RWTH AACHEN UNIVERSITY

Department of Computer Science

Diploma Thesis

Pandora's Bochs: Automatic Unpacking of Malware

Lutz Böhne

28th January 2008

First Examiner: **Prof. Dr. Felix C. Freiling**
Second Examiner: **Prof. Dr. Christian Bischof**
Advisor: **Thorsten Holz**

Laboratory for Dependable Distributed Systems
University of Mannheim

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 28. Januar 2008

Lutz Böhne

Abstract

This thesis highlights the threat that malware poses to computer systems today and illustrates techniques used by malware to prevent or hamper dynamic and static analysis. In particular it notes the problems that runtime-packing poses to static analysis and details the development of a software that aims to automatically unpack the original code from runtime-packed executables to make it available to subsequent static analysis.

The presented solution, dubbed Pandora's Bochs, extends the Bochs PC emulator to enable it to monitor execution of the unpacking stubs that are used by runtime-packed binaries to extract the original code into virtual memory. It detects the execution of newly generated code, extracts it from the virtual address space of the monitored process and attempts to generate a valid unpacked executable file from the extracted data.

Pandora's Bochs is evaluated on a set of (non-malicious) synthetic samples, so that the unpacking results can be compared to a known origin, as well as on a set of malware collected by the RWTH Aachen honeynet, to evaluate how it performs on unknown, malicious samples.

Zusammenfassung

Diese Diplomarbeit stellt die Gefahren vor, die Schadsoftware heutzutage für Computersysteme darstellt. Sie erläutert einige Techniken die von Schadsoftware benutzt werden um eine eingehende dynamische oder statische Analyse zu erschweren oder gar zu verhindern. Insbesondere wird darauf eingegangen vor welche Probleme Laufzeitpacker statische Analysemethoden stellen. Es wird die Entwicklung einer Software beschrieben, die automatisiert den ursprünglichen Programmcode aus laufzeitgepackter Schadsoftware extrahieren soll um ihn statischen Analysewerkzeugen zugänglich zu machen.

Die vorgestellte Lösung, genannt Pandoras Bochs, erweitert den Bochs PC Emulator so, dass er die Ausführung der Entpackroutinen von Laufzeitpackern überwachen kann. Sie kann die Ausführung von neu generiertem Programmcode bemerken, diesen aus dem virtuellen Addressraum des überwachten Prozesses extrahieren und den Versuch unternehmen, daraus eine gültige ausführbare Datei zu rekonstruieren.

Die Leistungsfähigkeit von Pandoras Bochs wird sowohl anhand von bekannten gutartigen lauftzeitgepackten Programmen bewertet, um das Ergebnis mit dem bekannten Original vergleichen zu könne, als auch anhand des Verhaltens auf einer Menge von vormals unbekannten, gepackten Schadprogrammen, die mit Hilfe des Honeynet der RWTH Aachen gesammelt wurden.

Contents

List of Figures

1 Introduction

1.1 Motivation

In recent years the Internet has evolved from a research tool into a critical part of today's communication infrastructure that is used for research, business and leisure alike. The number of computers that are connected to this global network is rapidly increasing, but despite all the positive effects of this development, there is also a dark side: malware. Malware is becoming increasingly common and poses a major threat to individuals and businesses, as it is employed by both criminals and vandals to steal valuable information, commit fraud, send junk mail or attack computer systems.

To effectively combat malware, security researchers need to analyse it and learn its tactics and weaknesses so that they can develop means to detect and counter it. Malware authors on the other hand try to prevent successful analysis by employing a variety of techniques to evade detection mechanisms and prevent successful analysis. One of these methods is the use of *runtime packers* and *executable protectors* that try to hide malicious code by compressing or encrypting it, and extracting it to memory only when the malware is executed. Furthermore, they try to protect the extraction routine by employing code obfuscation and anti-debugging techniques to prevent manual extraction of the malicious code by security researchers.

While there are other projects attempting to solve the problem of automatically unpacking malicious code that is hidden by runtime packers and executable protectors, most of these projects either show some deficiencies or are closed projects with only little information and no software publicly available in binary or source code form. Pandora's Bochs intends to remedy this by addressing some of the deficiencies and contributing source code back to the research community.

1.2 Tasks

The main task of this thesis was to modify the Bochs PC emulator so that it can unobtrusively monitor execution of runtime-packed malware samples within the emulated environment, determine when unpacking is complete, and store a memory dump of the monitored process as well as additional information gathered during the unpacking process for further analysis. Several problems had to be addressed and appropriate functionality be added to Bochs:

- The emulator needs to be enabled to identify processes running on the guest operating system and selectively monitor processes of interest.

