized computer) can only affect the virtual PC and not the real one. After performing a dynamic analysis run,

the infected hard disk image is simply discarded and replaced by a clean one (i.e., so called *snapshots*). Virtualization solutions are sufficiently fast. There is almost no difference to running the executable on the real computer, and restoring a clean image is much faster than installing the operating system on a real machine. Virtual machines are easier to handle than real machines and provide greater flexibility, e.g., they allow a researcher to save one or more snapshots of the machine state, which can later be restored. On the other hand, virtual machines are normally slower than real machines and the executable to be analyzed may determine that it is running in a virtualized machine and, as a result, modify its behaviour. In fact, a number of different mechanisms have been published that explain how a program can detect if it is run inside a virtual machine.

2.4.2.2. Sand-Boxing

Sand-boxing systems are a relatively new approach to handling malicious code. Sand-boxing solutions introduce cages, "virtual subsystems" of the actual operating system. The idea is to let the unknown program run on a virtual machine that accesses the same information which the user can access on the local machine but has access to a copy of the information within the cage. On the virtual system, the known program, such as a computer virus, will be able to read files that are "on the real system," even read the Registry keys and so on, but its networking capabilities are reduced. And when it attempts to make any changes, it makes them in the replica of information within the cage. Thus the virus is free to do anything it wants, but this will happen in a cage instead of on the real system. When the application finishes execution, the file and Registry changes can be thrown away, and malicious-looking actions can be logged. Unfortunately, this solution comes with a few limitations such as compatibility problems and the virtualized system might have holes that are similar to those of behaviour-blocking systems. Tricky malicious code might be able to execute unwanted functions on the real machine instead of the virtual machine [11].

2.4.2.2. Emulator

A PC emulator is a piece of software that emulates a personal computer (PC), including its processor, graphic card, hard disk, and other resources, with the purpose of running an unaffected operating system. The difference between virtual machines and PC emulators is that virtual machines can run an unaffected operating system but they execute a statistically dominant subset of the instructions directly on the real CPU. This is in contrast to PC emulators, which simulate all instructions in software. Because all instructions are emulated in software, the system can appear exactly like a real machine to a program that is executed, yet keep complete control. Thus, it is more difficult for a program to detect that it is executed inside a PC emulator than in a virtualized environment. However, there is one observable difference between an emulated and a real system: speed of execution. This fact could be exploited by malicious code that relies on timing information to detect an emulated environment. It is possible for the emulator to provide incorrect clock readings to make the system appear faster for processes that attempt to time execution speed.

In addition to differentiating the type of environment used for dynamic analysis, one can also distinguish and classify different types of information that can be captured during the analysis process. Many systems focus on the interaction between an application and the operating system and intercept system calls or hook Windows API calls. These tools are implemented as operating system drivers that intercept native Windows system calls. As a result, they are invisible to the application that is being analyzed. They cannot, however, intercept and analyze Windows API calls or other user functions. On the other hand, tools exist that can intercept arbitrary user functions, including all Windows API calls. The complete control offered by a PC emulator potentially allows the analysis that is performed to be even more fine grain. Similar to the functionality typically provided by a debugger, the code under analysis can be stopped at any point during its execution and the process state (i.e., registers and virtual address space) can be examined [16].

2.5. Unpacking

Unpacking consists of constructing a program instance which contains the embedded program, contains no code-generating routine, and behaves equivalently to the selfgenerating program. Packing does not change the relevant behavior of a program.

Hence, reverting back of a packed program (called unpacking) consists of recovering the original program that has the same relevant behaviour as the packed program. In a packed program the decryption routine precedes the execution of the original program. To ensure that the original program is correctly restored at runtime, the

decryption routine generates the same results every time the program is run, regardless of any input to the program. Furthermore, the operations of the decryption routine affect only program memory. As a consequence, it is possible to create an equivalent program not containing the decryption routine by setting all the values in the unpacked area to the expected results of its computation beforehand.

2.6. Unpacking Techniques

There are several ways to unpack packed executables. The main weakness of typical runtime-packers and executable protectors is that at some point, the original code must be executed. This can be exploited by manual unpacking, i.e., by debugging the executable to be unpacked, and determining when the original code is completely unpacked and the unpacking stub is about to branch to it. Then, the process memory can be dumped and an attempt be made to regenerate the original executable by fixing headers and reconstructing an import address table. Another method is reverse engineering the unpacking stubs of individual packers and using that knowledge to create packer-specific unpackers that can statically unpack executables, i.e., without executing them. While this is typically more time-consuming than unpacking a single executable, it can save a lot of time as soon as several files packed by an individual packer need to be unpacked. However, many runtimepackers

and executable protectors try to prevent these unpacking methods by
hardening their unpacking stubs against reverse engineering, e.g., by using antidebugging
and code-obfuscation techniques. Some of the commonly used unpacking
methods are: routine-based (static unpacking), emulator-based (dynamic unpacking)
and mixed routine-based and emulator-based unpacking.

2.6.1. Routine-based Unpacking

Routine-based unpacking is based on decompression algorithms. If the packer
author has used a standard compression algorithm then the decompression
algorithm can be applied to reverse packing. Examples of such algorithms are FSG
and UPack. Advantage of routine-based unpacking is high speed while disadvantage
is a lack of flexibility as every packer and each and every one of its variants would
require a separate unpacking routine. Some compression programs such as UPX
use their own decompression program included within them. Within a variety of all

packers, they are a minority. These stand-alone unpackers can be useful for
research purposes but they are not appropriate for incorporating into mainstream
antivirus products. Possibly, they could be a useful add-on for free antivirus solutions.
Single-purpose decompression program are the most frequently used method of
decompression of run-time packed files. This method works well when a packer uses
the same unpacking code each time. One possibility to write this decompression
program is to disassemble a sample of a packed file and just copy the bulk of the
resulting assembly language code and edit some parts of this code to be able to selfrun
(i.e., not run as part of compressed file) with a compressed file as input. The
resulting code will run quickly. However, the generated code will be processorspecific,
and therefore in order to run on many different processors it will need to be
patched to the correct form. Duplicating the original code is not a safe strategy. A
typical packer has minimal error checking. If code from this packer is used to unpack
the file, the product can overflow the memory allocated, which may eventually crash
the process, crash the entire system, or allow an attacker to gain a complete control
of the system. The solution of this most important problem is to encapsulate the code
within a defensive environment. Another solution would be to add many errorchecking
controls (buffer underflow/overflow, etc.). Since specific code is needed for
each packer and perhaps for each version of each packer, the size of an antivirus
product incorporating routine-based unpacking can grow quite large [17].

2.6.2. Emulator-based Unpacking

Emulator-based unpacking is based on loading and running a file in a virtual
environment. If file is stopped right at the original entry point, the unpacked file is
found. This technique is generic and can cope with different packers. Generic code
emulation is a very powerful decompression method. To program a code emulator is
not simple but once done, it can greatly speed up the process of adding new
unpackers. Often, part of the process of adding a new packer can be skipped,
because the emulator can decompress a file automatically. The code emulator loads
a program into a virtual environment and then runs until the file is decompressed – a
point which is defined heuristically, not algorithmically. How the heuristic defines a
state when a file is decompressed is the most difficult part of decompression by a
code emulator. Also, once an emulator is able to cope with anti-debugging tricks
used by one unpacker, it can cope with the tricks in PE-files compressed by any

other packer. Code emulation is slower than other decompression methods. This is
because the code emulator always has to maintain the entire CPU state.

2.6.3. Mixed Routine-based and Emulator-based

Since emulation based unpacking is too slow and static unpacking is too specific,
both can be combined and used as a hybrid approach. This technique combines the

efficiency of routine-based unpacking with the generic nature of emulator based unpacking. If the two types can work harmoniously, efficiency and flexibility can be attained. Routine-based unpacking can cope with the standard decompression algorithms while emulation can handle the modified packers or unknown packers. For particularly tricky packers, it is a good strategy to mix emulation and specific routines. For instance, emulation might take place until polymorphic encryption key has been found. Then the particular encrypted data can be decrypted by a specific routine. Its use is faster than emulation. Mixing code emulator with specific routines, however, brings additional complications [18].

2.6.4. Run and Dump Unpacking

Another technique that is quite successful in decompression of run-time packed executable files is running the code and then using a utility to capture the in-memory image and saving it on a disc. The main difficulty with this approach is finding out the right point when a running program is to be stopped and its image captured in the memory. The drawback of this method is that the executable must be loaded, which might not be acceptable in all cases as it cannot always be guaranteed that the program is terminated before any malicious functionality is executed. Besides using a virtual machine to avoid potential damage, it is also possible to unpack the executable by creating a separate tool out of the information gained from the unpacking routine included in the program.

With most packed programs, the first phase of execution involves unpacking the original program in memory, loading any required libraries, and looking up the addresses of imported functions. Once these actions are completed, the memory image of the program closely resembles its original, unpacked version. If a snap shot of the memory image can be dumped to a file at this point, that file can be analyzed

as if no packing had ever taken place. The advantage to this technique is that the embedded unpacking stub is leveraged to do the unpacking. The difficult part is determining exactly when to take the memory snapshot. The snap shot must be made after the unpacking has taken place and before the program has had a chance to cover its tracks. This is one drawback to this approach for unpacking. The other, perhaps more significant drawback is that the malware must be allowed to run so that it can unpack itself. To do this safely, a sandbox environment should be configured. Most operating systems provide facilities for accessing the memory of running processes.

The dumped process image is not executable till the executable’s header is corrected with the new values but the code itself is visible in its original form and disposed for reverse engineering and static analysis. This simple method works for most kinds of executable packers and encryptions, as the unpacking function typically extracts the complete program right at start, and does not interfere with later computations.

2.6.4.1. Debugger-Assisted Unpacking

Allowing malware to run freely is not always a great idea. If the executable file is unreliable and its exact function is not known, running it without any checks would give it the opportunity to wreak havoc before a dump of the memory image can be captured to disk to analyse the actual program code. Debuggers offer greater control over the execution of any program under analysis. The basic idea when using a debugger is to allow the malware to execute just long enough for it to unpack itself then utilizing the memory dumping capabilities of the debugger , the process image can be dumped to a file for further analysis. The problem with debugging is determining the stopping condition when the image of the process in memory is captured.

2.7. Identifying Packed Binaries

When having to analyse a large number of unknown binaries, it can be helpful to be able to determine whether a binary has been packed or not. There are heuristics that give good estimates whether a program contains hidden, packed code or not. Both packing and encryption transform one byte sequence into another, where the new

byte sequence typically has higher entropy than the original one (depending on the input data). This property can be leveraged to try to distinguish “regular" executables from packed or encrypted ones. Split each sample file into 256 byte sized blocks and record its average block entropies and its highest block entropy. Analysing the aggregated entropy scores statistically gives significant differences between packed and normal files leading to the conclusion that entropy metrics are indeed a valid heuristic to determine whether an executable has been packed or encrypted. Experiments show that a similar metric, based on the maximum entropy of sections within PE files, yields similar differences between unpacked and packed or encrypted files. It is possible to explicitly manipulate data within PE files to change their entropy. Random bytes can be added to increase a file’s entropy and the same byte can be added multiple times to lower it [14].

2.8. Published Unpacking Approaches

Various unpacking approaches have been published so far. Some of the published unpacking techniques are described in this section.

2.8.1. Malware Normalization

The proposed malware normalizer [19] is a system that takes an obfuscated executable, undoes the obfuscation and outputs a normalized executable. The technique uses code emulation to normalize executables. Unpacking consists of two basic steps. In the first step, the program to be normalized is executed in a controlled environment to identify the control-flow instruction that transfers control into the generated-code area. All writes to the code area are captured in this step. The second step uses information captured in the first step to construct a normalized program that contains the generated code.

2.8.1.1. Identifying the First Control Transfer into Generated Code

The first control transfer into generated code is identified by executing the program in an emulator, collecting all the memory writes (retaining for each address only the most recently written value) and monitoring execution flow. If the program attempts to

execute code from a memory area that was previously written, the target address of the control flow transfer is captured (i.e., the trigger instruction) and execution of code terminated. This technique is based on the assumption that the code generator and the instruction causing the control-flow transfer are reached in all program executions. By emulating the program and monitoring each instruction executed, the moment when execution reaches a previously written memory location can be identified. If heuristics are employed for determining this location, false negatives are generated.

2.8.1.2. Constructing a Non-self-generating Program

The captured data can be used to construct an equivalent program that does not contain the code generator. The data area targeted by the trigger instruction is replaced with the captured data. The memory write captured contain both dynamically generated code and the execution specific data e.g. the state of the program stack and heap. The executable file of the new program is a copy of the executable file of the old program with the byte values in the virtual memory range set from the captured data. The program location where execution was terminated is used as the entry point for the new program.

2.8.1.3. Limitations of Technique

This technique has several limitations. The unpacked executable is not ready-to-run. While all the code is present in the normalized executable, the imports table listing the dynamically linked libraries used by the program is not recovered, since most packing obfuscations replace it by a custom dynamic loader. This approach is open to resource consumption attacks and can have false negatives since the execution time in the sandbox often has to be heuristically restricted for performance reasons.

2.8.2. OmniUnpack

OmniUnpack [6] is a generic unpacking technique that incorporates a malware detector and is able to handle any type of packer and any type of self-modifying code. OmniUnpack monitors the program execution and tracks written as well as

written then-executed memory pages. When the program makes a potentially damaging system call, OmniUnpack invokes a malware detector on the written memory pages. If the detection result is negative (i.e., no malware found), execution is resumed. Code monitoring has been made efficient by tracking memory accesses at the page level (using non-executable pages or equivalent hardware mechanisms) instead of the instruction level. The resulting low overhead means that continuous monitoring can be deployed. Furthermore, OmniUnpack handles any number of unpacking and self modifying layers, each time communicating to the malwaredetection engine only the newly generated code that needs to be scanned.

The OmniUnpack algorithm follows a simple strategy to handle packed code. All memory writes and the program counter are tracked. If the program counter reaches a written memory address, it is identified that some form of unpacking or code generation occurs in the program. All written-then executed (or written-and-about-tobe-executed) memory locations should then be analyzed by a malware detector. The salient features of this approach are page-level tracking and continuous monitoring. These qualities as well as the disadvantages of OmniUnpack are touched upon in the succeeding subsections.

2.8.2.1. Page-level Tracking

Page-level tracking decreases the granularity of monitoring while greatly reducing the overhead of memory-access tracking. As a downside, it is less precise, often resulting in spurious detected unpacking stages. The spurious unpacking stages are caused by multiple layers of packing and by anti-disassembly and anti-static analysis techniques. Furthermore, code that executes from the same page on which it writes, even though non-self-modifying, also generates multiple spurious unpacking stages. It would be unnecessarily expensive to invoke the malware detector every time a written memory page is executed, because such an event (written-then-executed) is frequent. Written then-executed pages are indicative of unpacking but not indicative of the end of unpacking. The problem of determining when unpacking is completed can only be approximated. The end of an unpacking stage is approximated by using the heuristic that if the current execution trace indicates unpacking (i.e., memory pages were written and then executed), and if the program is about to invoke a dangerous system call, then it is assumed that an unpacking stage has completed and the malware detector is invoked. A dangerous system call has been defined as is

a call whose execution can leave the system in an unsafe state.

2.8.2.2. Continuous Monitoring

Because of the possibility of multiple unpacking stages and of the approximation used to detect them, it is insufficient to monitor and scan the program only once during an execution. OmniUnpack implements a continuous monitoring approach, where the execution is observed in its entirety.

2.8.2.3. Disadvantages of Approach

OmniUnpack raises performance concerns during the execution of benign programs. Furthermore, this approach is open to resource-consumption attacks and is prone to false negatives.

2.8.3. Pandora’s Bochs
The unpacker of Pandora’s Bochs [13] is based on ‘Bochs’ a portable x86 emulator. Bochs is a pure software virtual machine that is not subject to some of the designinherent flaws of reduced-privilege guest virtual machines for which well known and simple detection methods exists. There exist some methods to detect the presence of Bochs which are mainly due to errors in Bochs’s CPU implementation, not its architecture, and due to the possibility to fingerprint the emulated hardware. While the former can possibly be mitigated by fixing the emulation code, the latter is a problem of all virtual machines alike. Bochs provides a built-in mechanism for instrumenting code running on the emulated CPU. Major disadvantage of using Bochs is that it is quite slow. Also determining whether a program will unpack additional code and transfer control to unpacked code cannot be determined in the general case. It is however possible to estimate whether some monitored process is still showing any progress that might lead to the generation of new, unpacked code, or whether all monitored processes have reached a stable state. To that end, innovation is tracked for all monitored processes through memory writes versus branch targets, dynamic link libraries, execution of modified memory. Termination of the unpacking process is guaranteed by using an upper bound to the unpacker’s total

run time. The limitations to the proposed technique are slow speed, compatibility issues and a high number of malware samples failing to execute properly, despite being able to do so on other platforms. Import information is not always recovered correctly.
2.8.4. PolyUnpack
PolyUnpack [20] is a behaviour-based approach that uses a combination of static and dynamic analysis to automate the process of extracting the hidden-code. Hidden-code is automatically extracted based on the observation that sequences of unpacked code in a malware instance can be made self-identifying when the instance is executed in an environment with knowledge of the instance’s static code model. Starting with a malware instance, static analysis is performed to acquire a model of what its execution would look like if it did not generate and execute code at runtime. The statically derived model and the malware instance are then fed into the dynamic analysis component where the malware is executed in a sterile, isolated environment. The malware’s execution is paused after each instruction and its execution context is compared with the static code model. When the first instruction of a sequence not found in the static model is detected, representations of that unknown instruction sequence are written out and the malware’s execution is halted. PolyUnpack automates the process of extraction without requiring knowledge of how the malware unpacks its hidden-code.

PolyUnpack, like most instrumentation tools, is not transparent to the malware being processed. Therefore, there exists the possibility that an instance of malware being executed in PolyUnpack may detect that it is being instrumented and alter its behaviour (e.g., halting its execution instead of generating hidden-code) in order to evade extraction of its unpacked code.
2.8.5. Renovo
Renovo [21] is an emulation technique which monitors currently-executed instructions and memory writes at run-time. This approach maintains a shadow memory of the memory space of the analyzed program, observes the program execution, and determines if newly generated instructions are executed. Then it extracts the generated code and data.

After the packed executable starts, its attached decryption routine performs transformation procedures on the packed data, and then recovers the original code

and data. When the restoration completes, the decryption routine prepares the execution context for the original program code to execute, which includes initializing the CPU registers and assigning the program counter to the entry point of the newlygenerated code region. A packed executable may have multiple hidden layers, making it even more difficult to analyze. No matter what packing methods or how many hidden layers are applied, the original program code and data should eventually be present in memory to be executed, and also the instruction pointer should jump to the Original Entry Point (OEP) of the restored program code which has been written in memory at run-time. Taking advantage of this inevitable nature of packed executables, this technique has been proposed to dynamically extract the hidden original code and the OEP from the packed executable by examining whether the current instruction has been generated at run-time, after the program binary was loaded. For this purpose, the instruction pointer is monitored for jumps to the memory region which has been written after the program start-up. A memory map is generated when a program is loaded in memory and initialized as clean. Whenever the program performs a memory write instruction the corresponding destination memory region is marked as dirty, which means it is newly generated. Meanwhile, when the instruction pointer jumps to one of these newly-generated regions, it is determined that there is a hidden layer hiding the original program code, and identify the newly-generated memory regions to contain the hidden code and data, and the address pointed by the instruction pointer as the original entry point (OEP). To handle the possible hidden layers that may appear later on, the memory map is initialised as clean again after storing all the information extracted from the current hidden layer. The same procedure is repeated until time-out.
The emulated environment is not impervious to detection. The malicious code could measure elapsed time for certain instructions for which emulation these incurs high overhead, or check the results of certain instructions, because the results they generate are different under real and emulated environments.

CHAPTER 3

REQUIREMENTS

This chapter provides the description, functional and non-functional requirements and the constraints on SADE and has been supplemented with context mode and data flow diagrams. The contents given below were used as a guideline for designing the system.

3.1. The Overall Description

SADE will take a win32 portable executable file as input and will detect whether the file has been packed or not. The compressed (packed) file will be passed on to the main module of the software, the deobfuscation engine that will generically recover the original code that had been hidden by any packing obfuscation. The recovered data will finally be displayed in the output in a meaningful form (that should be of use and understandable to a security analyst) and the unpacked executable file will be the output of the software. The software will also display data about the file comprising of the modules loaded and resources handled etc. to facilitate the file analysis.

3.2. User Characteristics

The primary users of SADE will be security professionals. The software will extract the hidden code from a packed executable. When code is obtained from a compiled executable file, it is available in raw form (hex code or with further processing assembly language form) and requires that the analyst be familiar with the structure of the Windows Portable Executable (PE) files such as the portable executable file headers, file sections and data directories etc. The user should also have a rudimentary idea about how windows executable files are loaded on the windows platform and native windows functions are called to be able to understand and benefit from the output information from file analysis.

3.3. Constraints

- SADE might not work on every single type of packing obfuscation or on multiple layers of packing obfuscations.
- SADE will work only for Win32 PE files.
- The packed input executable file might contain code that can detect the presence of SADE and our software may or may not be able to handle it.

3.3.1. Assumptions and Dependencies

- SADE is being developed for Windows 32-bit platform.

3.4. Apportioning of Requirements

Requirements that may be delayed till future versions of the software are described here. These requirements have been delayed either because they are enormous undertakings on their own or because their implementation and integration into the system is not top priority and can wait till future versions of the application.

- Converting the unpacked executable file into a valid PE file that will be able to run on the windows platform can be performed in future work. This involves fixing the import section of the file and other modifications till it is fit to be run on the windows platform.
- Another requirement that can be catered for in future versions is the disassembly of the recovered code from the code section of the executable file into assembly language.
- Another possible extension to the software is to make it compatible on 64-bit windows platform.

3.5. Functional Requirements

Table 3.1 gives the functional requirements along with their descriptions.

Table 3.1 - Functional Requirements

	Requirement	**Description**
1	**Win32 Portable Executable Files**	SADE shall work only for Win32 Portable Executable files. The PE32 file given to the software as input might be packed or not, the software should be able to handle both cases accordingly.
2	**Detect Packing**	The program shall detect whether the input executable file is packed or not. The software shall detect the presence of compression and/or encryption without needing to discover the exact compression and/or encryption scheme used. Generic unpacking unpacks packed executable files independent of the packing technique used hence it is sufficient that the software is able to differentiate between a packed and an unpacked executable file. Only files that shall be found to have been packed will be passed to the generic unpacking routine so the process of detecting packing should be fool proof.
3	**Generic Unpacking**	The input executable file shall be generically unpacked i.e. some transformation or processing performed on it to uncover the original PE file code. The software shall uncover the original executable code without using any specific decompression and/or decryption techniques but rather provide a generic solution that should be one-fit-for-all i.e. it should work on a wide variety of packed files.

4 **Evade Running**
Harmful Code
The software shall not run the input PE32 file since the main
usage of the software will be analyzing hidden malicious code.
So harmful code should not be allowed to run.
5 **Information for File**
Analysis
The imports (modules loaded) and resources of the PE file shall
be shown in the output to help in file analysis.
6 **Dump of Recovered**
Code
The software shall create an image or dump of the recovered
PE file. The de-obfuscated code of the input PE32 file shall be
displayed for analysis purposes. The recovered executable file
containing the unpacked code may not necessarily be a running
version of the executable as emphasis is on the discovery of the
hidden code and making the hidden code available for analysis.

3.6. External Interfaces

Table 3.2 gives the external interface requirements.

Table 3.2 - External Requirements

Requirement Description

1 **Input** SADE will take a Win32 PE file as input. The input interface should be
able to display appropriate error message if wrong type of file is entered.
2 **Output** SADE will output a dump of the unpacked executable code if the file was
packed and for a PE32 file for which no packing is discovered,
information about the file such as the imports and resources should be
displayed for analysis. SADE will also display the modules loaded by the
file and resources and other structures contained in the file along with
the unpacked code in the output.

3.7. Non-functional Requirements

Following are the quality and performance characteristics that SADE must possess.
These requirements have to be testable just like the functional requirements.

3.7.1. Performance Requirements

Table 3.3 gives performance requirements for the system

Table 3.3 - Performance Requirements

1 Unpacking of input file should take around average 40 seconds and should not take more
than 90 seconds.
2 For a file in which no unpacking is detected, there should be no performance overhead
between inputting file and time taken to display PE file information.
3 SADE shall analyze only one file at a time.
4 Only one instance of SADE shall be run on any system at a time.

3.7.2. Quality Requirements

Table 3.4 gives quality requirements for the system.

Table 3.4 - Quality Requirements

Requirement Description

1
Reliability
SADE should be able to unpack 80% of input packed files. The
reliability of SADE is dependent on the complexity level of
packing and on the packer code being able to detect the
presence of any external software trying to force-unpack it.
2
Security
SADE should not compromise the safety of the system on which

it is being run by executing potentially malicious code. The
packed input file might be malicious as packing is used by
malware author to hide the true intent of their code so SADE
should not run any potentially harmful code.
3
Portability
SADE shall run on any architecture on which 32-bit Windows
platform is installed.

CHAPTER 4

SYSTEM DESIGN

4.1. Introduction

This chapter contains the design specification for SADE. The design of SADE has been described with different levels of abstraction in the subsequent sections.

4.1.1. Purpose

The design specification encapsulates the high level to the low level design covering all the aspects and levels of abstraction from the high level view of the project to the low level subcomponent details. Purpose of the document is to provide a detailed and unambiguous design which is coherent and consistent with the software requirements.

4.1.2. Scope

The design specification covers the overall design decisions and strategies, architectural design as well as low-level component design and abstract interface design. This chapter also covers the UML (Unified Modeling Language) diagrams and the Graphical User Interface design of SADE.

4.2. System Overview

SADE is an executable analysis toolkit that generically detects and unpacks a packed windows executable file (PE32 file) and makes the data hidden by encryption and compression available for analysis purposes. The unpacked executable file is not a valid windows executable as the import address table needs reconstruction (which is out of the scope of our project) but the unpacked code and other information about the file such as the modules and resources loaded by the

executable are available through the toolkit. The toolkit is invaluable to security analysts as their time is expensive and individual malware samples can take hours to analyze and manual unpacking is a tedious and error prone process.

4.2.1. Input

The input to the system is a PE32 executable file. The toolkit first checks the validity of the file. If the file is a valid 32-bit windows executable, it is loaded in memory and important file information, such as header information, section details and modules loaded by the file are extracted from the executable.

4.2.2. Major Processes

The first step in generic unpacking is to detect if an executable is packed. This saves processing and time spent in trying to unpack an already unpacked or normal executable that does not have any transformations such as compression or encryption performed on it. Once packing is detected the Generic Unpacking Engine is triggered, which uses file dumps and statistical analysis of those dumps to unpack hidden program code.

4.2.3. Output

The outputs produced by the system are PE file information, dump of the recovered code and result of statistical analysis in tabular and graphical form. The PE file information includes information gathered from the headers of the file such as the names of the Windows APIs called by the executable, the size and address of different sections in the executable and information about directories like the debug directory etc.

4.3. Design Considerations

This section describes some of the issues which need to be addressed or resolved before attempting to devise a complete design solution.

4.3.1. Assumptions and Dependencies

i) SADE works only for 32 bit windows portable executable files.
ii) SADE might not work on every single type of packing obfuscation or on multiple layers of packing obfuscations. The packed input executable file might contain code that can detect the presence of SADE and SADE may or may not be able to handle it.
iii) Disassembling module for the executable will not be implemented. A disassembler may be imported from an external source and integrated with the system to produce a more helpful and refined output.
iv) The application is designed for the Windows platform.
v) The primary users of SADE are security professionals. The software provides unpacked code from a packed executable. When code is obtained from a compiled executable file, it is available in raw form (hex code or with further processing assembly language form) and requires that the analyst be familiar with the structure of the windows Portable Executable (PE) files and also with how windows executable files are loaded on the windows platform to be able to understand and use the output information in file analysis.
vi) Possible and/or probable changes in functionality

4.3.2. General Constraints

Global limitations or constraints can have a significant impact on the design and development of any software system. The constraints and limitations that may affect SADE are listed and described briefly in the consequent subsections (the list is *not* exhaustive).

4.3.2.1. Hardware and Software Environment

The input executable file might be able to detect the debugging environment in which it is being unpacked and upon detecting a third party software trying to monitor and control its behavior, the input executable file can change its execution by for instance going into an infinite loop or by halting its execution altogether. The design of the system should try to evade this possibility or incase of its occurrence inform the user with an appropriate message.

4.3.2.2. End-user Environment

The software design only concerns 32-bit windows platform as the end user environment. Porting the software for 64-bit platform can be considered in future work.

4.3.2.3. Standards Compliance

SADE should comply with the latest standard for portable executable and common file format files released by Microsoft.

4.3.2.4. Security Requirements

The system has to avoid running harmful code as input executable file might be a hidden malicious program.

4.3.2.5. Memory Limitations

The software environment to be designed and developed for SADE should not over burden the system it is being run on or exhaust the system memory.

4.3.2.6. Performance requirements
SADE should enable the smooth operation of other software on the end user system it is being run on i.e. it should not overload the system and affect its overall performance.

4.3.3. Goals and Guidelines
The goal of this software application is to provide a professional toolkit which helps the security analysts analyze the packed executables and help in the detection of malicious software and the identification of signatures for malware. The design decisions have been made on the following principles:

i) Simple and unambiguous design graphically demonstrated so that it is conceivable and understandable.

ii) Extensible design that can smoothly incorporate future developments without requiring major paradigm changes. Object-Oriented design principles make this job easier and manageable.

iii) Optimal design based on best possible tradeoff between speed and memory usage.

iv) Preference to Security in the design i.e. the security of the system is of prime importance since the system will primarily deal with malicious executables.

v) Research incentive in the project was a major motivational factor and goal. The major incentive behind the project was to supplement and implement the latest research in the field of packed malware successfully and efficiently so that the end product is state-of-the-art and comparable with its contemporaries.

4.4. Unpacking Methodology
The unpacking methodology adopted for SADE is run and dump unpacking approach using a debugger. Run and dump unpacking does not have any of the limitations or problems of using virtual machines and emulators. The approach is based on the innate property of encrypted and compressed executables that regardless of the packing technique applied to the original program the original code or its equivalent will eventually be present in memory and get executed at some point at run-time. By taking advantage of this intrinsic nature of packed executables, one could potentially extract the hidden binary code or its equivalent as a raw memory dump. However, it is not clear which regions in the memory contain the hidden binary and when is the right time to dump such regions, i.e., when the execution context jumps to the hidden original code. The essential problem with this straightforward and simplistic approach of figuring out when to stop the run and dump process has been approximated by using several statistical and heuristic measures.

4.5. Architectural Strategies
Following are the design decisions and strategies that affect the overall organization of the system and its higher-level structures.

i) In order to provide debugging capabilities to run and dump a packed file, the windows debug API in kernel32.dll has been used.

ii) Open source and free ware code libraries for debugging have been employed to help construct the debugger for the generic unpacking module.

iii) Visual C# is the programming language used. The C# paradigm is helpful in constructing the graphical user interface and providing file reading, writing and data marshalling capabilities.

iv) In the future work, the software can be upgraded with a disassembling module to improve the readability of the unpacked code.

4.6. System Architecture

SADE has two principle responsibilities. First is to detect whether an input executable file is packed or not and the second is to unpack an executable that is found to be packed. The Unpacking task additionally has two separable responsibilities. The unpacking task incorporates a debugging module which runs the unpacking stub of the packed input executable to enable the recovery of original input file code. The unpacking job also requires a statistical analysis engine that is the system core where heuristic decisions are made on when to stop the debugging process. The debugging and analysis engine requires a dumping module to take snapshots of the process in memory as the state of the input executable changes while the process runs in the debugger. SADE has an auxiliary responsibility to extract information from the input executable file that supplements the unpacked code for analysis purposes. The decomposition of the system has been based on these singular responsibilities. The high level architecture of SADE is depicted in figure 4.1.

Figure 4.1 – Architecture of SADE

SADE has been divided into its respective components on the basis of required functionality. An executable file is presented as input to the system. The **Dump** module has the capability to take a dump of a process or file in memory. Dumping a file is the act of copying raw data from one place to another with little or no formatting for readability. The **Dump** module provides the data in hexadecimal and ASCII formats. A static dump of the executable file is taken and passed to the **Packing Detection** module. When provided with a valid input executable file, packing detection module detects whether the file is packed or not. The output of **Packing Detection** is to give a Boolean value indicating whether the file appears to have some packing transformation performed on it or not. If the input file is **not packed,** it does not need to be unpacked and can be sent straight to the **File Info Retrieval** component. The **File Info Retrieval** module extracts the portable executable file information such as the PE file headers and the PE sections etc. In the scenario where the file is found to be **packed** it is sent to the **Generic Unpacking Engine**. The Generic Unpacking module has further two distinct sub-modules: **Debugger** and **Statistical Analysis**. The **Debugger** module provides debugging functionalities that are required for the unpacking procedure. The **Statistical Analysis** component uses various statistical heuristics like entropy of the file being progressively debugged to provide a stopping criteria that decides when the executable file has been unpacked completely and to stop debugging the executable. The **Dump** module interacts with **Generic Unpacking** to capture changes in the file as it is executed and the unpacking routine in the packed file runs to uncover the original program code that was obscured by packing. The **Generic Unpacking** runs the executable in the **Debugger** in a debug loop that executes a small portion of the debugee (program being debugged) program at a time till the stopping criteria has been met. Once the **Generic Unpacking Engine** has recovered the unpacked file, it is sent to **File Info Retrieval** to extract PE file information that is further helpful to security analysts for file analysis. The output from SADE is the **unpacked executable** and **file info**.

4.7.UML Diagrams

This section covers the UML diagrams for SADE. This includes the use case diagram of the system, the class diagram and the interaction diagrams for the different scenarios that can occur in the system. The interaction diagrams cover both sequence and communication diagrams. Furthermore, the data flow diagrams of the different modules of the system are presented in this section.

4.7.1. Use Case Diagram

The use case diagram of SADE, figure 4.2, describes all the functionalities that the user will be able to avail through the use of the application. The user can load any PE32 file and can use the system to discover if the input file has been altered with some packing transformation or not. The user can also retrieve portable executable file information; which includes information about the modules loaded as well as section and header information. The user can also retrieve the dump of both packed and unpacked versions of the file. The dump of original file has section and header information while the dump of unpacked file can be viewed in ACSII and Hexadecimal format. Further future work includes providing disassembled unpacked code. The users of the system have been classified as novice users and security analysts. A novice user of the system is someone who has nominal understanding of the executable files and file analysis. Security analysts are familiar with executable files and can use the system for malicious software analysis to create malware signatures etc.

Figure 4.2 – Use Case Diagram of SADE

Use Case Description

Table 4.1 describes the details of the class diagram in figure 4.3. Descriptions are given for each use case of figure 4.3.

Table 4.1 - Use Case Description

Use Case	**Description**
Load Executable File	User browses through the end-system to select an input executable file.
Get Packing Detection Information	SADE informs user whether the input executable file has some packed obfuscation performed on it or not.
Unpack Executable	User prompts SADE to unpack or recover the hidden original program code.
Dump Original File	Dump of the input executable file in its original form. This is also referred to as a static dump.
Get File Information	User can get the portable executable information from the system.
Get File Header Information	Information contained in the portable executable file header. Includes file signature, size etc.
Get File Section Information	Information about the portable executable file's sections. These can include the data, code and resource section etc.
Get Modules Loaded Information	Information about the modules that is external resources imported by the executable file. These include the Windows API Dynamic Link Library (.DLL) files. This is useful for file analysis as different modules loaded correspond to different system-level functionalities and can reflect on the tasks performed by the system like using network resources etc.
Get Unpacked File Dump	Dump of the unpacked file. This is the dump of the code that was hidden by the packing transformation and was recovered on runtime and captured by our software.
Hex Dump	Dump of executable file in hexadecimal format.
ASCII Dump	Dump of executable file in ASCII character set.