- The emulator must be enabled to gather information about processes being monitored, such as what executable files and shared libraries are mapped into their address spaces and what symbols they import or export.
- The emulator needs to be enabled to trace code execution and memory writes of processes that run on the guest system. In particular, it should be enabled to note when code execution reaches memory that was previously modified.
- The emulator must be enabled to create memory dumps of processes.
- The emulator should be enabled to coerce the guest operating systems demand-paging mechanism to bring in memory pages that are needed by the monitoring code, e.g., to be able to completely parse executable files in virtual memory, or to generate complete memory dumps.

Additionally, an attempt should be made to generate valid executable files from process dumps and additional information that is gathered during unpacking, if at all possible. The final goal would be – if possible – to execute a packed binary within the enhanced Bochs PC emulator and return an unpacked, normalized version of this binary for further (static) analysis.

1.3 Results

A working prototype was developed on top of Bochs's instrumentation interface that can identify and monitor individual processes executing on a Windows XP SP2 guest operating system within Bochs. Memory writes and control transfers can be monitored, execution of modified memory – indicative of execution of new, unpacked code – can be detected, and any part of the virtual address space can be dumped as necessary.

To unpack a runtime-packed executable, its execution within the modified emulator is closely monitored and memory writes and branch instructions are instrumented and logged to a database. Upon executing previously modified memory, a process image is dumped to that database, parts of the virtual address space are marked as unmodified and execution continues to unpack possible additional packed layers. The modified emulator stops unpacking either guided by heuristics that track "innovation" of monitored processes, or after a configurable timeout that guarantees termination. Once the emulator terminates, the presented solution attempts to reconstruct a valid PE image from the information collected during unpacking.

The presented solution succeeds in extracting hidden code from most runtime-packers and executable protectors. It is transparent to the guest operating system and the binaries to be unpacked, thus unaffected by anti-debugging techniques and therefore hard to detect and evade. Reconstructing valid, executable PE images succeeds for many runtime-packers but fails for some more advanced protection schemes, e.g., for executable protectors that manipulate the original code or that obfuscate import information. The presented solution does not perform well on packers for which unpacking is of high

algorithmic complexity, rendering automated unpacking of these infeasible. This is in part due to poor performance of the underlying PC emulator.

1.4 Thesis Outline

The remainder of this thesis is organised as follows:

Chapter 2, "Prerequisites", introduces terms, concepts and technologies that form the prerequisites for the following chapters. It introduces malware and common types thereof, shows why it is important to analyse malware and highlights some of the problems typically faced during malware analysis. Furthermore it introduces the concept of runtime-packing executable files, details the file format used on the Windows platform and illustrates typical behaviour of runtime-packers targeting that file format. It gives a brief introduction into segmentation and paging on the x86 architecture and presents an overview of x86 virtualization technology. Lastly, it presents related work.

Chapter 3, "Implementation", describes the implementation of Pandora's Bochs. It details the design goals that drove development of the presented solution and sets the scope of what behaviour of runtime packers it is designed to handle. It then elaborates on the software that was chosen as a basis for Pandora's Bochs, and lastly, gives details on the actual implementation.

Chapter 4, "Evaluation and Results", shows how the implemented solution's performance was evaluated on synthetic samples, and unknown malware samples. It presents the results of that evaluation and analyses some of problems that occurred.

Chapter 5, "Conclusion", comments on these results, highlights the achievements and deficiencies of Pandora's Bochs and presents potential solutions to the problems that were encountered as potential future work. It concludes this thesis by recapitulating the achievements, comparing the presented solution to related work finishing with some closing remarks.

1.5 Acknowledgements

First and foremost, I wish to express my deepest gratitude to *Prof. Dr. Felix C. Freiling* for giving me the opportunity to pursue such an interesting project for my thesis, and likewise I want to thank *Prof. Dr. Christian Bischof* for being my second examiner. Furthermore, I would like to thank *Thorsten Holz*, my advisor, for his support and valuable feedback.

Many thanks go to *Jan Göbel* of the Center for Computing and Communication of RWTH Aachen University for supplying me with malware samples from the RWTH Aachen honeynet.

I would also like to thank #openrce on the freenode IRC network for bearing with my many questions about Windows internals.

Christina, thank you for your encouragement, your support, and your love.