4.7.2. Class Diagram

The class diagram shown in figure 4.3 is the visual representation of the object oriented model for SADE. It includes the classes described in table 4.2.

Figure 4.3 – Class Diagram of SADE

Table 4.2 - Class Diagram Description

Class	Description
Packing Detection	This class contains the functionality for identifying whether an input file is packed or not. Provides a function that returns true if packing is found.
Unpack	The unpack class contains the debugging code that runs the input executable till complete dump is found.
Dump	Class encapsulates the byte image of entire file. Provides function to view file dump in different formats.
Analysis	Used by unpack class to detect when to stop the run-and-dump process i.e. the point where the hidden program code is completely dumped.
Debug Process	File containing variables such as process id assigned by windows and functions such as setting breakpoints on a line of code on the process loaded in memory being debugged (called the “debugee”).
Debug Event	An event that is raised by the debugger each time a debugging event occurs for instance a module is loaded etc.
Kernel32.dll	Windows API used for debugging tasks. Contains functions to start and suspend threads, create a new process, read process memory etc.
File Info	Class that extracts the portable executable file information such as headers, sections, file directories and resources etc.
Debug Module	Class that stores information for each individual module imported by the debug process.

4.7.3. Interaction Diagram

Interaction diagram includes both sequence and communication diagrams. Following interaction diagrams will define the detailed object design of SADE. There are two possible scenarios for a file. The input executable file could be a packed file or in the second case it could be a benign or normal file with no packing obfuscation. The behaviour of the system is different for the two scenarios.

4.7.3.1. Interaction Diagrams for Packed File Scenario

i) The user opens the application to see the input window.
ii) User selects executable through the “browse” window to load executable.
iii) Now user can view the static dump of the file.
iv) System uses static dump to generically detect some packing transformation on the file.
v) If packing is found, system unpacks it generically and returns the dump of unpacked file.
vi) The user is now able to retrieve information of original file.

Figure 4.4 shows the sequence diagram for the “packed file” scenario and figure 4.5 shows the communication diagram for the “packed file” scenario.

Figure 4.4 – Sequence diagram for scenario where the input file is packed

4.7.3.2. Interaction Diagrams for Normal (Benign) File Scenario

i) The user opens the application to see the input window.
ii) User selects executable through the “browse” window to load executable.
iii) Now user can view the static dump of the file.

iv) System uses static dump to generically detect some packing transformation on the file.

v) In case of no packing detected, no further processing is required on the file.
vi) File information can be directly retrieved now without needing to pass the executable through generic unpacking engine.
Figures 4.6 and 4.7 show the sequence and communication diagram for the case where the input executable is not packed.

Figure 4.5 – Communication diagram for scenario where input file is packed

Figure 4.6 - Sequence diagram for scenario where input file is not packed

Figure 4.7 - Communication diagram for scenario where input file is not packed

4.7.4. Data Flow Diagrams

This sub-section contains data flow diagrams for all the processes in SADE. The data flow diagrams of SADE have been divided into two parts. One is for packing detection and unpacking (figure 4.8) and the second is for retrieving file information (figure 4.9).

4.7.4.1. DFD of Packing Detection and Unpacking

As shown in figure 4.8, the flow of data in SADE starts from the input "Browse" window where the user specifies the path of the input executable file. The path is used to open the executable file in memory i.e. loading the executable file. A static dump of the file is taken i.e. dump of the executable file in its original form. Once the debugging process starts, the unpacking stub attached by the packer to the input executable file is run and the unpacking stub recovers the actual or real program in memory at runtime. The static dump of the input executable file is passed to the packing detection module where the decision is made about whether the file is packed or not. If the input file is found to have been altered by some compression or encryption transformation (i.e. packing), it is sent to the debugging module. If no packing is detected, the original file dump requires no further processing and is sent

to the PE file information retrieval module, where file information is extracted from the executable file which is useful to security analysts for file analysis. If packing was found in the file, the entry point of the executable is taken from the optional header of the process (file loaded in memory) and a breakpoint is set on it. From then onwards, the debugger goes into an unpacking loop where the file is run for a very short interval and then the process execution is halted, latest dump of the file is taken from memory and statistical analysis is performed on it. The results of statistical analysis are taken by debugger to check for stopping criteria i.e. the point at which the actual program code has been completely uncovered and the original entry point is found. The original entry point is the entry point of the executable that was packed and a packing stub attached to it. Once the unpacking process completes and the debug loop breaks when stopping criterion has been met, the PE file information is retrieved and the unpacked code with additional file information is shown as output.

Figure 4.8 – Data flow diagram of packing detection and unpacking

4.7.4.2. DFD of PE File Information Retrieval

The data flow of the information retrieval module has been shown separately in figure 4.9, as the information retrieval process is independent of the identification and unpacking process. When a file is sent to the information retrieval process, the PE file signature is checked. This confirms that the input file is actually a 32 bit portable

executable file. If the PE signature is not found an error is shown and program terminates. The PE file header which includes the section table is extracted next. The section table contains the starting address and size of all the sections present in the executable file. The section headers are displayed. Each section entry is used to retrieve the entire section and then section information is displayed.

Figure 4.9 – Data flow diagram of PE File Information Retrieval

4.8. Detailed Subsystem Designs

This section describes in detail, the design of components of SADE.

4.8.1. Generic Unpacking

The sub-components of Generic Unpacking Engine have been described in this subsection.

4.8.1.1. Debugger

The debugger module provides various debugging utilities that are instrumental to the unpacking process as the unpacking principal is based on running the input executable to the point that the unpacking routine attached by the packer has completely recovered the original program code and then halting program execution

before the unpacked code has chance to run. In this regard, the debugger provides the facility to create a suspended process or a process with debugger attached to it. The debugger for SADE has the following major responsibilities:

i) Create a suspended process or create a process with debugger attached to it.

ii) Suspend and resume process thread during execution.

iii) Raise a debug event each time a debugging event happens such as a module is loaded, a new thread is created or a child process is created, a debug exception is raised.

iv) Set and reset breakpoints on any portion of the code.

v) Get thread context for the process. Thread context is a structure provided by windows that contains the current values of all the registers such as instruction pointer, stack pointer etc. Thread Context can only be taken when the thread whose context is being taken is suspended.

vi) Set thread context of the process being debugged. This involves changing values of the registers and raising bit flags such as the Int3 flag to cause a breakpoint. The Get and Set thread functionality should be used with utmost care.

4.8.1.2. Statistical Analysis

Statistical Analysis is the brain center of Generic Unpacking Engine. Statistical Analysis is required to determine the point where the debugger should halt execution of the executable being debugged; this point is generally referred to as the **Stopping Condition**. Statistical analysis assimilates the stopping criteria to detect when the stopping condition has been met. Statistical Analysis can use various heuristic and statistic measures in the stopping criteria.

4.8.1.2.1. Entropy

Change in entropy is the primary criterion to detect when the stopping condition has been met. "***Entropy is the measure of redundancy in the file***". A file can have byte entropy in the range 0 to 8. Packed files are less redundant and have higher entropies. An unpacked file or a file that has not been compressed or encrypted has a more uniform distribution of data, has a higher amount of redundancy and has less entropy than a packed file. Entropy calculation involves finding the occurrence of the data elements in the file i.e. the histogram.

Entropy of a file dump containing data with values X in the range $\{x_0, x_1, \ldots\ldots. x_n\}$ is where $x_i = \{x_0, x_1, \ldots\ldots. x_n\}$,

p(x) = count of occurrence of x / length of data segment and b=2.

The following variations of entropy have been used as statistical metrics:

i) ***File Entropy*****:** Entropy of the entire Portable Executable (PE) file.

ii) ***Block Entropy*****:** The block entropy is not a conclusive measure for statistical analysis and determining the stopping condition. It does not show an identifiable pattern or change in the progressive dumps of the file being debugged. Block entropies have been used to find an identifiable pattern in the changing entropies as the file executes, more of the unpacked code is recovered and the redundancy in the file changes. Block entropy is calculated by dividing the file into 256 or 1024 byte blocks and then calculating the entropy of each block.

iii) *File Section Entropy*: Another useful entropy measure is the entropy of the portable executable file sections. A PE file can have several different sections like the code section, data section and resource section. Entropy of the code section can be viewed in isolation as that is the part of the PE file where the original program code is restored by the unpacking routine generated by the packer.

iv) *Portable Executable File Header Entropy*: Every portable executable file has a header that contains information about the location and size of all the sections and other necessary information required by the operating system to run the file. The entropy of the file header is yet another measure that is part of the stopping criteria.

4.8.1.2.2. Checksum

A checksum is a form of redundancy check, a simple way to protect the integrity of data by detecting errors in data. It works by adding up the basic components of a message, typically the assorted bits or bytes and storing the resulting value. Some interesting checksum properties are: Two or more blocks which are very similar at binary levels have very close checksum values. The simplest form of checksum

which simply adds up the assorted bytes in the data can help classify when the blocks of data have changed.

4.8.2. Packing Detection

The packing detection module can use various metrics to decide whether an input file is in normal state or it has been altered by some packing transformation.

4.8.2.1. PE Header, Code, Data, File and Block Entropies

The encrypted code of a packed application is usually stored in a code or data section of the PE file (A section can be identified as a code section if the Executable section flag is set, otherwise it can be considered as the data section). As the code of the program is usually somehow encrypted, it will look like “random”, loosely speaking. On the other hand non encrypted code sections contain well “structured” information, namely the opcode of executable instructions and the memory location of the operands. Non-encrypted data sections also contain somehow structured information. Following this observation, the byte entropy of the code and data sections in the PE file can be measured. If the entropy of a section is close to 8, which is the maximum byte entropy, the section likely contains encrypted code. The code and data sections are not the only places where the executable packing tool may hide the code of the original application. There are parts of the PE header dedicated to optional fields that are not necessary for the correct loading of the program into memory by the operating system. Some packing tools may therefore hide encrypted code in those unused portions of the PE header. For this reason it is useful to measure the byte entropy of the PE header as well. Considering that the PE file is quite complex and contains other such unused spaces, the entropy of the PE file as a whole is also taken into account. If a portable executable file is found to have average and maximum entropy above a respective threshold, it can be classified as packed.

4.8.2.2. Number of Entries in the IAT (Import Address Table)

The Import Address Table (IAT) of an executable contains the address of the external functions called by the application. These external functions are imported from Dynamic Linked Libraries (DLL). Each imported function has an address in the

IAT which is written by the operating system loader after the application is launched and the PE file is mapped into memory. Every time the application calls an external function, the IAT is queried in order to resolve its address in memory. Most nonpacked executables import many external functions. For example, they usually import many functions from the native Windows API, which are used to read/write from and to files, open new windows on the screen and manage a network connection and so on. Therefore, the IAT will usually contain many entries, one per each imported function. On the other hand, packed executables often import very few external functions. The main reason is that the unpacking routine does not need many external functions. The basic operations the unpacking routing performs are read and write memory locations in order to decrypt the code of the packed application on the fly. For example, no window on the screen or network operation is usually needed. This is reflected in a small number of entries in the IAT of a packed executable. Hence, the number of entries in the import address table is an excellent criterion for deciding whether a file has been packed or not.

4.8.2.3. Number of Standard and Non-Standard Sections

Normal portable executable files usually contain a well defined set of standard sections. For example, applications compiled using Microsoft Visual C++ usually contain at least one code section named .text, and two data sections named .data, and .rsrc section. On the other hand, packed executables often contain code and data sections which do not follow these standard names. For example, the UPX executable packing tool usually creates a PE file that contain two sections named .UPX0 and .UPX1, respectively, and a section named .rsrc. Hence the number of standard and non-standard sections can be used as an auxiliary criterion for detecting packing.

4.9. Graphical User Interface (GUI) Design

This section describes the interface of Software Analysis & De-obfuscation engine (SADE).

4.9.1. Design Principles

This section describes the design principles for the interface of SADE.

4.9.1.1. Usability

Interface design emphasizes clarity and ease of use. Microsoft Visual Studio 2005 has been used primarily with the intent to implement the latest GUI standards.

4.9.1.2. Improved User Experience

The simplified application architecture increases the user experience by removing all confusion and complications. This process prevents user error and enhances perceived product value.

4.9.1.3. Enhanced Design

The engaging design process renders the technology more attractive to its target audience because its appearance is inviting, clean and easy to use.

4.9.1.4. Coherence

Interface is "intuitive", for the technical operators who are familiar with the information security terms, or more precisely packing terminologies. The behavior of the program is designed to be internally and externally consistent. In other words logical, consistent, and easily followed. Internal consistency means that the program's behaviors make "sense" with respect to other parts of the program. For example, if one attribute of an object (e.g. color) is modifiable using a pop-up menu, then it is to be expected that other attributes of the object would also be editable in a similar fashion. One should strive towards the principle of "least surprise". External consistency means that the program is consistent with the environment in which it runs. This includes consistency with both the operating system and the typical suite of applications that run within that operating system. One of the most widely recognized forms of external coherence is compliance with user-interface standards.

4.9.1.5. State Visualization

Changes in behaviour will be reflected in the appearance of the program in the form of graphs and dumps. It is important that this internal state be visualized in a way that is consistent, clear, and unambiguous.

4.9.1.6. Shortcuts

System will provide both concrete and abstract ways of getting a task done. Once a user has become experienced with an application, he/she will start to build a mental model of the application. She will be able to predict with high accuracy what the results of any particular user gesture/command will be in any given context. Hence, the program's attempts to make things "easy" by breaking up complex actions into simple steps may seem cumbersome.

4.9.2. Software Features

The salient features of the GUI of SADE have been described in this subsection.

i) PE file information Retrieval
ii) Generic Unpacking engine
iii) Memory Dump- HEX and ASCII Dump
iv) Assembly Code View
v) Display of Statistical Matrices
vi) User Help

Support

SADE requires .net platform installed on the system. Pre-requisite of installing and running the application on any system is that it should have the .net framework 3.5 installed.

4.9.3. Interface Design Model

Figure 4.10 shows the overall layout of the system, with screens shown as boxes and navigation path shown as arrows.

The screens or windows in the interface design model are ‘Main Window’, ‘Statistical Details’, ‘File Information’, ‘Generic Unpacking’ and ‘View Dump’. The ‘Main Window’ is the GUI that shows when the application is opened. All other windows are navigated though it. ‘‘Statistical Details’, ‘File Information’ and ‘Generic Unpacking’ can be opened through the ‘Main Window’. The ‘View Dump’ window can be navigated through ‘Generic Unpacking’ screen.

Figure 4.10 – Interface Design Model of SADE

Figure 4.11 – GUI of PE File Header Information

Figure 4.12 – GUI of PE Sections and Details

4.9.3.1. Generic Unpacking

The window in Figure 4.13 is the GUI for Generic Unpacking of the PE File. The user selects and loads the file. System will then provide the user to view the static dump as well as the unpacked code of the file.

Figure 4.13 – GUI of Unpacking Details

4.9.3.2. File Dump

The system displays the dump of unpacked file in ASCII and Hexadecimal formats as shown in figure 4.14.

Figure 4.14 – GUI of ASCII and Hex Dump

4.9.3.3. Statistical Details

The system displays the statistical results of both packed and unpacked file in graphical form as shown in figures 4.15 and 4.16. This will help the user to compare the data of both files easily.

Figure 4.15 – GUI of Entropy Results
62
Figure 4.16 – GUI of Statistical Result
63

CHAPTER 5

IMPLEMENTATION

This chapter contains the implementation details of SADE. Each SADE module will be elaborated separately in this chapter (without any programming jargon) with the help of block diagrams. The overall algorithm of SADE is described in figure 5.1. The major decisions that the system has to make have been shown along with a high level view of the tasks that it has to perform.

Figure 5.1 – Algorithm (High Level View) of SADE

The subsequent sections provide implementation details for each SADE module.

5.1. Packing Detection

The first and foremost task that SADE has to perform when presented with a valid 32-bit input executable file is to send it to the Packing Detection module. Packing detection generically establishes whether the input executable is packed or not i.e. it is not concerned with the packer and packing algorithm used to pack the executable but simply finds out whether any brand of packing transformation has been applied on the file. SADE unpacks the packed executables generically so the details of the
64
specific encryption and/or compression algorithms used to pack the executable are redundant. The packing detection results should show a high level of accuracy as only those files which are found to be packed by the packing detection module are sent to the Generic Unpacking Engine. If the Packing Detection fails to detect a packed file, then it will not be unpacked by Generic Unpacking and the goal of the project to automate the process of malware analysis will fail. Packing detection is deduced through static analysis while Generic Unpacking works on the principle of dynamic analysis. Various metrics were considered for packing detection before settling for the block entropy of the static dump of input executable file as primary criterion.

5.1.1. Entropy

The concept of information is too broad to be captured completely by a single definition. However, for any probability distribution, a quantity called *entropy* can be defined which has many properties that agree with the intuitive notion of what a measure of information should be**.** Entropy is a measure of the average uncertainty in a random variable. It is the number of bits on average required to describe the random variable. Entropy is measured in bits by taking logarithms to the base 2. If 'X' is a discrete random variable then the set X = {$x_0, x_1, \ldots\ldots, x_n$} then probability mass function p(x) = P_r\{X Є x\} [22]. The entropy of a random variable X with a probability mass function p(x) is denoted by

For the purpose of detecting packing, the entropy of the entire file as a whole was taken but it did not provide any discernible information i.e. there was no detectable difference between the file entropies of packed and normal files which could be made a decisive factor for detecting packing. Next, the entropies of the executable file header and all the file sections were taken separately but they also failed to contribute to the static analysis for packing detection criteria. Finally, the block entropy was gauged as the principle metric for the packing detection criterion. For calculating block entropies, different block sizes were considered such as 128-byte blocks, 256-byte blocks and 1024-byte blocks. The size of one block was chosen to be 256-byte after experimental evaluation and comparison. The next subsection describes the packing detection criterion using 256-byte blocks.

5.1.2. Packing Detection Criterion using Block Entropy
Block entropy is determined by dividing the static dump of the input executable file into 256-byte blocks and then calculating the entropy of each block. A histogram for the block entropies is maintained. Byte entropy has the range 0 to 8. The histogram shows the frequency of byte entropies. Normal files were found to have block entropies less than 7 while packed files lean towards block entropies above 7. The frequency of blocks having entropy in the range 0 to 7 are classified as normal or unpacked blocks while number of blocks having entropy above 7 are labelled as packed blocks. The ratio of packed to unpacked blocks is then used as the deciding factor for determining whether the input file has some packing obfuscation performed on it or not. Packing results were studied for a multitude of files packed with different packers and an experimental threshold was discovered that provides accurate packing detection for all files. If the ratio of packed blocks in an input executable is found to be greater than 20%, then the file is classified as packed.
5.1.3. Number of Entries in the IAT (Import Address Table)
Number of entries in the import address table is used as the second determining factor in packing detection. Through experimentation and study of a vast array of packed files, a general rule was deduced about packed files that the packed files always have less than 30 entries in the import address table while normal files have entries far beyond this number. Packed files have very few entries in the import address table because packing tends to hide the actual import address table of a program in order to hide the structure of the file being packed; only those entries are available which are required by the unpacking stub attached by the packer. The unpacking stub performs basic read, write and allocate memory operations so the entries in the import address table of packed files never exceeds a certain threshold which was experimentally found to be 30.
Block Entropy and number of entries in the Import Address Table collectively serve as the Packing Detection criterion used in the packing detection module. Figure 5.2 shows the algorithm for SADE that detects whether the input executable has been modified with any encryption and/or compression transformations or not. The outcome of the algorithm is either ‘packing detected’ or ‘no packing detected’.

Figure 5.2 – Algorithm for Packing Detection
Input Executable
Static Dump of Input Executable
Find First 256-Byte Block
Calculate Entropy of Block
Update Frequency Histogram
There exists next
256-Byte block
in static dump?
Calculate Entropy of Block
Update Frequency Histogram
Yes
Calculate Ratio of
Packed Blocks using
Frequency Histogram
Ratio of
Packed Blocks
> 0.2

Number of
Entries in
IAT < 30
Packing Detected
No Packing
Detected
No
Yes
Yes
No
No

5.2. Generic Unpacking

The Generic Unpacking Engine takes the input executable files that are detected as packed by Packing Detection and unpack them using the debugger. The Generic Unpacking Engine can be divided into two distinct modules. The Debugger module loads the packed executable in a carefully contained environment and the Statistical Analysis module determines the stopping point when the unpacking process is estimated to have completed its task. The algorithm is shown in flowchart in figure 5.3.

Figure 5.3 – Algorithm (High Level View) for Generic Unpacking

5.2.1. Debugging

Debuggers exist primarily to assist software developers with locating and correcting errors in their programs but they can also be used as powerful reversing tools. In modern operating systems debuggers can be roughly divided into two very different flavours: *user-mode debuggers* and *kernel-mode debuggers*. User-mode debuggers are the more conventional debuggers that are typically used by software developers. As the name implies, user-mode debuggers run as normal applications, in user mode, and they can only be used for debugging regular user-mode applications. Kernel-mode debuggers are far more powerful. They allow unlimited control of the target system and provide a full view of everything happening on the system, regardless of whether it is happening inside application code or inside operating system code.

5.2.1.1. User-Mode Debugger

For SADE, a user-mode debugger has been implemented using the Win32 Debugging API. User-mode debuggers are conventional applications that attach to another process (the *debuggee*) and can take full control of it. The debugger and the debuggee (the process being debugged) have a parent-child relationship, if the debugger created the process. If the debugger is "attached" to an already executing process, then the debugger and the debuggee are independent and the debuggee can detach from the debugger without terminating. User-mode debuggers have the advantage of being very easy to set up and use, because they are just another program that is running on the system (unlike kernel-mode debuggers). The downside is that user-mode debuggers can only view a single process and can only view user mode code within that process. Being limited to a single process is not a problem for SADE because it will be dealing with one packed executable at a time. Being restricted to viewing user-mode code is not a problem unless the product being debugged has its own kernel-mode components (such as device drivers) which will not be the case for SADE normally as it will be debugging executable applications. When a program is implemented purely in user mode there is usually no real need to step into operating system code that runs in the kernel. Beyond these limitations, some user-mode debuggers are also unable to debug a program before execution reaches the main executable's entry point (this is typically the .exe file's WinMain callback). This can be a problem in some cases because the system runs a

significant amount of user-mode code before that, including calls to the DllMain callback of each DLL that is statically linked to the executable [23]. This is not a problem for SADE either because it is able to detect when a DLL is loaded and raises a debug event for it and halts the debugger execution. Furthermore, process information is useful to have while debugging. There is an endless list of features that could fall into this category, but the most basic ones are a list of the currently loaded executable modules and the currently running threads, along with a stack dump and register dump for each thread.

5.2.1.2. Win32 Debug API

The Windows API are Microsoft's core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems. Almost all Windows programs interact with the Windows API; a small number (such as programs started early in the Windows startup process) use the Native API. The Win32 debug API provides services over which a native code debugger can be built. It provides functionality to load a program for debugging (or attach to an existing program). Information of interest about the process being debugged can be obtained. Win32 API offers services that provide notifications when debugging-related events are generated in the debuggee process or thread starting or exiting, DLLs being loaded or unloaded etc. The debug API also allows reading from and writing to the debuggee memory and instruction stream. The steps to debugging are explained in the subsequent sections [24].

5.2.1.2.1. Create Debuggee Process or Attach to Existing Process

The Debugger can create the debuggee (process being debugged under the debugger) as a child process using the CreateProcess(...DEBUG_ONLY_THIS_PROCESS…); API call or the debugger can attach to an existing process by using DebugActiveProcess(ProcessID); function. If the process being debugged is created as a child process it will exist only when linked to its parent process and terminate when the debugger terminates but if the debugger attached to an already existing process, then the attached process can exist independently once it is detached. SADE typically creates a child process for debugging.

5.2.1.2.2. Continue Execution

The debugee process is created in suspended state and waits for the debugger to allow continue execution by calling ContinueDebugEvent(…);. The debugee process will then continue execution till the next debug event occurs. The debugger can go into a debug loop where it waits for debug events to occur, handles each event accordingly and keeps executing till the stopping criteria is met. When the debugger is processing debug events it has full control over the debuggee. The Operating System stops all debuggee threads and does not schedule them until the debugger says so. Every time a debug event occurs, the debugger halts program execution. The ContinueDebugEvent(DBG_CONTINUE); is called to continue execution.

5.2.1.2.3. Debug Events

Windows defines several debug events that are fired during the lifetime of the debuggee. The debugger goes into waiting after the WaitForDebugEvent(..) API is called. This debugging function can be passed time in micro seconds as argument and the debugger will timeout after the allotted time whether a debug event has occurred or not. The debugger waiting in the debug loop is notified of these events. The debug events being used by SADE are Create Process Debug Event, Create

Thread Debug Event, Exception Debug Event, Exit Process Debug Event, Exit Thread Debug Event, Load DLL Debug Event, Unload DLL Debug Event and Output Debug String Event. They are summarized in table 5.1.

5.2.1.2.4. Read and Write Memory

Reading from and writing to a debuggee process's memory space is supported through the **ReadProcessMemory** and **WriteProcessMemory** API functions (e.g. modifying debuggee code). The ReadProcessMemory is used to get an image (dump) of the process in memory.

5.2.1.2.5. Get CPU Registers and Set Breakpoints

The Win32 debug API can be used to get or set the current context or CPU registers using GetThreadContext(…) and SetThreadContext(…) API calls. The debug API

services can be further extended to set breakpoints. Debuggers use breakpoints extensively behind the scenes to control their debuggees (e.g. while stepping over a function call, "running to cursor", or to Break execution). A breakpoint corresponds to a "breakpoint instruction", the instruction mnemonic is 'INT 3' on the Pentium (0xCC is the OpCode).

Table 5.1 - Debug Events

Debug Event	**Description**
Create Process Debug Event	It is the first event generated by the kernel for a process just before it begins executing in user-mode. The Create Process Debug Event indicates that the process was loaded and not executed.
Create Thread Debug Event	This event is generated whenever a new thread is created in a process being debugged.
Exception Debug Event	Generated whenever an exception occurs in the process being debugged. Examples of exceptions include breakpoint exception, single stepping code exception, illegal memory usage etc. The Exception Debug Event is fired before the first instruction of the process is executed – called the initial breakpoint.
Exit Process Debug Event	Fired when a process exits.
Exit Thread Debug Event	Fired whenever a thread that is a part of the process being debugged exits.
Load DLL Debug Event	Fired each time the debuggee loads a DLL. Can be used by the debugger to load the symbol table corresponding to the DLL.
Unload DLL Debug Event	Fired whenever a process unloads a DLL. Can be used to unload corresponding loaded symbol tables.
Output Debug String Event	Fired in response to the debuggee making the OutputDebugString API call.

5.2.2. Statistical Analysis

Generic Unpacking Engine carries out statistical analysis in conjunction with debugging the packed executable to determine the point at which the unpacking stub attached to the process has finished unpacking the original code in memory. It is not possible to know this exact location therefore it is approximated using heuristics and statistical analysis. The chief heuristics used for detecting the stopping condition of the debugger are described in the rest of the section.

5.2.2.1. Code Section Entropy

Change in code section's entropy is used as a primary metric for detecting the stopping condition. Change in the entropy of the code is indicative of the end of an unpacking stage. These entropy values are calculated each time the debugger halts and whenever a change is detected, the stopping criteria is compared with the process's current state to determine whether unpacking is finished and debugging can be halted or to carry on debugging and wait for next change in section entropy.

5.2.2.2. Block Entropy

Each time the debugger is halted when a debug event occurs or the debugger times out, snapshot of the debuggee is taken and block entropies are calculated for the memory dump similar to the process carried out for Packing Detection. The memory dump is broken into 256-byte blocks, entropy calculations are carried out for each block and then the frequency of byte entropies is determined to find the ratio of packed and unpacked blocks in the memory dump. The stopping criteria is said to be met when the ratio of packed blocks in the memory dump is less than 10%.

5.2.2.3. Checksum

Another useful metric for determining end of unpacking is checksum. As the compressed code is decompressed in memory, the checksum of the file is increased. The change in checksum can be compared against a pre-defined threshold to support the detection of stopping condition.

5.2.2.4. ASCII String Literals

Packing tends to mask all the string literals in an executable file. As the unpacking routine uncovers the hidden code in memory, the string literals inside the executable are unmasked. The stopping condition can be determined by tracking the number of string literals inside the executable file loaded in memory as the unpacking progresses.

5.3. Portable Executable File Information Retrieval

The file information retrieval module uses the structure definitions and details provided in the Microsoft Portable Executable and Common Object File Format Specification to extract executable file structures which contain useful information. The retrieved structures include the Image Header, Optional Header, Section Headers, Import Address Table, Debug Directories etc.

First the file header is initialized and the module can check the PE file signature to determine that the input executable file is indeed a 32-bit windows executable. The optional header contains a field that specifies the number of section headers that follow the optional header. Each section header is read which contain fields indicating the relative virtual address of the section and size of the section etc. Each section is read and the directories that it contains are retrieved from it. The import address table is read which is hierarchal i.e. each entry in the table points to an entry for a DLL and that entry in turn points to all the functions called by that particular DLL. The import address table is traversed to find the name of each DLL and the names of all the functions called by that DLL till the end of the import address table is reached.

5.4. Entropy and Statistical Information Graphs

The entropy of the original file as well as the entropy of the final unpacked dump and other statistical information such as file entropy, section entropy, average entropy and checksum etc. is displayed to the user as graphs, bar charts and pie charts using the services provided by “ZedGraph.dll”.

CHAPTER 6

SOFTWARE TEST PLAN

Approved by

Supervisor Date
Evaluation Panel Date
Evaluation Panel Date
Evaluation Panel Date

[Other] Date

Project Information

Project Release Information

Table 6.1 - Project Release Information

Project Name	Software Analysis and De-obfuscation Engine
Project Code	SADE
Project Release	Version 1.0
Project Release Date	6th August, 2009
Testing Dates	1st July, 2009 to 25th July, 2009
Testing Iteration Number	1
Project Modules Information	Software/ Application

Project Team Information

Table 6.2 - Project Team Information

Group Leader	Faiza Khalid
Developers	Komal Babar, Faiza Khalid
Quality Analyst	Abdul Wahab
Tester	Nauvera Rehman

6.1. Introduction

This test approach document describes the appropriate strategies, process, workflows and methodologies used to plan, organize, execute and manage testing of software project SADE.

6.1.1 Scope

In Scope: *Test Plan* defines the unit, integration, system, regression, and acceptance testing approach. The test scope includes the following:

- Testing of all functional, application performance and use cases requirements listed in the *Use Case* document.
- Quality requirements.
- End-to-end testing and testing of interfaces of all system components that interact with the SADE.

Out of Scope: The following are considered out of scope for SADE system Test Plan and testing scope:

- Functional requirements testing for systems outside SADE
- Testing of Business SOPs, disaster recovery and Business Continuity Plan.

6.1.2. Quality Objective

6.1.2.1 Primary Objective

A primary objective of testing application systems is to: ***assure that the system meets the full requirements, including quality requirements (non-functional requirements), fits metrics for each quality requirement and satisfies the use case scenarios and maintains the quality of the product.*** At the end of the project development cycle, the user should find that the project has met or exceeded all of their expectations as detailed in the requirements.

Any changes, additions, or deletions to the Requirements Document, Functional Specification, or Design Specification will be documented and tested at the highest

level of quality allowed within the remaining time of the project and within the ability of the test team.

6.1.2.2. Secondary Objective

The secondary objective of testing application systems will be to: ***identify and expose all issues and associated risks, communicate all known issues to the project team, and ensure that all issues are addressed in an appropriate manner before release.*** As an objective, this requires careful and methodical testing of the application to first ensure that all areas of the system are scrutinized and, consequently, all issues (bugs) found are dealt with appropriately.

6.2. Assumptions and Constraints for Test Environment

Below are some minimum assumptions:

 For User Acceptance testing, the Developer team has completed unit, system and integration testing and met all the requirements (including quality requirements) based on Requirement Traceability Matrix.
 User Acceptance testing will be conducted by the supervisor and evaluation panel.
 Test results will be reported on daily basis. Failed tests and defect list with evidence will be sent to developer directly.
 Use cases have been developed for User Acceptance testing. Use cases are approved by the evaluation panel.
 Test results are developed and approved periodically by the Team Lead.
 Test Team will support and provide appropriate guidance to supervisors and developers to conduct testing.
 Testers should clearly understand on test procedures and recording a defect or enhancement. Testing Team will schedule meetings with Developers and supervisors to train and address any testing related issues.
 Developer will receive consolidated list of request for test environment set up, data set, defect list, etc.
 Developer will support ongoing testing activities based on priorities.
 Test scripts must be approved by Test Lead prior to test execution.

 Test team is responsible to identify dependencies between test results and submit clear request to set up test environment

6.3. Test Methodology

6.3.1. Overview

The purpose of the Test Plan is to achieve the following:

 Define testing strategies for each area and sub-area to include all the functional and quality (non-functional) requirements.
 Divide Design Specification into testable areas and sub-areas (do not confuse with more detailed test specification). Be sure to also identify and include areas that are to be omitted (not tested).
 Define bug-tracking procedures.
 Identify required resources and related information.
 Provide testing Schedule.

6.3.2. Usability Testing

The purpose of usability testing is to ensure that the new components and features will function in a manner that is acceptable to the customer.

Development will typically create a non-functioning prototype of the UI components to evaluate the proposed design. Usability testing can be coordinated by testing, but actual testing must be performed by non-testers (**as close to end-users as possible).** Testing will review the findings and provide the project team with its evaluation of the impact these changes will have on the testing process and to the project as a whole.

6.3.3. Performance Testing

Performance of SADE was evaluated by two ways. First, known executable files were packed using a variety of different packers (using their default options) and an attempt was made to uncompress them using SADE. The unpacking results were then compared to the original binary. Secondly, some packed copyrighted

executables were downloaded from the internet and the unpacking results were examined to gauge their quality.
The size of the smallest executable tested on the system was 15KB and the largest file tested on the system was 3MB.

6.3.3.1 Synthetic Samples

A test set of packed binaries was generated from two different executables. The first one was Glow.exe which is a very small file of 15 KB and the second one was notepad.exe, a text editor that comes with the default installation of Windows XP. It is a fairly small executable of 69120 bytes.

6.3.3.2. Packers

Following is the list of packers used on the test files for packing them.

- UPX 3.01w, a free and open source, cross-platform runtime packer, currently available at http://upx.sourceforge.net/
- ASPack 2.12, a commercial runtime packer. An evaluation version is currently available at http://www.aspack.com/
- ASProtect 1.35, a commercial executable protector. An evaluation version is currently available at http://www.aspack.com/
- eXpressor 1.5.0.1, a commercial executable protector. An evaluation version is currently available at http://www.cgsoftlabs.ro/
- EXECryptor a freeware executable packer available at www.freedownloadscenter.com/...Tools/EXECryptor.html
- PE Pack 1.0 a freeware packer downloaded from internet
- RLPack a trial version available at http://rlpack.jezgra.net
- PETITE 2.2 trial version available at http://wareseeker.com/Utilities/petite-2.2.zip/57874
- PETITE 2.3 trial version available at http://www.wareseeker.com
- Winupack 3.9 packer trial version available on internet
- XPack available at http://www.eurodownload.com/downloadsoftware/...../Download W32/

- PE Compact packer free trial version available at http://www.softpedia.com/get/Programming/Packers-Crypters-Protectors/PECompact.shtml
- BeRoExePacker Packer trial version available at http://leechermods.blogspot.com/2008/02/exe-packer-collection-3-bymodssubcc.html

Obtaining this collection of runtime packers was not an easy task. Some packers were only available for a fee or as demo versions that require user interaction before they transfer control to the original executable, some had no easily locatable “official" home where they could be downloaded from (the reason for including URLs in the above list), and many packers failed to create working executables from the chosen sample executables.