2 Prerequisites

2.1 Malware

The term *malware* is a conjunction of the words "malicious" and "software" and is generally used to refer to software that is intended to perform tasks on computer systems without the owners' consent. This section first presents some malware-related terminology in section 2.1.1, then details why malware poses a threat to computer systems in section 2.1.2 and lastly presents the most common method that is used for malware detection in section 2.1.3.

2.1.1 Terminology

This section describes some of the terms commonly used to describe the properties of malware. Some of these terms are related to a malware's infection or propagation strategy, and some are related to the exhibited malicious behaviour. Much of the information in this section is based on [Szö05, section 2.3], [Eil05, chapter 8] and [SZ03].

2.1.1.1 Viruses

The term *virus* was coined in 1984 by Fred Cohen "as a program that can 'infect' other programs by modifying them to include a possibly evolved copy of itself" [Coh84]. These other programs can then infect even more programs, causing the virus to spread. This spreading mechanism sets viruses apart from other malware that are standalone programs and propagate by other means. In the early days of malware, when most personal computers were not networked, viruses were the most prevalent form of malware, replicating by users sharing infected software, e.g., on floppy disks or bulletin board systems. While Cohen's formal definition of a virus also allows for benign programs with the infection property (he gives the example of a runtime packer), today the term is typically used only for referring to malware, often even regardless of whether that malware is file-infecting or not.

Viruses that are true to this definition are rare today, as malware now typically replicates over the Internet.

2.1.1.2 Worms

The term *worm* describes a program that propagates over networks, e.g., the Internet. They typically do so without user interaction by exploiting software vulnerabilities in operating systems, network services or applications, e.g., web browsers, mail user agents,

or instant messaging clients, but there are also e-mail worms that rely on users opening malicious file attachments. With most computers today connected to the Internet, worms have the potential to spread quickly and uncontrollably. Worms are typically standalone programs, though some worms also exhibit the file-infecting properties of a virus.

2.1.1.3 Trojan Horses

Trojan Horses (or short: *trojans*) are malware in the disguise of benign programs or other benign files. A trojan horse can be a slightly modified version of a non-malicious program, or made from scratch, possibly emulating all or a subset of the expected benign behaviour. Trojans sometimes also try to hide their malicious functionality by pretending to be innocuous files normally not associated with executable code, like image, audio, or video files. This method is particularly effective on the Windows platform that determines a file's "type" by its file extension. Windows also hides known extensions from users by default and allows for a variety of (unknown to the common user) file extensions that indicate executable files, for example the *.scr* filename extension that is commonly used for screensavers. However, these are in fact are just common executable files.

A third way for seemingly innocuous files to execute code is to exploit software vulnerabilities in applications or commonly used libraries. In the past, several such vulnerabilities have been found in image file parsing libraries, and file compression libraries.

Trojan horses propagate by tricking users into executing them, e.g., by presenting themselves as a fun game, screensaver or e-postcard. For a very detailed discussion on trojans, please see [SZ03, chapter 6].

2.1.1.4 Backdoors

The term *backdoor* refers to a certain kind of (malicious) behaviour that can be exhibited by malware. According to [SZ03, chapter 5], "a backdoor is a program that allows attackers to bypass normal security controls on a system, gaining access on the attacker's own terms", i.e., a backdoor allows an attacker to perform unauthorized actions on a computer system. A typical example of a backdoor is a program that allows an attacker to remotely control a computer system.

2.1.1.5 Bots

The term *bot* is short for "robot" and is used to refer to software that acts autonomously on behalf of its owner. Non-malicious bots are used by search engines to automatically index websites, or on Internet Relay Chat (IRC) to provide useful functionality to users of a particular IRC network or channel. Malicious bots typically form *botnets* consisting of up to several thousand infected computers. These botnets can be controlled by their owners through one or more *command & control* (C&C) servers, which commonly run an IRC server (or a slightly modified one) to which the bots connect and then wait for commands. Other means of communication that are employed by bots are the *HTTP* protocol, or *peer-to-peer* protocols.

Botnets can be used to launch distributed denial-of-service (DDoS) attacks, e.g., for the botnet owners' entertainment, or, to blackmail companies by threatening to attack on their critical infrastructure. Other uses of botnets are for sending illegitimate bulk email, spying on infected computers and their users (e.g., stealing their authentication credentials, credit card information or other private data) or to perform click fraud. More information about bots and botnets can be found in [BHKW05] and [Hon07].