6.3.3.3. Packing Detection

There is no overhead involved in packing detection, as static analysis of file is done which is quite instant. The detection time for various executables is shown in table 6.3. The packing detection time for all input executable files is always same because the packing detection is performed through static analysis of the input file. The static packing detection time was found to be 15 milliseconds.

6.3.3.4. Generic Unpacking

Table 6.4 lists the detailed results from executing each synthetic sample under SADE. For each packer it was noted whether it could generate an executable file from the original image and whether the packed executable required user interaction to complete unpacking (some demo versions of commercial packers did so). Then, the details of the analysis run are listed.

- The **termination reason** can be either *automatic* for SADE's successful halt of executable before its hidden code executes and *liveness* when SADE terminated because it could not detect any progress, or *abnormal* termination if for example the file detects the presence of SADE and terminates itself.

- The **correct unpacking** column shows whether the SADE was able to show correct unpacking results or not after unpacking routine has completed its operation.
- The **unpacking time** column contains the absolute run time of SADE for a given sample.

Table 6.3 - Packing Detection Results

Executable Name	**Packer used**	**Packing Detection Speed (millisecond)**	**Correct detection**
Glow.exe	UPX	15 (0.0156250 sec)	✓
Aspkgen.exe	UPX	15	✓
Noepad.exe	ASPack	15	✓
Notepad.exe	ASProtect	15	✓
Notepad.exe	BeRoExPacker	15	✓
Notepad.exe	ExeCryptor	15	✓
Notepad.exe	PEPack	15	✓
Notepad.exe	Pe Compact	15	✓
Notepad.exe	Petite	15	✓
Notepad.exe	Expressor	15	✓
Notepad.exe	XPack	15	✓
Notepad.exe	-	15	✓
Glow.exe	-	15	✓
InternetExplorer.exe	-	15	✓

Table 6.4 summarizes the results of SADE on the synthetic samples.

Table 6.4 - Generic Unpacking Results

Packer	**Packed file size(KB)**	**Termination reason**	**Correct Unpacking**	**Unpacking time (millisecond)**
UPX	32	Automatic	✓	484
ASPack	52.5	Automatic	✓	468
ASProtect	328	Abnormal	✓	750
BeRoExPacker	35.5	Automatic	✓	390
Execryptor	98.0	Automatic	✓	156
PEPack	98.0	Automatic	✓	343
PE Compact	48.5	Automatic	✓	531
Petite 2.2	39	Automatic	✓	453
Petite 2.3	38.5	Automatic	✓	421

Expressor	56.8	Automatic	✓	531
XPack	45.7	Automatic	✓	625

- **Output similarity** measure was used in which the original executable dump was compared to the data that the unpacking result contains. Figure 6.1 to 6.7 containing the graphs of entropy distribution show the results for files tested in table 6.4:

Figure 6.1 – Notepad.exe (not packed)

Figure 6.2 – AsPack (unpacked file by SADE)

Figure 6.3 – BeRoExPack (unpacked file by SADE)

Figure 6.4 – ExeCryptor (unpacked file by SADE)

Figure 6.5 – PE Compact (unpacked file by SADE)

Figure 6.6 – Expressor (unpacked file by SADE)

Figure 6.7 – XPack (unpacked file by SADE)

6.3.3.5. Unknown Packed Samples (Original Files Not Available)

The packed samples of table 6.5 were run on the system for which original executable files were not available. Hence the accuracy of results is measured in terms of *strings available in output dump*. This criterion is useful, since all the ASCII strings which will finally be available in file at runtime can be found in the dump as well (if dump is of unpacked file). One of the main characteristics of executable encryption and compression is that it hides all the string literals in the executable. As a packed executable is unpacked and the hidden code becomes available, the number of string literals in the unpacked code rises. This trend has been used to test accuracy of results of files for which the unaltered executables were not available for comparison.

Table 6.5 - Unknown packed samples' results

File name	**Packed file size (KB)**	**Termination reason**	**strings available in output**	**Unpacking time (millisecond)**
Aspkgen.exe	193	Automatic	✓	218
Keygen.exe	83.5	Automatic	✓	296
GLOW.exe	15.0	Automatic	✓	281
PETGUI.exe	54.2	Automatic	✓	953
Unlocker.exe	42.0	Automatic	✓	421
USBVaccine.exe	392	Automatic	✓	250
BatteryDoubler.exe	833	Automatic	✓	734
BitDefenderRemoveTool.exe	195	Automatic	✓	828

6.3.4. Testing Completeness Criteria

The milestone target is to place the release/application (build) into production after it has been shown that the system has reached a level of stability that meets or

exceeds the client expectations as defined in the Requirements, Functional Specifications and Design document.

6.4. Test Levels

Testing of an application can be broken down into three primary categories and several sub-levels. The test categories and test levels are defined in the subsequent subsections.

6.4.1. Build Tests

6.4.1.1. Level 1 - Build Acceptance Tests

The application was tested to check if it can be built and installed successfully on different computers. If any Level 1 test case fails, the build is returned to developers un-tested. The system specifications and environment on which the tests were carried out are shown in table 6.6

Table 6.6 - Build Test Specifications

S#	System Specifications	Windows
1	Intel CPU, 1.73 GHz, 760 MB RAM	Windows XP 2002
2	Intel Core Duo CPU, 1.60 GHz, 1.75 GB RAM	Windows XP (Service Pack 3)
3	Intel Core 2 Duo CPU, 2.0 GHz, 2 GB RAM	Windows Vista (Service Pack 2)

6.4.1.2. Level 2 - Smoke Tests

These test cases verify the major functionality at high level. The objective is to determine if further testing is possible. These test cases emphasize breadth more than depth. All components have been touched, and every major feature has been tested briefly by the Smoke Test. If any Level 2 test case fails, the build is returned to developers un-tested.

Level 2a - Bug Regression Testing: Every bug that was "Open" during the previous build but marked as "Fixed, Needs Re-Testing" for the current build under test, will need to be regressed, or re-tested. Once the smoke test is completed, all resolved bugs need to be regressed.

Bug Regression is a central tenant throughout all testing phases. All bugs that were resolved as "Fixed, Needs Re-Testing" were regressed when testing team was notified of the new drop containing the fixes.

When a bug passes regression it will be considered "Closed, Fixed". *When a Severity 1 bug fails regression, adopters Testing team also puts out an immediate email to development.* The Test Engineer is responsible for tracking and reporting to development team and Team Lead the status of regression testing.

6.5. Deliverables Matrix

Table 6.7 provides the list of artifacts that are process driven and produced during the testing lifecycle. This matrix has been updated routinely throughout the project development cycle in Test Plan.

Table 6.7 - Deliverable Matrix

Deliverable
Documents
Test Plan
Test Schedule
Test Case / Bug Write-Ups
Test Cases / Results
Reports
Test Results Report

6.6. Test Environment

6.6.1. Hardware

The Hardware used for testing the system was:

- Intel ® Core Duo CPU
- 1.60 GHz

- 2 GB RAM

6.6.2. Software

The following list of software is the minimum requirement for testing the system

- Windows NT
- MS Office 2000+ Professional
- Task Manager (Testing Tool Server)
- PEiD (freeware commercial tool for testing PE file)

6.7. Bug Severity and Priority Definition

Bug Severity and Priority fields are both very important for categorizing bugs and prioritizing if and when the bugs will be fixed. The bug Severity and Priority levels will be defined as outlined in the following tables below. Testing will assign a severity level to all bugs. The Test Lead will be responsible to see that a correct severity level is assigned to each bug.

6.7.1. Severity List

The severity levels of the detected bugs are given in table 6.8. It includes severity ID, severity level and severity description.

6.7.2. Priority List

Table 6.9 gives the priority list of the detected bugs.

Table 6.8 - Severity List

Severity ID	Severity Level	Severity Description
1	Critical	The module/product crashes or the bug causes nonrecoverable conditions. System crashes, GP Faults, file corruption, or potential data loss, program hangs requiring reboot are all examples of a Severity 1 bug.
2	High	Major system component unusable due to failure or incorrect functionality. Severity 2 bugs cause serious problems such as a lack of functionality, or insufficient or unclear error messages that can have a major impact to the user, prevents other areas of the app from being tested, etc. Severity 2 bugs can have a work around, but the work around is inconvenient or difficult.
3	Medium	Incorrect functionality of component or process. There is a simple work around for the bug if it is Severity 3.
4	Minor	Documentation errors or signed off Severity 3 bugs.

Table 6.9 – Priority List

Priority ID	Priority Level	Priority Description
5	Must Fix	This bug must be fixed immediately; the product cannot ship with this bug.
4	Should Fix	These are important problems that should be fixed as soon as possible. It would be an embarrassment to the project team if this bug shipped.
3	Fix When Have Time	The problem should be fixed within the time available. If the bug does not delay shipping date, then fix it.
2	Low Priority	It is not important (at this time) that these bugs be addressed. Fix these bugs after all other bugs have been fixed.
1	Trivial	Enhancements/ Good to have features incorporatedjust are out of the current scope.

6.8. Test Personnel

The test personnel and their respective responsibilities are given in table 6.10.

Table 6.10 - Test Personnel

Parties	Contact Person	Roles and Responsibilities
DS	Aisha Khalid	Overall supervision Work stream management

Review of Test Plan and Test Cases
Monitor testing schedule and procedure
Test Team Lead Komal Babar QA team Lead
Development of Test Plan and Test Cases
Managing and directing testing activities
To ensure testing activity comply with project
requirements and test plan
Test Engineer Nauvera Rehman Conduct testing
Develop test cases and conduct testing
Submit Bug Reports

6.9. Test Schedule

The test schedule is given in table 6.11. It documents the start and end dates of testing for the different SADE components as well as the number of days spent on testing for each module. The requirements document and execution of test cases took the longest time in the testing process while testing the design document and walkthrough of test plan took the least time.

Table 6.11 - Test Schedule

Task	**Start Date**	**End Date**	**Days**
Document Requirement	01-07-09	05-07-09	5
Document Design	06-07-09	07-07-09	1
Create Test Cases	08-07-09	10-07-09	2
Conduct walk-through of Test Cases and Test Plan	10-07-09	11-07-09	1
Test Case Execution	12-07-09	17-07-09	5
Regression Testing	17-07-09	21-07-09	3
Sign off on Test Results	30-07-09	30-07-09	2

6.10. Test Cases

This section contains the test cases for all the modules of the application. Each test case has been given a test case number and has certain preconditions. The name of the tester and date of testing have also been documented.

6.10.1. Module information

Test cases were written for each module of the system. Table 6.12 contains the names of the modules for which test cases have been generated.

Table 6.12 - Module Information

Description	The software comprises of various modules integrated together.
Modules/ Test Components	1. Packing Detection 2. Generic Unpacking (Stopping Criteria) 3. PE File Information 4. User Interface

6.10.2. Test case table

Tables 6.13 to table 6.20 describe the test cases for testing the application.

Table 6.13 - Test Case: Packing Detection

Test case Name: Packing Detection	
Test case Number	FN-REQ-02-PackD0001
Precondition	Visual studio 2005 must be installed PE 32 Input file must be selected.
Procedure	Test all options -> Click browse button -> Input executable file -> Compare the packing detection result with PEiD results

for the same executable
Expected Result Packing detection result matches PEiD Results.
Actual Result Packing detection result matches PEiD Results.
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes
throughout debugging>
Tester Komal Babar
Date 18-06-09
Remarks <Additional remarks>

Table 6.14 - Test Case: Generic Unpacking
Test case Name: Generic Unpacking
Test case Number FN-REQ-03-Unpack0001
Precondition Visual studio 2005 must be installed
PE 32 Input file must be selected.
Packing has been detected.
Procedure Test all options
-> Click browse button
-> Input executable file
-> Click 'Yes' on message box which pops up and ask user
permission to extract hidden file.
-> Compare the output file dump with Packed input file
dump.
Expected Result Meaningful strings found in output dump reflecting
unpacked file UI and string outputs of executable.
Actual Result Meaningful strings reflecting file menu and other UI
features found in output.
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes
throughout debugging>
Tester Faiza Khalid
Date 18-06-09
Remarks <Additional remarks>

Table 6.15 - Test Case: PE File Information
Test case Name: PE File information
Test case Number FN-REQ-05-PEInf0001
Precondition Visual studio 2005 must be installed
PE 32 Input file must be selected.
Procedure Test all options
-> Click browse button
-> Input executable file
-> Click on "Header information" in file Info menu
-> Click on "COFF header" button
-> Click on "Optional Header" button
->Click on "Section Header" button
-> Click on "Data directories" button
-> Click on "Section information" link in file Info menu
->Click on Tree-view Sections to see section details
Expected Result All fields correctly match with PEiD results for file info.
Actual Result All fields correctly matched with PEiD results.
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes

throughout debugging>
Tester Nauvera Rehman
Date 18-06-09
Remarks <Additional remarks>
95
Table 6.16 - Test Case 1: Graphical User Interface
Test case Name: Graphical User Interface
Test case Number FN-REQ-04-GUI0001
Precondition .net platform must be installed
Procedure Test all options in left panel (menu) and related controls
-> Click browse button
-> Input executable file which is not packed.
Expected Result -> The Packing Detection icon turns green and title bar displays "File is not packed"
-> Menu options "runtime details" and "other statistics" disabled.
Actual Result -> Packing Detection icon turned green and title bar displayed "File is not packed"
-> "runtime details" and "other statistics" links disabled.
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes throughout debugging>
Tester Abdul Wahab
Date 18-06-09
Remarks <Additional remarks>
96
Table 6.17 - Test Case 2: Graphical User Interface
Test case Name: Graphical User Interface
Test case Number FN-REQ-04-GUI0002
Precondition .net platform must be installed
Procedure Test all options in left panel (menu) and related controls
-> Click browse button
-> Input executable file which is packed.
-> A message box prompts if "you want to extract the hidden code". Click No
Expected Result -> The Packing Detection icon turns red and title bar displays "Packing detected"
-> Menu options "runtime details" and "other statistics" disabled.
Actual Result -> Packing Detection icon turned red and title bar displayed "Packing detected"
-> "runtime details" and "other statistics" links disabled.
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes throughout debugging>
Tester Abdul Wahab
Date 18-06-09
Remarks <Additional remarks>
97
Table 6.18 - Test Case 3: Graphical User Interface
Test case Name: Graphical User Interface
Test case Number FN-REQ-004-GUI0003
Precondition .net platform must be installed

Procedure Test all options in left panel (menu) and related controls
-> Click browse button
-> Input executable file which is packed.
-> A message box prompts if "you want to extract the
hidden code". Click Yes
Expected Result -> The Packing Detection icon turns red and title bar
displays "Packing detected"
-> All menu options are enabled.
Actual Result -> Packing Detection icon turned red and title bar
displayed "Packing detected"
->All menu options were enabled.
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes
throughout debugging>
Tester Abdul Wahab
Date 18-06-09
Remarks <Additional remarks>
98
Table 6.19 Test Case: Packed Blocks Graph
Test case Name: Packed Blocks Graph
Test case Number FN-REQ-000-Graph0001
Precondition .net platform must be installed
Input file must be loaded.
Procedure Test all options for graph display of packed blocks
-> Click the link "Packed Blocks" on left panel.
Expected Result -> The Entropy graph is displayed correctly
Actual Result -> The Entropy graph displayed correctly
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes
throughout debugging>
Tester Komal Babr
Date 18-06-09
Remarks <Additional remarks>
99
Table 6.20 - Test Case: Graph Other Statistics
Test case Name: Graph-Other statistics
Test case
Number
FN-REQ-004-Graph0002
Precondition .net platform must be installed
Packed file is loaded and unpacked by system
Procedure -> Click on "Other statistics" link on left panel
Expected Result -> Line graph is displayed showing statistical analysis
Actual Result -> Line graph is displayed showing statistical analysis
Status Pass
Bud ID <In case the test fails, specify a bud ID to track its changes
throughout debugging>
Tester Komal Babar
Date 18-06-09
Remarks <Additional remarks>
100

6.11. Test Results Report

This subsection summarizes the test results.

 All software modules preliminary execution shows desired results
 All modules integration testing reflect no problem
 Desired results of packing detection and generic unpacking algorithms demonstrate improved values of parameters.
 The mean values of the results are given in table 6.21.

Table 6.21 - Test Results Report

Packing detection speed	15 millisecond
Packing detection rate	100%
Generic unpacking speed	470 millisecond
Similarity of output dumps with original file (comparing entropy results).	78%
PE file information accuracy	100%

CHAPTER 7

USER GUIDE

This chapter provides a walkthrough for the users of SADE. The subsequent sections explain how to use SADE from start to finish with the help of screen shots.

7.1. Starting SADE

SADE has been developed for the Windows platform and requires 32-bit portable executable files as input. Start SADE by double-clicking the SADE shortcut icon on your Windows desktop or through the Start Menu. Figure 7.1 shows the main screen of the program. This is the first window that opens when SADE is started.

Figure 7.1 – Starting SADE

7.2. Loading the Executable

Whenever you open this toolkit, you will select input file by clicking on “Browse” button. Load your required executable through the browser window that will pop up. Figure 7.2 shows the location of the “Browse” button and figure 7.3 shows the browser window that navigates the path of the executable file that the user wants to load.

Figure 7.2 – “Browse” Button

Figure 7.3 – Selecting Input File

Once a file is selected, an alert box will appear with a message mentioning whether the file is packed or not. If the file is found to be packed, user is queried if he/she wants to extract the hidden code or not. Figure 7.4 shows the state of SADE when the input executable file is encrypted and compressed while figure 7.5 is screenshot of SADE when the input executable is benign or normal.

Figure 7.4 – Packing Detected

Figure 7.5 – Packing Not Detected

7.3. Viewing PE File Information

After allowing the application to extract hidden code; the user will be able to view all the static and dynamic information of the file. Static file information contains:

- o Header Information
- o Section Information
- o Static Dump

7.3.1. Header Information

In this window, all PE file header are displayed accordingly. It begins with File

Header that consists of a COFF file header and an optional header.

7.3.1.1. COFF (Common Object File Format) Header

The COFF header describes the type of target machine, size of section table and creation date and time of the file etc. Figure 7.6 shows the 'COFF Header' tab.

Figure 7.6 – View COFF Header

7.3.1.2. Optional Header

Every image file has an optional header that provides information to the loader. This header is optional in the sense that some files (specifically, object files) do not have it. For image files, this header is required. The optional header magic number determines whether an image is a PE32 (32-bit) or PE32+ (64-bit) executable. Figure 7.7 shows the 'Optional Header' tab in the 'PE File Header Information' window pane.

In this portion, following sections of the file are displayed:

 The size of the code (text) section or the sum of all code sections if there are multiple sections.

105

 The size of the initialized data section, or the sum of all such sections if there are multiple data sections.

 The size of the un-initialized data section, or the sum of all such sections if there are multiple BSS sections.

 The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.

Figure 7.7 – View Optional Header

7.3.1.3. Section Header

Each row of the section table is, in effect, a section header. This table immediately follows the optional header, if any. This positioning is required because the file header does not contain a direct pointer to the section table.

In this portion following sections of the file are displayed:

 Name of the section; an 8-byte, null-padded UTF-8 encoded string.

 The total size of the section when loaded into memory.

 Virtual address of the executable i.e. the address of the first byte of the section relative to the image base when the section is loaded into memory.

106

 The size of the section (for object files) or the size of the initialized data on disk (for image files).etc

 The file pointer to the first page of the section within the COFF file.

 The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.

 The file pointer to the beginning of line-number entries for the section.

 The number of relocation entries for the section. This is set to zero for executable images.

 The number of line-number entries for the section.

Figure 7.8 shows the 'Section Header' tab in the 'PE File Header Information' window pane.

Figure 7.8 – View Section Header

7.3.1.4. Data Directories

Each data directory gives the address and size of a table or string that Windows uses. These data directory entries are all loaded into memory so that the system can use them at run time. A data directory is an 8-byte field. This portion contains information about the import table i.e. its size and address. Figure 7.9 shows the 'Data Directories' tab in the 'PE File Header Information' window pane.

107

Figure 7.9 – View Data Directories

7.3.2. Section Information

This portion of file information retrieval displays different sections of the import table of the file. The import directory table contains address information that is used to resolve fix up references to the entry points within a DLL image. The import directory table consists of an array of import directory entries, one entry for each DLL. The last directory entry is empty (filled with null values), which indicates the end of the directory table.

Each import directory entry contains following Information:

o DLL Name

o Time and date stamp

o Name RVA (The address of an ASCII string that contains the name of the DLL) etc.

Figure 7.10 shows the Import Address Table as shown in the 'PE Sections and Details' window of SADE.

108

Figure 7.10 – View Import Address Table (IAT)

For each DLL, the details of APIs called by it are displayed accordingly as shown in Figure 7.11.

Figure 7.11 – View APIs Called

7.3.3. Static Dump

In this section, the static dump of the executable is displayed in ASCII and Hex format. This dump is taken without running the file. Figure 7.12 shows the window that displays the dump in ASCII format.

109

Figure 7.12 – Static ASCII Dump

Figure 7.13 shows the window that displays the Hex representation of the static dump.

Figure 7.13 – Static Hex Dump

7.4. Viewing Runtime Details

This portion describes the runtime details of the executable. The runtime details are gathered during the unpacking process so the runtime or dynamic details are only available for those files that are classified as packed. For normal or benign executable files, only the information gathered through static analysis is available. For packed files, dynamic analysis is performed and the gathered data is presented to user in useful format such as tables and graphs. These runtime details include

110

o Debug Events

o Modules Loaded

o Unpacked File Dump

Figure 7.14 shows the 'Runtime Details' window of SADE. The Process ID of the input executable when it was loaded in memory is also shown here.

Figure 7.14 – Runtime Details

7.4.1. Viewing Debug Events

Figure 7.15 shows the 'Debug Events Details' window of SADE.

Figure 7.15 – Debug Events

111

In this section, the details of debug events raised by the process being analyzed are shown in the table. The entries of the tables are

o Event (name of the debug event)

o Address (the memory address of the raised event)

- Source (The source of the raised event)

7.4.2. Viewing Modules Loaded

In this section the modules loaded by the executable for its execution are displayed. Each module has a base address, entry point and size etc. Figure 7.16 shows the 'Debug Events Details' window of SADE.

Figure 7.16 – Modules Loaded

7.4.3. Viewing Unpacked File Dump

This section displays the dump of complete code in ASCII and Hex format. This memory dump is larger in size than the static dump. Figure 7.17 shows the window of SADE that displays the unpacked file dump which is available in ASCII as well as Hexadecimal representation.

Figure 7.17 – Unpacked File Dump

7.5. Viewing Entropy Distribution

This section presents graphical representation of entropy results. The Block Entropy Distribution of the original file is shown side by side with the Block Entropy Distribution of the final unpacked dump.

Figure 7.18 shows the 'Packed Blocks' window of SADE. The entropy distribution of the input file is shown in its original form and below it the entropy distribution of the unpacked data is displayed. The x-axis of the graphs shows byte entropy which has the range 0 to 8. The y-axis of the graphs show the number of 256-byte blocks. The bar graphs of the original packed file have peaks towards the right of the graph because packed files have high block entropies. The unpacked files have a more random distribution of data as well as greater size of data than the input file. The unpacked file's entropy distribution bar graph has peaks towards the left showing majority of 256-byte blocks having small entropy values.

Figure 7.18 – Entropy Distribution

7.6. Other Statistics

Other statistical measures like checksum, file entropy, average block entropy of the file etc. are displayed in this section. Figure 7.19 shows the 'Other Statistics' window.

Figure 7.19 – Other Statistics

7.7. History

This section displays the percentage of packed files and normal files in the form of a pie-chart. SADE maintains a history of files executed on it and this history is displayed as shown in figure 7.20.

Figure 7.20 – History

7.8. Help

SADE provides a help document with a walk through for novice users, an index and searching capabilities within the document. The snapshot of the help document is shown in figure 7.21.

Figure 7.21 – Help Window

CHAPTER 8

FUTURE WORK

SADE successfully unpacks the hidden code from an obfuscated file and retrieves useful information from the executable structure but still there is room for future work in the project which is described in the following subsections.

8.1. Disassembly of Code

A disassembly module can be integrated with SADE as future work to convert the compiled code inside the executable to assembly language. This will be of further assistance in executable analysis. SADE has been designed to be extensible and fully supports both implementation of a disassembler module or integration of an outsourced disassembler into the project. The disassembly module was out of the scope of our project because it is a huge undertaking in itself. With the introduction of a disassembler, SADE can be used as a cracking tool by the reverse engineering community. The debugger that has been implemented for SADE has the functionality to put breakpoints on any line of code and to view the state of the CPU registers as well as to see the process stack. The disassembly module can be integrated with the debugger to allow the user to put a breakpoint on any assembly instruction, single step through the code as well as see the state of the registers at any point during the process execution.

8.2. Reconstruction of Dumped Executable

The unpacked dump is taken as a snapshot of the process in memory. The unpacked dump is not executable because the Import Address Table requires reconstruction. Fields in the PE file header such as the Entry Point of the executable need to be corrected before the executable file can be run without first executing the unpacking routine code. This task is usually independently performed by executable reconstruction tools such as ImpRec. However, an import reconstruction tool can be

integrated into SADE to provide an output unpacked executable file that is executable and can be analyzed further with any standard executable tool.

8.3. Portability with 64-bit Windows Platform

SADE can be ported to work on 64-bit windows executable file as well. Currently it only works for 32-bit files.

ANNEX - A

PROJECT CHARTER

Project Charter
for
SADE
(Version 1.0 approved)
Prepared by
Faiza Khalid, Komal Babar, Nauvera Rehman, Abdul Wahab
Supervised by: Lec. Aisha Khalid, Dr. Fauzan Mirza

Project Description

The intent of the project is to develop a software analysis toolkit that will generically (without finding out the specifics of the compression and encryption scheme used) detect and unpack a packed windows executable file (PE32 file) and make the unpacked code available for analysis. The motivation behind the project is that the problem to generically unpack malware executables has been solved commercially but the competitive nature of the anti-virus software industry refrain them from publishing a solution. There is hence a lack of publicly available generic unpacking tools that can handle a wide range and variety of packed executable files without knowing the exact packer used to pack it. Furthermore, the growing epidemic of malware has strengthened the need to have more freely available tools to help in analyzing packed executable files.

Business Objectives and Success Criteria

Business Objective Success Criteria
Design and implement a technique to generically detect obfuscated (packed) windows executables and to extract and dump the code without running the executable.
Objective is measurable by testing the implemented technique on sample packed executables and comparing results with published statistics of existing unpackers.
Design and develop a user-friendly interface to analyze the recovered executable code.
Objective is measurable by feedback of toolkit from beta testers.
Contribute to research in the domain of code obfuscation and malware analysis.
Objective is measurable by publication of research.
119

Stakeholders

Internal Stakeholders:
Project Team Members
Project Supervisor
CS Dept, MCS
External Stakeholder:
Security Analysts
Vision
Software Analysis and De-obfuscation Engine (SADE) is a toolkit that generically unpacks packed executables for security analysts who need to analyze potentially malicious packed executables for creating signatures and understanding attacks.

Project Scope

The software product will be a toolkit that will generically detect and unpack a packed windows executable file (PE32 file) and make the encrypted and compressed file available for analysis purposes. The chief users of the application will be security analysts and main area of application for software is malware analysis. Malware authors use packing techniques to hide their malicious code and security analysts need to uncover the hidden executable code for creating signatures and understanding attacks. The unpacked executable file may or may not be a valid executable (i.e. able to run on Windows platform) but the unpacked code and other information about the file such as the modules and resources loaded by the executable will be available through the toolkit. This toolkit will be invaluable to security analysts as their time is expensive and individual malware samples can take hours to analyze and manual unpacking is a tedious and error prone process.
120

Assumptions and Dependencies

SADE is for Windows 32-bit platform and works for Portable Executable files. SADE might not work on every single type of packing obfuscation or on multiple layers of packing obfuscations. The packed input executable file might contain code that can detect the presence of SADE and our software may or may not be able to handle it.

Constraints

SADE might not work on every single type of packing obfuscation or on multiple
layers of packing obfuscations.
SADE will work only for Win PE32 files.
The packed input executable file might contain code that can detect the presence of
SADE and our software may or may not be able to handle it.

Milestones

Milestone Completion Date

1. Research paper on Generic
Unpacking Techniques
4th January,2009
2. Requirements Document 28th January, 2009
3. Packing Detection Algorithm 15th April, 2009
4. SADE Implemented 4th May, 2009
5. SADE tested 16th June, 2009
6. Project completion 20th June, 2009

Deliverables

Deliverables

1 Chief deliverable of project will be a toolkit for executable binaries with a deobfuscation
engine for the use of security analysts. Software Analysis toolkit:
SADE
2 Research paper on the developed unpacking and de-obfuscation technique.
3 Project Scope Statement
4 Requirements Model, Analysis Model and Design Model
5 WBS, project schedule, software development plan, software Implementation
plan, software test plan
6 Software Implementation description
7 Final Project Report
8 User Manual

Approvals

Approval Decision

☐ Approved, development of detailed project plan is authorized
☐ Approved, project execution is authorized
☐ Approved, but project is on hold until future notice
☐ Revise charter and resubmit for approval
☐ Charter and project proposal are rejected

Role or Title Name and Signature Date

Revision History

Name Date Reason For Changes Version

ANNEX - B

RESEARCH PAPER

“Generic Unpacking Techniques”

***Abstract—Traditional signature-based malware
detection techniques rely on byte sequences, called
signatures, in the executable for signaturematching.
Modern malware authors can bypass
signature-based scanning by employing the
recently emerged technology of code obfuscation
for information hiding. Obfuscation alters the byte***

sequence of the code without effectively changing the execution behavior. A commonly used obfuscation technique is packing. Packing compresses and/or encrypts the program code. Actual code stays hidden till runtime (when the executable is unpacked) making it immune to static analysis. Since every packer has its associated unpacker to undo packing, successful generic unpackers are difficult to come by. A few automated unpacking techniques have been published so far that attempt to unpack packed binaries without any specific knowledge of the packing technique used. In this paper, we aim to provide a comprehensive summary of the currently published prevalent generic unpacking techniques and weigh their effectiveness at dealing with the spreading nuisance of packed malware.

Dynamic analysis is a promising solution to the packing problem as every packed binary has to inevitably unpack itself for execution. Emulation (running code in a virtual environment) is an effective and powerful technique for generic unpacking. We will be reviewing various unpacking techniques based on emulation and a few other hybrid and alternative approaches.

Index Terms – obfuscation, generic unpacking, malware, dynamic analysis, emulation, virtual machines

INTRODUCTION

ne of the most prevalent features of modern malware is obfuscation. Obfuscation is the process of modifying something so as to hide its true purpose. Obfuscation increases the complexity of a program to make reverse engineering harder. Three of the most important practical obfuscations are packing, code reordering, and junk insertion. This paper only discusses packing, which is the most commonly used anti-reverse engineering technique. The packing obfuscation replaces a binary (code and data) sequence with a data block containing the binary sequence in packed form (encrypted or compressed) and a decryption routine that, at runtime, recovers the original binary sequence from the data block. The result of the packing obfuscation is a program that dynamically generates code in memory and then executes it. There are a large number of tools available for this purpose commonly known as executable packers [29, 30, 31, 32, 33]. Packing describes the process of encrypting a program and adding a runtime decryption routine to it, such that the behavior of the original program is preserved. Programs obfuscated by packing consist of a decryption routine (an instruction sequence that generates code and data), a trigger instruction that transfers control to the generated code, an unpacked area (the memory area where the generated code resides), and a packed area (the memory area from where the packed original binary is read) [10]. Packers embed an unpacking stub into the packed program and modify the program entry point to point to the unpacking stub. When the packed program executes, the operating system reads the new entry point and initiates execution of the packed program at the unpacking stub. The purpose of the unpacking stub is to restore the packed program to its original state and then to transfer control to the restored program. Packers vary significantly in their degree of sophistication. The most basic packers simply perform compression of a binary's code and data sections. More sophisticated packers not only compress, but also perform some degree of encryption of the binary's sections. Finally, many packers will take steps to obfuscate a binary's import table by compressing or encrypting the list of functions and libraries that the binary depends upon. In this last scenario, the unpacking stub must be sophisticated enough to perform many of the functions of the dynamic loader, including loading any libraries that will be required by the unpacked binary and obtaining the addresses of all required functions within those libraries [1].

Packing is applied on legitimate software to reduce the size of executable files and to protect the intellectual property that is distributed with the code. Malware writers use packing to bypass signature-based detection as packing completely modifies the binary foot-print of a program. The malicious code resides in the executable file in an encrypted form, and is not exposed until the moment the executable is run. A static analysis of a packed program will view the obfuscated block as non-instruction data or omit its analysis entirely, thereby hiding the program's true intentions. The percentage of new malware that is packed is on

Generic Unpacking Techniques

Komal Babar and Faiza Khalid, *MCS, NUST.*

O

the rise, from 29% in 2003 to 35% in 2005 up to 80% in 2007. This situation is further made complex by the ease of obtaining and modifying the source code for various packers. Modifications to the source code can introduce changes in the compression or encryption algorithm, create multiple layers of encryption and/or add protection against reverse engineering. Currently, new packers are created from existing ones at a rate of 10–15 per month [21]. According to WildList 03/2006, over 92% of malware file out there are runtime packed. Only 54 out of 739 files are not packed [17].

Unpacking is the recovering of the original program that has the same relevant behavior as the packed program. Unpacking consists of constructing a program instance which contains the embedded program, contains no code-generating routine, and behaves equivalently to the self-generating program [12]. Malware authors understand that analysts will attempt to break through any obfuscation, and as a result they design their malware with features designed to make deobfuscation difficult. De-obfuscation can never be made truly impossible since the malware must ultimately run on its target CPU; it will always be possible to observe the sequence of instructions that malware execute. In all likelihood, the malware author's goal is simply to make analysis sufficiently difficult that a window of opportunity is opened for the malware in which it can operate without detection [1]. Dedicated decryption routines can be developed to detect any packed virus but writing such a routine requires that the virus be analyzed completely. A thorough analysis of the malware and then developing and testing a specific decryption routine could take a lot of work and time to accomplish. Moreover, dedicated routines fail to detect modifications to the packing routine. Generic unpacking attempts to unpack obfuscated binaries without determining the exact packing technique used to pack the program. In this paper, we intend to identify, compare and contrast various existing generic unpacking techniques and highlight their strengths and weaknesses.

DETECTING PACKING

It can be useful to first detect that an executable has been packed using some encryption or compression technique before setting to the task of unpacking it generically. One of the heuristic methods used for packer detection is to see how the byte distribution (entropy) is changed by the packers as well as check import tables of the executable under observation [8]. Analyzing byte distribution involves determining the frequency of occurrence of the byte distributions of the file contents. Such a frequency analysis is advantageous in detecting compressed data as effective compression techniques tend to increase the entropy of byte distributions in the file. This is done without unpacking data in the file from its compressed form and therefore helps in detecting compressed files without actually decrypting its contents which otherwise would make the system vulnerable to potentially malicious executables [9]. However, this technique alone cannot be used as a criterion to identify a packed executable as legitimate copyright protected software also use packing for information hiding of an executable to evade disassembly of binary code and reduce size for distribution over the Internet.