2.1.2 Threats

A computer system is typically deemed secure if the *confidentiality*, *integrity* and *availability* of the information contained therein is warranted [ISO06, p. 9,p. 31].

It is easy to see that malware typically violates one or more of these properties. If malware infects a computer system, the integrity of the system is compromised, as that malware was typically created on the system without authorization. Furthermore, as soon as malware executes on a system, it can usually alter data at will. It is not uncommon for malware to spy on users of a computer system and report its findings back to the malware's owner, thus violating the confidentiality of that data. Finally, malware can violate the availability of a computer system in several ways, e.g., that of its host by consuming processor cycles, memory, or bandwidth, or that of another system by being used in a distributed denial of service attack against it.

Violation of confidentiality is probably the most severe threat posed by malware today. Malware can steal personal information such as real names, birth dates or social security numbers to be sold and/or utilized for identity theft, it can acquire authentication information for email services, bank accounts or online games, it can steal credit card information, or, in a corporate environment, even trade secrets. According to [Sym07], there are lively underground economies were such information is sold in bulk quantities.

Additionally, malware is ever increasing in numbers. In the first half of 2007, Symantec Corporation detected 212201 new malware threats, a 185% increase over the previous half year [Sym07].

2.1.3 Signature-based Malware Detection

A common and fairly simple method employed by anti-virus software to detect malware matches files against a database of *signatures*[Szö05, chapter 11]. Note that "anti-malware" software would be a more appropriate term, as most malware today are not viruses as defined in section 2.1.1.1. However the term "anti-virus" software was coined at a time when viruses were the most common form of malware and it has since stuck. It will thus be used in this thesis to denote any anti-malware software.

A signature is a "fingerprint", normally a byte string or pattern, that should match a certain instance or family of malware while not matching any legitimate software. Signature databases are maintained by vendors of anti-virus software who capture and analyse malware samples and generate signatures for them. The databases need to constantly be updated, so that the window of vulnerability between a new piece of malware appearing in the wild and anti-virus software being able to detect it stays

as small as possible. Only being able to detect known malware is one of the main disadvantages of signature-based malware detection. It can however be performed very efficiently, that is why it is still the most widely used detection method today.

Simple signatures can be just byte strings representing an instruction sequence found only in a certain piece of malware. Pattern-based signatures can be used to make up for slight differences within a malware family. As malware signature databases typically contain tens of thousands of signatures, maximizing scanning performance is of utmost importance. One way to reduce scanning complexity is by generating signatures based on the *entry-point* of a program. The entry point is the first instruction that the operating system passes control to after initializing a new process's virtual address space. Its address is normally noted in an executable file's headers, so that it is easily locatable. Thus, signatures can be generated for the bytes at or around the entry-point, effectively limiting the space which anti-virus software has to search for signature matches.

Evading Signature-Based Detection Over the course of malware evolution, several methods where invented and refined by malware authors to evade signature-based malware detection methods (see [SF01],[Szö05, chapter 7]). The first step were *encrypted* malware, consisting of an encrypted malware body accompanied by a decryptor stub, i.e., a small unencrypted routine that decrypts the malware body into memory at runtime and transfers control to it. As different encryption keys can be used for every instance of a malware, this makes signature-based detection of the malware rather hard.

In addition to helping malware to evade signature-based detectors, the encrypted malware body also cannot be analysed without decrypting it first. It is common for malware to therefore try to protect the decryptor stub from reverse engineering attempts by using anti-analysis and anti-debugging techniques (see also section 2.2). These techniques are also used by benign software to protect itself from reverse engineering attempts, so that many such executable protectors were developed over the course of time.

However, signature-based anti-malware software can still detect decryptor routines used by malware, albeit being unable to distinguish different kinds of malware or to distinguish malware from benevolent software employing software protection mechanisms for legitimate reasons. Thus *oligomorphic* and *polymorphic* malware where the next step in evading signature-based detection.

Oligomorphic malware tries to avoid detection of the decryptor stub by not only choosing a new encryption key for each new generation, but also by slightly mutating the decryptor, e.g., by choosing one out of a (small) set of different decryptors. Polymorphic malware takes this concept to another level by being able to mutate the decryptor to a large number of different forms. This is commonly achieved by *mutator engines* that actively transform decryptor code at the machine code level into functionally equivalent forms. Metamorphic malware extends the concept of mutating decryptor code to mutating the whole malware body, not necessarily using encryption.

As implementing a runtime encryptor from scratch is not a trivial task, most malware authors today seem to just use existing ones to try and prevent their malware from being detected and analysed.