DYNAMIC ANALYSIS TECHNIQUES

Once a packed executable has been detected, the executable then needs to be unpacked correctly. Dynamic analysis is a promising answer to the problem of hidden code extraction because it does not depend on signatures. Dynamic analysis techniques make use of the fact that no matter what packing technique is applied to the executable, the actual code or its equivalent will ultimately be available in memory and sooner or later, it will execute at some point at run-time. This innate property of a packed executable is the key to extracting the hidden binary code or its equivalent as a raw memory dump. However, it is not certain where the hidden binary code lies in the memory and when the execution flow jumps to the hidden code. Apart from this, another essential piece of information for analysis of an executable is the original entry point (OEP). *The original entry point is the first hidden instruction being executed when the program control flow is transferred from the decryption/unpacking routine to the hidden code* [18]. Dynamic analysis is less susceptible to being tricked by the use of obfuscation or self-modifying code. When using dynamic analysis techniques, the issue arises in the choice of environment in which the sample should be executed. The use of a sacrificial lamb (*a dedicated standalone machine that is reinstalled after each dynamic test run*) is not an efficient solution because of the overhead involved. In addition to determining the type of environment to be used for dynamic

analysis, one can also discern the different types of information that can be captured during the analysis process [15].

Run and Dump Unpacking

Generally, a packed program upon executing first unpacks the program in memory, loads the required libraries, and accesses the imported functions by scanning the import address table of the executable. The structure of the original program is exposed and a snapshot of the memory image can be taken at this point and stored in a file (called dumping). This file can then be analyzed for signature analysis by virus scanners. The advantage of this technique is that the critical and complex task of unpacking is done by the stub itself. The paradox of this technique is figuring out exactly when the snapshot of the memory should be taken. If the snapshot of the memory is taken prematurely (before the program has been completely unpacked) the entire hidden code will not be obtained for analysis. And if the snapshot is delayed for too long, the program will start executing the unpacked code, making the system vulnerable to malicious attacks [1]. Furthermore, the dumped executable requires some additional fixing of header structures, but the code itself is visible in its original form and available for reverse engineering and static analysis. This simple method works for most kinds of executable packers and encryptions, as the unpacking function typically extracts the complete program right at the start, and does not interfere with later computations (but this is not always the case). The biggest drawback of this method is that the executable must be loaded, which might not be acceptable in all cases as it cannot always be guaranteed that the program is terminated before any malicious function is called. Sandbox environments can be used to avoid the potential damage [11]. Sandbox is a virtual environment provided to the executables to run, so that they cannot exploit the actual system while they are being analyzed [2]. *Sandboxes are usually found in a kid's playground. Kids use it to play in, building and tearing down structures. A sandbox inside a scanner engine is also a playground – for computer files* [19]. Sandboxing can be performed in two ways: Sandboxing using Virtual Machine-where the executable runs on a subset of the actual system in a constrained controlled environment and Sandboxing using Emulationwhere the sandbox is a virtual world where everything is emulated drawing a concrete wall between the real and the emulated environment.

Virtual Machine

Running the executable in a virtual machine (i.e. a virtualized computer), such as one provided by VMware [24], is a popular choice. In this case, the malware can theoretically only damage the virtual PC and not the real one. After performing a dynamic analysis, the infected hard disk image is simply discarded and replaced by a clean one (i.e., so called *snapshots*). Most (or all) code is run directly in an isolated hardware environment which can be done using software solutions (VMware) or using hardware features (new Intel/AMD processors, IBM z/VM). It requires support from the OS or kernel-level modifications (drivers). In virtualization, code is not (usually) analyzed or cached. It is just run in an isolated environment [27]. Virtualization solutions are sufficiently fast. There is almost no difference to running the executable on the real computer, and restoring a clean image is much faster than installing the operating system on a real machine. The running code inside of a self contained environment can be more closely controlled than raw hardware [15]. Unfortunately, a significant drawback is that the executable to be analyzed may determine that it is running inside a virtual machine and may become inactive or execute differently in order to evade the virtual setup. All current virtual machines exhibit identifiable features and detecting them is one of the most common methods available to a malware author to protect malicious code from analysis [5].

Emulation

A PC emulator is a piece of software that emulates a personal computer (PC), including its processor, graphic card, hard disk, and other resources, with the purpose of running an unmodified operating system [15]. Generic code emulation is a very potent de-obfuscation technique. A virtual machine is implemented to simulate the CPU and memory management systems to impersonate the code execution. The packed executable is replicated in the virtual environment and no actual code (which may contain malicious content) is executed by the real processor. The purpose of the code emulation is to mimic the instruction set of the CPU using virtual registers and flags. It is also important to define memory access functions to fetch 8-bit, 16-bit, and 32-bit data.

Furthermore, the functionality of the operating system must be emulated to create a virtualized system that supports system APIs, file and memory management [2]. All the hardware resources are virtualized using data structures.

When the packed executable runs in the emulator, each instruction triggers some software routines that update the respective data structures in such a way that the program gets the same response it would get if run on an actual processor. Each instruction is first decoded to find the instruction type, length, operands and other information that need to be updated. Once the necessary information is available, the respective emulation routines are called which update any emulated hardware resources (which are actually data structures), if required. The address of the next instruction is obtained either as result of instruction decoding or computed by the emulation routine.

Generally, Program's Entry Point marks the beginning of emulation which executes instructions sequentially. Complexity of the unpacking process posses little difficulty to the emulation environment provided that the unpacking stub is available in the executable and the emulator has enough resources to complete unpacking. Since a program would use only a small subset of all the resources of system, it is quite affordable to provide emulator with these resources.

On the other hand, code emulation is significantly slower than running the code on an actual processor. Decoding a single instruction in a program requires hundreds of instructions to be executed at the back-end. After instruction decoding, several routines are called for updating data structures, find where the next instruction lies in memory, followed by many other steps to simulate a correct response. All these factors make emulation considerably slower and inefficient. This difference in speed of execution is one of the very strong tools used by anti-emulation techniques which could be embedded inside an executable being analyzed. However, such anti-emulation technique can as easily be fooled by emulator by providing incorrect clock readings so that the system appears faster to the program.

Another problem is that, for some packers, the program is not completely unpacked at one time, or some parts of its code may have been moved around by the packer. Also, if the emulation engine does not emulate correctly, error tracing becomes very complex since emulator executes a lot more instructions than an actual processor. One of the inherent problems of dynamic analysis techniques is deciding when to stop the emulation process. Heuristic checks can be used to help make this decision [16]. Occurrence of an event, which may be statically defined, could be used to stop emulation. In addition, some form of resource exhaustion limit, for example number of emulated instructions or emulation time, is needed in order to avoid infinite-loop execution [6]. Some other common ways to trick emulation is using fake API calls, using complex program logic (which can greatly slow down the emulation process). Since one of the major weaknesses of emulation is its speed, the goal of anti-emulation techniques is that the emulator quits without finishing the unpacking process (usually a maximum timeout is predefined in emulator) [25].

Difference between Emulators and Virtual Machines

It is important to differentiate emulators from virtual machines. Like PC emulators, Virtual Machines can run an unmodified operating system, but they execute a statistically dominant subset of the instructions directly on the real CPU. This is in contrast to PC emulators, which simulate all instructions in software. Because all instructions are emulated in software, the system can appear exactly like a real machine to a program that is executed, yet keep complete control. Thus, it is more difficult for a program to detect that it is executed inside a PC emulator than in a virtualized environment. A PC emulator has complete control over the sample program. It can intercept and analyze both native Windows operating system calls as well as Windows API calls while being invisible to malicious code. The complete control offered by a PC emulator potentially allows the analysis that is performed to be even more fine grain [15]. On the other hand virtual machines are much faster than PC emulation; they are almost up to the native speed of the system being used.

Unpacking Methods Using Emulation:

1. MALWARE NORMALIZATION

The method for malware normalization attempts to unpack malware generically. The presented method [12] assumes that the code generator and the instruction causing the control-flow transfer are reached in all program executions. Another assumption is that code generation is independent of inputs or the runtime environment. This is generally a

valid assumption to make as malware is usually independent and designed to automatically run and unpack itself on a variety of different victim systems.

Unpacking by malware normalization consists of two basic steps. First, execute the program in a controlled environment to identify the control-flow instruction that transfers control into the generated-code area. i.e. execute the program in an emulator, collect all the memory writes (retaining for each address only the most recently written value) and monitor execution flow. If the program attempts to execute code from a memory area that was previously written, capture the target address of the control flow transfer (i.e., the trigger instruction) and terminate execution. By emulating the program and monitoring each instruction executed, the moment when execution reaches a previously written memory location can be identified. Second, with the information captured in the previous step, construct a normalized program that contains the generated code. In the second step, construct a non-self-generating program. Using the captured data, an equivalent program can be constructed that does not contain the code generator. The data area targeted by the trigger instruction is replaced with the captured data. The memory write captured contain both dynamically generated code and the execution specific data e.g. the state of the program stack and heap. The executable file of the new program is a copy of the executable file of the old program with the byte values in the virtual memory range set from the captured data. The program location where execution was terminated is used as the entry point for the new program.

This technique has some major drawbacks. The unpacked executable is not ready-to-run. Although the packed coded can be successful unpacked and produced in the normalized executable, but since its not the actual file, the import table which lists the dynamically linked libraries and API calls used by the program may not be recovered, since most packing obfuscations replace it by a custom dynamic loader. This approach is open to resource consumption attacks and can have false negatives since the execution time in the sandbox often has to be heuristically restricted for performance reasons.

2. *RENOVO*

Reference [18] details a useful unpacking technique using emulation. The Renovo emulation technique is a fully dynamic method which monitors currently executed instructions and memory writes at run-time. Renovo uses an approach similar to malware normalization but with a few customizations. The approach maintains a shadow memory of the memory space of the analyzed program, observes the program execution, and determines if newly generated instructions are executed. Then it extracts the generated code and data. Assuming nothing about the binary compression and encryption techniques, it provides a means to extract the hidden code and information, which is robust against antireverse-engineering techniques.

After the packed executable starts, its attached decryption routine performs transformation procedures (also called hidden layers) on the packed data, and then recovers the original code and data. After this, the decryption routine sets up the execution context for the original program code to be executed. This involves initializing the CPU registers and

setting the program counter to the entry point
of the newly-generated code in memory.
A packed executable may have multiple
hidden layers, making it even more difficult to
analyze. But irrespective of the packing
method and the hidden layers, the original
program code and data will ultimately be
available in memory. Also, the instruction
pointer should jump to the OEP (Original
Entry Point) of the restored program code
which has been written in memory at run-time.
Making use of these properties of packed
executables, an algorithm to dynamically
extract the hidden original code and the OEP
from the packed executable has been suggested
in [18] which examines whether the current
instruction has been generated at run-time,
after the program binary was loaded. The
instruction pointer is monitored to see if it
jumps to the memory region which has been
written after the program start-up.
When a program is loaded in memory, a
memory map is generated and initialized as
clean. Whenever the program performs a
memory write instruction, the corresponding
destination memory region is marked as dirty,
which means it is newly generated.
Meanwhile, when the instruction pointer jumps
to one of these newly-generated regions, it is
determined that there is a hidden layer hiding
the original program code, and the newlygenerated
memory regions are indentified to
contain the hidden code and data, and the
address pointed by the instruction pointer as
the original entry point (OEP). To handle the
possible hidden layers that may appear later

on, the memory map is initialized as clean
again. The same procedure is repeated until
time-out.
The advantages of this approach are
threefold: Firstly, nothing is assumed about the
packing methods except the inevitable fact that
the original hidden code should eventually be
written and executed at run-time. Therefore,
the approach is able to handle any sort of
packing techniques applied to the binaries.
Secondly, the approach can determine the
exact memory regions accommodating the
code or data generated at run-time. Since the
information about memory writes are kept at
byte-level, it is possible to efficiently extract
the newly-generated code and data. Lastly, this
approach does not rely on any information on
the code and data sections of the binary.
When analyzing an executable, it is run in an
emulated environment. The emulated
environment facilitates simulating CPU
instructions in a fine-grained manner, in

particular the instructions that perform memory writes. This technique also suffers from the weaknesses of emulation and can easily be evaded by anti-emulation techniques. Packers also use anti-memory dumping technique that involves the deletion of a section of code immediately after it is executed.

3. SAFFRON [5]

Saffron is a generic automated malware unpacker that employs dynamic instrumentation using software tools such as intel PIN which provides facilities to closely monitor and interact with a program's execution. Saffron uses Pin to monitor execution flow and memory reads and writes of malware. As with other tool already discussed, SAFRON unpacking mechanism also depends on the original entry point of the program. If execution jumps to previously written memory, the target memory becomes a candidate original entry point and the memory is dumped to a file.

It however fails if the program uses a checksum to verify the integrity of the program’s address space before transferring control to the stub procedure. Such executables cannot be unpacked using Pin, as it is easily detectable and it also modifies the instrumented processes' address space. Another drawback of this approach is that it is fairly slow and standard anti-debugger techniques cause problems with Pin.

The other method used by Saffron is page fault handler debugging, which works by modifying Microsoft Windows's page fault handler and subverting the x86 architecture's paging mechanism to trace memory accesses to individual pages. During their presentation at the Black Hat security conference in August 2007 the authors mentioned that page fault handler debugging does not work within virtual machines, so that extra care must be taken to isolate malware samples to be analyzed, e.g., by using a real machine as a sacrificial lamb. The authors also stated during their presentation that their method does not yet automatically choose a most likely out of several candidate original entry points and that they currently rely on third-party software to reconstruct valid PE files from memory dumps [26].

4. Pandora's Bochs

Unpacking in Pandora's Bochs is done by Bochs [28], a portable x86 emulator which is a pure software virtual machine that provides a built-in mechanism for instrumenting code running on the emulated CPU. Pandora's Bochs collects a lot of information during

execution of a malware sample, such as all memory write and all branches. This information is useful in several ways. It is used during the reconstruction of the executable (without the unpacking stub) to help regenerating a program's import information. It can also be helpful for performing a detailed analysis of the inner works of unpacking stubs as it provides an accurate execution trace at the basic block level that is unaffected by code obfuscation and anti-debugging techniques.
The goal of the thesis [26] was to enhance the Bochs PC emulator to unobtrusively monitor execution of samples of packed malware in the emulated environment, use heuristics to determine when the unpacking process is done, and finally store a memory image of the malware process along with additional information for further (static) analysis. To achieve this goal, the Bochs PC emulator has been enhanced in several ways, such that it can identify processes running within the guest operating system, gather information about the processes such as which dynamic libraries are mapped into the processes' address spaces and which symbols these libraries export, trace execution of processes and determine which library functions they call and whether the execution path covers memory areas that were modified by the process and dump memory images of running processes, and to that end, also force the guest operating system to bring

in all virtual memory pages needed for a full dump. Additionally, an attempt is made to reconstruct the original, unpacked malware executable. The goal is to execute a packed binary within the enhanced Bochs PC emulator and return an unpacked, normalized version of this binary for further analysis. The unpacking process utilizes Boch's save/restore mechanism that can save and restore the emulator's CPU, memory, and device state. Pandora's Bochs takes a slightly refined approach to OEP detection. Moreover, to account for multiple unpacking stages, Pandora's Bochs simply continues execution of the monitored process while it still shows some progress, up to a user-defined timeout. Needless to say, this technique is also very slow in performance [26].

Dynamic Translation

The dynamic translation is an improvement of emulation with a better execution speed and performance. In this approach, the executable under analysis is disassembled and then an equivalent unpacked code is generated. The executable code obtained as a result of translation is *persisted*; if execution of

program enters a loop, the persisted coded can usually be executed directly without requiring translation. Code is translated only on the first iteration and for the subsequent iterations persisted code dos not need any translation. Thus the method eliminates redundant analysis of repeating code sequences. Although code translation makes execution slower, it is many times faster than pure emulation. To add here, the computational cost of disassembly is comparable to emulation.
The dynamic translation engine has to determine whether translated code is available for any given instruction, and if so, locate the corresponding code. One way to do this is to maintain a table with virtual addresses of translated instructions and addresses of corresponding executable code. However searching a virtual address in this table for each processed instruction will introduce additional overhead, negating the speed advantage of executing translating code. A much more efficient way is to partition the original code into blocks of instructions and only store a table entry for each block. This would improve table look-up operation. But dividing the original code into blocks is not as simple as it sounds. A block must have some specific properties that limit its size. On the other hand bigger the blocks are, more efficient the storage and searching will be. A basic building block is a contiguous block of code having a single entry point at the beginning of the block and a single exit point at the end of the block. If the code within such a block is executed via a call instruction to the beginning of the block, all the instructions in the block will be executed. A single instruction is needed at the end of the block to return the control to the caller. As a consequence, any basic building block of original code will contain at most one jump instruction.
Discovering and delimiting basic blocks is a dynamic process, meaning that new blocks maybe discovered or existing blocks could be modified as a result of processing previously discovered blocks. After translating a block and executing the resulted code, the beginning address of the next block to be processed will be the destination address of the jump instruction at the end of the block that was just executed. The main advantage of this technique is that if several blocks are executed inside a loop, searching for a successor block needs to be done only once for each successor. There is no need to search for successor of that block at any subsequent loop iteration.
Given any arbitrary program code to be analyzed, the code could be unsafe. It is

possible to translate the given code into another code sequence that is functionally equivalent to the original one and that can be safely and correctly executed on the host machine. There are multiple ways in which code translation that meets the above criterion can be achieved. Simplest technique is translating directly from the original code to target code. Each original instruction will be decoded and then an equivalent instruction or instruction sequence will be generated for the target code. Another method is translating using an intermediate language (IL). Each original instruction will first be translated to an intermediate code sequence and then the intermediate code will be translated to target native code. This method is preferable for multiple sources and multiple target languages. It is possible to perform code optimizations using the intermediate language form. The intermediate language should be platform independent. The intermediate language would need to support all the possible operators, operand types and a combination of these from all source languages raising its degree of complexity. Combining the above two methods could be achieved in a way that preserves the advantages of both without any of their disadvantages. Most instructions could

be translated using a fairly simple intermediate language while the most complex ones that would require a complicated intermediate language will be translated directly. Typically, the code obtained as a result of translation will not be as efficient as the original code. This may happen for various reasons like some instructions or operand encodings from the source language might not have a 1:1 correspondence in either the IL or the target language. It is possible to perform some optimizations at basic block level at translation time to improve efficiency of the translated code. If the analyzed code is linear, the code will be translated once and executed once. As the execution time is negligible as compared to the translation time, in this case translation would account for most of the analysis time. As improving the efficiency of the translated code is done at the cost of translation speed, a compromise between these two must be obtained.