2.2 Malware Analysis

As highlighted in section 2.1.2, malware poses a threat to computer systems. Criminals are discovering its value for illicit business and the number of different malware is constantly increasing. To counter this development, security researchers need to find ways to fight malware, i.e., they need to obtain malware samples, analyse them to gain an understanding of malware tactics and weaknesses, and use that understanding to develop effective countermeasures.

2.2.1 Obtaining Malware

Malware samples can be obtained in several ways. Firstly, they can be collected during incident response, when analysing infected computer systems. Secondly, some businesses and researchers also run websites to which users can submit suspicious files. Some websites typically also offer to analyse them, e.g., [VT, CWS], while others, like [OC07], try to promote anti-malware research by providing a platform that researchers can use to share malware samples among one another. Thirdly, an effective approach to actively collecting malware is the use of *honeynets*. Honeynets are networks of *honeypots*, where "a honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource" [Hon04, p. 18]. In practice, this means that honeypots provide services that no legitimate user has a reason to access, so that any interaction with it can be considered unauthorised and therefore malicious and worthy of further investigation.

Honeypots are generally divided into two categories, *high-interaction honeypots* and *low-interaction honeypots*. Their main difference, as the name suggests, is the amount of interaction they permit, i.e., how many degrees of freedom an attacker has when interacting with the honeypot. High-interaction honeypots are typically real or virtual machines, running full operating systems and applications. They are useful for catching unknown attacks, but typically require more effort to set up and maintain. HoneyBow[ZHH+07, mwc07a] is a high-interaction virtual honeypot that aims to automatically collect malware that is transferred to it.

Low-interaction honeypots emulate only a limited subset of a system or service, and are generally easier to implement. They normally also consume fewer resources than high-interaction honeypots, allowing them to monitor a wide range of IP addresses using just one or a few computer systems to simulate the presence of many more. Low-interaction honeypots can not only simulate services or applications, but also known software vulnerabilities, so that they can be used to capture attacks specifically targeting these vulnerabilities. Nepenthes [mwc07b] is a software implementing a low-interaction honeypot that emulates known vulnerabilities that worms use to spread. It can interpret shellcode that is used in the initial exploitation attempt and capture malicious payloads following "successful" exploitation. Nepenthes is free software, modular and extensible.

2.2.2 Static Analysis

Static analysis is a generic term referring to analysis methods that do *not* involve *executing* the program to be analysed, for the sake of brevity henceforth called a *specimen*. Static analysis can be used to gather a variety of information about a specimen, 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.

Some of that information is not directly related to a specimen's purpose yet it can still be useful. Information about the file format, shared library or compiler can aid disassembly or decompilation. 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 (see also section 2.3.4).

Static analysis has several advantages over dynamic approaches. As static methods do not involve executing a potentially malicious specimen, 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 specimen 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.

2.2.2.1 Disassembly

Disassembly is the process of analysing a program and generating a textual representation of its machine code. A software that strives to achieve this goal is called a *disassembler*. Disassemblers face several challenges when trying to disassemble x86 machine code. First of all, the x86 architecture, being a von Neumann architecture, does not distinguish code and data. This means that data can be embedded in the instruction stream, which the disassembler might incorrectly try to decode as instructions.

Secondly, unlike RISC (reduced instruction set computer) architectures, where instructions are normally aligned on machine word boundaries, the x86 architecture is a CISC (complex instruction set computer) architecture, which allows for non-aligned instructions of varying length, from one up to more than ten bytes, making it harder to detect and recover from disassembly errors.

There are two typical approaches to disassembly, called *linear sweep* and *recursive traversal*.

Linear Sweep Linear sweep disassembly typically starts at the beginning of a program's code section, disassembling one instruction after another, until the end of the code section. Linear sweep is a very simple disassembly algorithm. Its main weakness is that it does not take a program's control flow into account, which might lead to it classifying data embedded in the instruction stream as executable code and incorrectly trying to disassemble it. These kinds of mistakes can only be detected if it results in the disassembler trying to disassemble an invalid opcode.

Malicious code can take advantage of this behaviour by inserting *junk bytes* into the instruction stream, at positions that are unreachable at runtime. These junk bytes have no effect on the program's execution but cause linear sweep disassemblers to incorrectly disassemble portions of the executable code. The following short assembly language snippet demonstrates the use of junk bytes to fool disassembly:

```
1   xor eax, eax            ; clear eax register, sets the zero flag
2   jz dest                 ; jump to dest if zero flag set
3   db 0xe9                 ; junk byte
4 dest:
5   mov eax, 0x7c39542c     ; load 0x7c39542c into eax
6   jmp eax                 ; jump to the address contained in eax
7   nop                     ; no operation
8   nop                     ; no operation
```

The following disassembly was generated by a linear sweep disassembler from the bytecode that was generated using the above code:

```
00000000 31 c0              xor         eax, eax
00000002 74 01              jz          0x5
00000004 e9 b8 2c 54 39     jmp         39542cc1
00000009 7c ff              jl          0xa
0000000b e0 90              loopnz      ffffff9d
0000000d 90                 nop
```

As one can see, the junk byte at offset 0x04 was used by the disassembler as the first byte of an unconditional jump instruction, even though the program's design precludes it from being executed. The following instructions are incorrectly disassembled as well, up to the last nop instruction, which is correctly disassembled again, exemplifying that x86 disassembly tends to self-repairing as noted in [LD03, section 3.1].

Recursive Traversal Recursive traversal disassemblers take a program's control flow into account and try to construct a *control flow graph* that is made up of *basic blocks*. A basic block is an instruction sequence that does not contain any branch instructions except possibly at the end and that does not contain any branch targets except at the beginning, Thus, if the first instruction of a basic block executes, all other instructions are executed as well. Basic blocks typically have one or more predecessor blocks, i.e., blocks that branch to it, and one or more successor blocks, i.e., blocks that it branches to. In conjunction with the branch relation, all basic blocks form the control flow graph,

i.e., a directed graph that describes all possible control flow between individual basic blocks within a program.

A recursive traversal disassembler starts disassembling instructions at a program's entry point and follows the instruction stream until it reaches a control transfer instruction. It then tries to determine the possible successors to that control transfer instruction and recursively continues disassembling at their respective addresses. For conditional branches and function calls (to functions within the program), the branch target and the instruction following the branch instruction under consideration would generally be considered valid successors, for unconditional branches the branch target only.

While this approach seems to solve the problem of incorrectly classifying data as code, it is not without flaws. It might not be possible to determine all successors to a control transfer instructions, e.g., for indirect branches, i.e., branches where the branch target is stored in a register, or memory location. It is possible to handle some kinds of indirect jumps correctly by using heuristics and data flow analysis, however, short of executing all possible code paths it is impossible in the general case.

The example shown above illustrates some of the difficulties faced by recursive traversal disassemblers as well. Even if the conditional branch in line 2 is designed to always execute after clearing the eax register, the disassembler would have to notice that this instruction sequence will cause the branch to be taken at all times, and it would have to assure that the program's control flow will never cause the branch to fall through. The latter could happen if some branch within the program were to jump directly to the branch instruction, without clearing the eax register first. A recursive traversal disassembler faces similar difficulties in line 6, as it might not be able to establish whether the eax register always has the value 0x7c39542c at the time the unconditional branch is executed. In the case of the conditional branch in line 2, the disassembler would end up disassembling an instruction sequence starting at the branch target, and one starting at the junk byte (the fall-through case), thus generating two conflicting instruction sequences which merge at the second nop instruction. In the case of the unconditional branch in line 6, a recursive traversal disassembler could use heuristics to determine the value of the eax register at the time the branch is executed, it might however miss other execution paths.

Another way to thwart recursive traversal disassembly is by obfuscating subroutine calls, i.e., branches using the `CALL` instruction. The `CALL` instruction is a branch instruction that pushes the address of the following instruction onto the stack before transferring control to the branch target. It is normally paired with a matching `RET` (return) instruction at the end of called function. The `RET` instruction pops an address off the stack and branches to that address. Recursive traversal disassemblers generally assume this behaviour, and thus traverse along the branch target, and the instruction following the `CALL`. However, functions can manipulate the return address on the stack to redirect control flow to locations not expected by the debugger. It is also possible to use a combination of `PUSH` and `RET` like a general-purpose unconditional branch instruction by first pushing the target address and then "returning" to it.

Other Approaches As the linear sweep and recursive traversal disassemblers obviously have trouble dealing with a variety of situations that might be found in x86 byte code, better methods are needed to counter these deficits.

One such method is *speculative* disassembly [CERL02]. Speculative disassembly essentially employs the recursive traversal method, but when there are no branch targets left for the disassembler to traverse, it restarts the disassembly for every gap in the code section. Should it encounter invalid instructions, it discards the disassembly for that gap and moves on to the next one.