For each file that needs to be scanned for malware, analysis consists of sequentially analyzing and processing of basic blocks. At the beginning of the analysis the current address is initialized to the entry point of the program to be scanned. After each basic block is processed, the current address is updated to the destination address of the jump instruction at the end of this block. The Dynamic Translation engine must provide access to various hardware devices (disk drives, keyboard, mouse, network interface, video card, real-time clock, etc.) as well as software resources such as BIOS data structures and routines and operating system APIs. Since the code provided to the dynamic translation engine could be malicious, most of these resources need to be virtualized.

Writing unpacking routines for all the packers publicly available takes a lot of development and test effort. In some cases, writing the unpacker would require reverse engineering the packer. In the absence of a dedicated unpacking routine, a packed executable would be emulated until the unpacked code is obtained. However unpacking with an emulator could be very slow, especially for large packed files that would require emulating several millions of instructions. Using dynamic translation, a program can be unpacked significantly efficiently as compared to emulation providing detection for malware packed with new packers, with reasonable speed performance, before a dedicated routine is developed. In some cases, the generic unpacking using Dynamic Translation could prove fast enough that dedicated routines won't even be needed [23].

HYBRID APPROACHES

Detection of packed malware requires the use of emulation and sandboxing technologies which are open to resource consumption attacks since the execution time in a sandbox has to be restricted for performance reasons [10].

1. Mixing Code Emulation and Specific Routines

If an executable is packed with a complex packer, it is quite useful to use a combination of emulation and specific unpacking routines for known packer. Although using specific routines is an efficient approach but integrating it with emulation introduces additional complexity to the system. [6].

As emulation based unpacking is too slow and static unpacking is too specific, a hybrid approach is a solution which combines the advantages of both. The hybrid approach would involve an emulator for IA-32 instructions, flat memory, Win32 APIs, Win32 system and Static un-packers (called "miniemulators"(MEs)). The generic emulator can be optimized by caching memory accesses, independent CPU flags emulation and avoiding CPU flags setting when possible. The static unpackers do not handle heavily polymorphic code. It is handled by slow generic emulation instead. Advantage of this technique is that it is universal by generic emulation and fast as the specific unpacking is independent of the code being emulated. It is easily expandable and can be implemented and deployed incrementally, as initially small number of specific unpacking routine are written and with time, the results of emulations can be used to widen the scope of specific mini-emulators. MEs have relatively simple implementation as each one handles one single algorithm. It is much faster than generic emulation and also faster than Dynamic Translation as no need for code analysis and tranlation or code optimization [14].

2. PolyUnpack

Reference [22] gives a hybrid unpacking approach called PolyUnpack. It is a behaviorbased approach that uses a combination of static and dynamic analysis to automate the process of extracting the hidden-code of packed malware. The core emphasis of this technique is on the results (i.e., runtime generated code execution) of unpack-execution rather than the unpacking mechanism used. It supercedes other approaches as prior knowledge about the packer or explicit programming of the semantic behavior capturing all instances in an unpacking class is not required. It first generates a static code view of the packed program (i.e. the code sequence of program does not produce any code at run-time) in memory using static analysis. The static code model is then forwarded to dynamic analysis engine (i.e. emulation or any virtual environment). This dynamic analysis differs from the usual runtime analysis as it has the ability to verify if the observed sequence of execution matches any part of the static model. While the stub (restoration routine) is being executed, the code will follow the same sequence as the stub’s static view. After execution of each instruction, the execution context is compared to the static model. The point in run-time when the code deviates from the static view indicates that the code has been unpacked (since unpacked code sequence is not available in static model). This is the stopping condition of analysis and provides the unpacked code of the executable under analysis. This technique uses the fact that the hidden code which has been obfuscated is not available to the static model. Thus this approach gives a wider scope to the static analysis which otherwise is too specific and provides a mechanism which is independent of packing mechanism.

The fundamental step of static analysis is to disassemble the program to identify code and data. The code portion is then partitioned from the data area to generate a static model.

This is not all the unpacker has to do as it is not always the case that absence of a code sequence in static model will be the hidden code. The dynamic link libraries (DLLs) loaded during Windows binary execution also result in execution of code that is not available in the static model. Misinterpretation of these DLLs has to be catered for. So, whenever, a DLL is loaded, the memory it occupies is noted. During single-step execution, the program's program counter (pc) is continually compared against all known memory areas. If the pc points to memory occupied by a DLL, the return address from the stack is read and a breakpoint set there, allowing single-step execution to resume after the call's return.

To deal with the increased program complexity due to multiple packed layers, the PolyUnpack technique can be extended to proceed iteratively. Every time the resultant hidden code is attained, the static model of the code is generated. Execution resumes from the first instruction of the code under analysis. If at any step the binary sequence of executed code does not match any part of static view, the next iteration begins. The final unpack code is the one whose static model is totally consistent with its run-time execution sequence. In other words, the last iteration would be running the unpacked code.

This technique like most instrumentation tools is not transparent to the malware being processed. Therefore, there exists the possibility that an instance of malware being executed in it may detect that it is being instrumented and alter its behavior (e.g., halting its execution instead of generating hidden-code) in order to evade extraction of its unpacked code [22].

ALTERNATIVE APPROACH

As described earlier, all the covered techniques have their own limitations. Even, poly-unpack suffers from inefficiency as it involves singlestep debugging. Hence an alternative approach to the above mentioned techniques is *OmniUnpack [21]*. Omni unpacking technique addresses the shortcomings of existing systems. This is a generic approach to handle any type of packer and any type of selfmodifying code. It does not depend on virtual environment, emulation or debugging for unpacking. It monitors the program execution and tracks written and written-then-executed memory pages. *Written-then-executed pages are indicative of unpacking but not indicative* *of the end of unpacking, as there could be multiple unpacking stages.* The approach uses heuristics to approximate the end of unpacking. In order to improve the performance, page-level monitoring is done. When stub, which is embedded in the program, writes the unpacked code to memory, the destination page is marked as writable but not executable. At the end of the unpacking stage, when the program accesses the same page for execution, the lack of execution permission causes a protection exception. If the program then makes a potentially damaging system call, a malware detector is invoked on the written memory pages. If the detection result is negative (i.e., no malware found), execution is resumed. The resulting low overhead means that this technique can be used for continuous monitoring of a production system.

When the virtual-physical address mapping needs to be updated or when the memory protection is violated, the hardware signals to

the OS through an exception and allows the OS to repair the memory state before continuing execution. Existing features are used to intercept the first moment when a page is written and the first moment when a page is about to be executed after a write.

A disadvantage of the technique is imprecision of page-level tracking. Page-level tracking decreases the granularity of monitoring although it significantly reduces the overhead of memory-access tracking. It is less precise, often resulting in incorrectly detecting unpacking stages. It would be unnecessarily expensive to invoke the malware detector every time a written memory page is executed, because such an event (written-thenexecuted) is frequent. Hence determining the end of unpacking is hard to decide and is only an approximation. The technique assumes that a packed program will generally be malicious. Therefore, it provides the facility to automatically call the malware detection engine if a dangerous system call is made. This introduces anther level of complexity in the algorithm as decision has to be made about the choice of dangerous system calls. A dangerous system call is a system call whose execution can leave the system in an unsafe state. To achieve its malicious goal, the malware has to interact with the system. As a simple solution, any system call that modifies OS state is considered dangerous in this technique.

Because of the possibility of multiple unpacking stages and of the approximation being using to detect them, it is insufficient to monitor and scan the program only once during an execution. This technique implements a continuous monitoring approach, where the execution is observed in its entirety. This is a necessary departure from the traditional view of unpacking and scanning as separate, one-time stages of the malware detection process. Also, efficiency can further be increased if all memory-page accesses are not observed. It is sufficient to observe the first memory access in an uninterrupted sequence of accesses of the same type. For example, only the first write to a page is useful, subsequent writes to the same page do not impact the result of the algorithm and can be ignored. Thus, this unpacking technique aims to be very generic as it supports binaries packed with any arbitrary algorithms applied any number of times [21].

CONCLUSION

Current approaches for automatic analysis suffer from a number of shortcomings. One problem is that malicious code is often equipped with detection routines that check for the presence of a virtual machine or a simulated OS environment. When such an environment is detected, the malware modifies its behavior and the analysis delivers incorrect results. Malware also checks for software (and even hardware) breakpoints to detect if the program is run in a debugger. This requires that the analysis environment is invisible to the malicious code [15]. As any other dynamicanalysis technique, emulation places a time limit on the execution of the packed program and is restricted by the reliability of the emulation environment. Extracting packed binaries and finding the original entry point using dynamic analysis is feasible but these approaches either rely on some heuristics or require disassembling the packed program. However, heuristics about packed code may not be reliable in all cases and can be easily evaded. In addition, correctly disassembling a binary program itself is challenging and errorprone. Hybrid approaches for packed code extraction perform a series of static and dynamic analysis which leads to performance overhead.

REFERENCES

[1] SHON HARRIS, ALLEN HARPER, CHRIS EAGLE AND JONATHAN NESS, "GRAY HAT HACKING", 2ND ED., MCGRAW-HILL, CHAP 21.

[2] Peter Szor (2005, Feb 3). "Art of Computer Virus Research and Defence", Chap 11, [Online] Available: http://safari.oreilly.com/0321304543/ch15lev1sec4.

[3] Dr. Jose Nazairo, "Botnet Tracking: Tools Techniques and Lessons Learned", presented at Lockdown 2007 University of Wisconsin-Madison, page 12.

[4] Gaith Taha, "Counterattacking the packers", presented at AVAR 2007 Conference in Seoul, page 1.
[5] Danny Quist and Valsmith, "Covert Debugging: Circumventing Software Armoring Techniques", presented at Black Hat Briefings USA August 2007, page 1-2.
[6] Miroslav Vnuk and Pavol Navrat, "Decompression of run-time compressed PEfiles.", presented at IIT.SRC 2006 – Student Research Conference, Slovak University of

Technology, Faculty of Informatics and Information Technologies, page 2-4.
[7] Andrew Lee and Pierre-Marc Bureau, "The Evolution of Malware", presented at Virus Bulletin Conference November 2007, page 8-10.
[8] Proceedings of the 5th Australian Digital Forensics Conference 3rd December 2007, Edith Cowan University, Mount Lawley Campus. Page 67, [Online] Available: http://scissec.scis.ecu.edu.au/conference_proceedings/2007/forensics/00_Forensics2007_Complete_Proceedings.pdf
[9] Andreas Beetz, "File Analysis", US Patent US 2004/0236884 A1, Nov. 25, 2004.
[10] Mihai Christodorescu, Somesh Jha, Johannes Kinder, Stefan Katzenbeisser and Helmut Veith, "Software Transformations to Improve Malware Detection" In Journal in Computer Virology, vol. 3, (4): pp. 253–265, November 2007.
[11] Johannes Kinder , "Model Checking Malicious Code", Diplomarbeit, Technische Universität München, 2005.
[12] Mihai Christodorescu and Somesh Jha, "Malware Normalization", Technical Report #1539, Nov. 2005, page 7-9.
[13] Raymond J. Canzanese, Matthew Oyer, Spiros Mancoridis and Moshe Kam, "A Survey of Reverse Engineering Tools for the 32-Bit Microsoft Windows Environment", Jan. 2005, page 17-20.
[14] Mario Alberto López, "Unpacking, a Hybrid Approach", presented at 2nd International CARO Workshop 1st & 2nd May 2008, The Netherlands.
[15] Ulrich Bayer, Andreas Moser, Christopher Kruegel and Engin Kirda, "Dynamic Analysis of Malicious Code", in Journal in Computer Virology 2(1): 67-77 (2006), page 2-5.
[16] Tobias Graf, "Generic Unpacking How to handle modified or unknown PE Compression Engines" presented at Virus. Bulletin Conference 2005.
[17] Tom Brosch and Maik Morgenstern, "Runtime Packers: The Hidden Problem.", presented in Black Hat briefings USA 2006, page 3.
[18] Min Gyung Kang, Pongsin Poosankam, and Heng Yin, "Renovo: A Hidden Code Extractor for Packed Executables", presented at 5th ACM Workshop on Recurring Malcode (WORM 2007), page 1-4.
[19] Kurt Natvig, "Sandbox technology inside AV Scanners", presented at Virus Bulletin Conference, September 2001, page 2.
[20] Norman SandBox Whitepaper, [Online] Available: http://www.norman.com/Download/White_papers/en , page 14.
[21] Lorenzo Martignoni, Mihai Christodorescu and Somesh Jha. "OmniUnpack: Fast,

Generic, and Safe Unpacking of Malware",
presented at 23rd ACSAC (Annual Computer
Security Applications Conference) in Miami
Beach FL USA (2007), page 1-4.
[22] Paul Royal, Mitch Halpin, David Dagon,
Robert Edmonds, and Wenke Lee,
"PolyUnpack: Automating the Hidden-Code
Extraction of Unpack-Executing Malware",
presented at 22nd ACSAC (2006), page 1-6.
[23] Adrian E. Stephan, "Defeating Polymorphism:
Beyond Emulation" presented at Virus
Bulletin Conference, October 2005, page 1-7
[24] VMware: server and desktop virtualization,
2006. http://www.vmware.com
[25] Xiaodong Tan, "Anti-unpack Tricks in
Malicious Code", Security Labs, Websense
Inc. presented in AVAR 2007, Seoul. Page 5-
29.
[26] Lutz Bohne, Diploma Thesis "Pandora's
Bochs: Automatic Unpacking of Malware",
28th January 2008, page 23-44.
[27] Jarkko Turkulainen, F-Secure Corporation
"Emulators and disassemblers", T-110.6220,
page 21-31.
[28] The Bochs Development Team. Bochs - The
Cross Platform IA-32 Emulator 2007.
http://bochs.sourceforge.net/.
[29] ASPack Software. ASPack and ASProtect
http://www.aspack.com/.
[30] Bitsum Technologies. PECompact2.
http://www.bitsum.com/pec2.asp.
[31] Obsidium Software,
http://www.obsidium.de/show.php?home
[32] Teggo. MoleBox Pro,
http://www.molebox.com/download.shtml
[33] Silicon Realms Toolworks. Armadillo,
http://siliconrealms.com/index.shtml.

ANNEX - C

Bibliography

[1] Andrew Lee and Pierre-Marc Bureau, "*The Evolution of Malware*", presented at Virus Bulletin Conference November 2007.
[2] Mihai Christodorescu, Somesh Jha, Johannes Kinder, Stefan Katzenbeisser and Helmut Veith, "*Software Transformations to Improve Malware Detection*" published in Journal in Computer Virology, Vol. 3, November 2007.
[3] Raymond J. Canzanese, Matthew Oyer, Spiros Mancoridis and Moshe Kam, "*A Survey of Reverse Engineering Tools for the 32-Bit Microsoft Windows Environment*", January 2005.
[4] Gaith Taha, **"***Counterattacking the packers*" presented at AVAR 2007 Conference in Seoul.
[5] Xiaodong Tan, "*Anti-unpack Tricks in Malicious Code*" Websense Inc., Security Labs, AVAR 2007, Seoul
[6] Lorenzo Martignoni, Mihai Christodorescu and Somesh Jha, "*OmniUnpack: Fast, Generic, and Safe Unpacking of Malware*" presented at 23rd ACSAC (Annual Computer Security Applications Conference) in Miami Beach FL USA, 2007.
[7] Kim-Kwang Raymond Choo, Russell G Smith and Rob McCusker, "*Future directions in technology-enabled crime: 2007–09*", Australian Institute of Criminology, 2008.
[8] Symantec Internet and Security Threat Report, Vol IX
[9] Matt Pietrek, "*An In-Depth Look into the Win32 Portable Executable File Format*".
[10] *Microsoft Portable Executable and Common Object File Format Specification* (Revision 8.0)

[11] Chapter 11, "*Antivirus Defense Techniques: The Art of Computer Virus Research and Defence*", Peter Szor, 2005.
136
[12] Johannes Kinder, "*Model Checking Malicious Code*", 2005.
[13] Shon Harris, Allen Harper, Chris Eagle and Jonathan Ness, "*Gray Hat Hacking*", 2nd Edition, McGraw-Hill.
[14] Lutz Bohne, "*Pandora's Bochs: Automatic Unpacking of Malware*", 2008.
[15] PEiD software, www.peid.info
[16] Ulrich Bayer, Andreas Moser, Christopher Kruegel and Engin Kirda, "*Dynamic Analysis of Malicious Code*", Journal in Computer Virology, Vol. 2, 2006.
[17] Miroslav Vnuk and Pavol Navrat, "*Decompression of run-time compressed PEfiles*", presented at IIT.SRC 2006 – Student Research Conference, Slovak University of Technology.
[18] Mario Alberto López, "*Unpacking, a Hybrid Approach*" presented at 2nd International CARO (Netherlands) Workshop 2008.
[19] Mihai Christodorescu, Somesh Jha, Johnnes Kinder, Stefan Katzenbeisser, Helmut Veit, "*Malware Normalization*", Technical Report #1539, 2005
[20] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee, "*PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware*", presented at 22nd ACSAC (Annual Computer Security Applications Conference) 2006.
[21] Min Gyung Kang, Pongsin Poosankam, and Heng Yin, "*Renovo: A Hidden Code Extractor for Packed Executables*", presented at 5th ACM Workshop on Recurring Malcode, WORM 2007.
[22] Thomas M. Cover and Joy A. Thomas, "*Elements of Information Theory*" 2nd Edition published John Wiley and Sons, 2006.
[23] Eldad Eilam, "*Reversing: Secrets of Reverse Engineering*", Wiley Publishing, Inc. 2005.
[24] Mario Hewardt and Daniel Pravat, "*Advanced Windows debugging*" published by Addison-Wesley, October, 2007.