A method that employs statistics to deal with obfuscated and hard to disassemble code was proposed in [KRVV04]. It starts by trying to identify functions to reduce complexity, by employing a heuristic that scans the byte stream of a program for *function prologues*, i.e., sequences of instructions that are commonly inserted by compilers at the start of subroutines to set up the local stack frame. The proposed method then deals with the intra-procedural control flow of the identified functions as follows: It attempts to decode instructions at every byte within those functions and tries to identify those that are intra-procedural branches, i.e. branches with at least one *known* successor basic block within the function. It uses these and the resulting basic blocks as a skeleton for the intra-procedural control flow graph. This control flow graph is a superset of the "real" CFG, and might contain *conflicting* basic blocks, i.e., basic blocks with overlapping address ranges that contain instruction sequences that cannot be merged. The proposed algorithm then removes those nodes that are not part of the real CFG as follows:

1. It marks the start of the function as valid. It then continues to recursively mark those basic blocks as valid, that can be reached from a valid basic block. The set of valid nodes will be equal to the set of nodes a recursive traversal disassembler would generate when invoked on the function's start address. After marking valid nodes, any node that is in conflict with a valid node can be removed from the control flow graph.

2. It is assumed that instructions cannot overlap. Thus, it is not possible to reach two conflicting nodes from a valid basic block. Remove all nodes that are the common ancestor of a two conflicting basic blocks.

3. For two conflicting basic blocks, it is assumed that the one that is more tightly integrated into the control flow graph is more likely to be valid. Integration of a node is approximated by the number of its predecessors. For any two conflicting nodes, remove the one with fewer predecessor nodes.

4. Outgoing edges of a node represent branches within the function, thus, valid nodes are likely to have more direct successors than invalid nodes. For two conflicting basic blocks, remove the one with fewer direct successors.

5. For any two conflicting basic blocks, remove one at random.

If there are gaps in the disassembly after performing these steps, the algorithm tries to identify the most likely valid instruction sequence within a gap guided by a collection of statistics about instruction sequences appearing in real-world binaries.

2.2.3 Dynamic Analysis

Dynamic analysis is a way of analysing an unknown program by executing it and observing its behaviour. When executing potentially hostile code, careful consideration must be given to securing the analysis environment, so as not to risk its destruction or even damage to other computer systems on the same network. The simplest solution to contain 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 [FV04]. There also exist hardware and software software solutions to automate the task of restoring a real machine to a pristine state.

However, a more common approach today is to contain malicious code within virtual machines. The advantage of this method is that they are easier to handle than real machines and provide greater flexibility, e.g., they oftentimes allow a researcher to save one or more snapshots of the machine state, which can later be restored. On the downside, virtual machines are normally slower than real machines, and there exist ways for code running on them to differentiate them from real machines, a property that hostile code can leverage to its advantage by changing its execution flow once it detects it is being run in a virtual environment. For a more thorough discussion of this subject, please see section 2.5.

Dynamic analysis can be performed at various 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. This can work well on operating systems where system calls are well documented, e.g., on Linux, BSD and other UNIX(-like) systems. On Windows operating systems, documentation of system calls is scarce. In fact, the official documentation does not cover most of the so-called *native API*, a library ("ntdll.dll") that wraps system calls, and is called into by the well-documented Win32 API to perform interaction with the operating system kernel. System call tracing can also quickly generate a lot of data that might be hard for a human analyst to process.

Thus, in the case of Windows operating systems, or, 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). Note that the specimen can still invoke system calls directly and thus bypass library call tracing.

In the extreme case, researchers can resort to *instruction-level tracing* or *internal call tracing*, which typically involves using a debugger to control execution of specimen. Dynamic analysis at this level can be a very time-consuming task, but it can be necessary

when analysing crucial portions of a program that do not make use of well-documented library functions or system calls, e.g. cryptographic algorithms. Additionally, tracing the control flow within a program can be helpful to identify subroutines and thus aid disassembly during static analysis.

Dynamic analysis has several advantages. At high levels of abstraction, e.g., library call tracing, it can quickly give a researcher an overview over what a certain program does (for an extensive discussion of this subject, see [Wil06]). It is largely immune to obfuscation techniques that target static analysis methods, like self-modifying code, including runtime-packing or -encryption, or anti-disassembly tricks. But there are also drawbacks. Dynamic analysis typically only covers one possible execution path through a program. There is currently ongoing research on remedying this limitation at the Secure Systems Lab at Technical University Vienna, documented in [MKK07]. Their software is not publicly available, however, and there appear to be no other tools implementing similar techniques.

2.2.3.1 Anti-Debugging techniques

There are several techniques that can be used to hamper debugging of an executable. This section shall give a brief overview of the challenges faced when debugging hostile code. More information on this topic (including example code) can be found in [Yas07] and [Fal07].

Detecting the presence of a debugger can be useful to a process that wants to avoid being analysed, as it can opt to behave differently in the presence of a debugger, e.g., by terminating with an error message, terminating silently, or by exercising code paths that are not normally used to skew the analysis. There are a variety of ways for a program to detect that it is being debugged. The Windows API even provides a `isDebuggerPresent()` function that returns the value of a flag within a user-mode structure associated with the current process (the *process environment block*, PEB), a similar method just checks the value of this flag directly, without invoking the API call. However, this method can easily be circumvented by resetting that flag, as it is purely informational. Other, similar methods use less well-known flags within process-management related data structures, like the *NtGlobalFlag* within the PEB, or less well-known API calls, such as `CheckRemoteDebuggerPresent()` or `NtQueryInformationProcess` to check for the presence of a debugger. Other detection methods are based on the fact that debuggers typically handle certain exceptions without passing them to the program that is being debugged, such as the breakpoint exception (interrupt 3) or the debug exception (interrupt 1). Applications can thus detect the presence of a debugger by registering an exception handler which could for example change the value of a flag within the application and then manually causing a debug or breakpoint exception by executing an `INT1` or `INT3` instruction, checking the value of the flag afterwards. If it is unchanged, the presence of a debugger is likely. Other methods use timing checks to determine whether execution of the program takes considerably longer than expected, indicating that some time is being spent inside of a debugger, or try to find well-known debuggers in the window or process list.

Some more offensive techniques try to interfere directly with a debugger's operation. One common method is to use the operating system's exception handling mechanism and custom exception handlers so that the application can throw exceptions on purpose to redirect control flow. Debuggers typically catch most exceptions, halt the debugged program and present the exception to the human analyst to decide what to do. Throwing many exceptions as part of normal operation can make debugging cumbersome and error-prone. Other methods include attempts to block input so that a reverse engineer cannot control the debugger anymore, asking the operating system to not generate debug events for an application's threads, disabling hardware breakpoints by modifying debug registers, or trying to locate software breakpoints (`INT3` instructions) and overwriting them.

2.3 Runtime Packed Binaries

A *runtime packer* is a program that takes an existing executable file, compresses or encrypts it, and attaches a decompression or decryptor stub to it to create a new executable file. When that file is run, the stub decompresses or decrypts the original file to memory and transfers control to its *original entry point*, so that it can execute just like the original executable file. For the purposes of this thesis, the terms *packer*, *packing*, *unpacker* and *unpacking* shall refer to both compression and encryption of executables.

Malware commonly uses runtime packers and executable protectors to prevent researchers from successfully reverse engineering it, and also to evade signature-based malware detectors. Because the main target for malware is the Windows platform, of which Windows XP SP2 for the x86 architecture is the most prevalent, this thesis will deal exclusively with executable packers for that platform.

This section will first detail the executable file format used on the Windows Platform in section 2.3.1, and then highlight typical behaviour for packers operating on such files in section 2.3.2.

2.3.1 PE File Format

This section will give an overview of the *Portable Executable (PE) and Common Object File Format (COFF)* as specified by Microsoft [Mic06], in the remainder of this thesis referred to as *PE file format*. Some more information can be found in [Pie02a, Pie02b].

The PE file format is an architecture independent file format for *image* and *object files* on Microsoft Windows operating systems. Relevant to this thesis are only image files, i.e., files that can be executed directly by Microsoft Windows (normally carrying the *.exe* filename extension), or *Dynamic Link Libraries* (DLLs) (normally carrying the *.dll* filename extension).

2.3.1.1 Headers

For backwards compatibility, image files retain a MS-DOS 2.0 compatible section, which consists of the *EXE Header*, a stub program that normally outputs "This program cannot be run in DOS mode", should an attempt be made to execute the image under DOS, and, at offset 0x3c, the offset where the four-byte *PE Signature*. "PE\0\0" can be found in a valid image. This signature is followed by the *PE Header*, which consists of the *COFF File Header* and the *Optional Header*.

The COFF File Header is mandatory for image and object files. It contains information about what CPU architecture the image file is intended for, the number of sections (see also section 2.3.1.2), the size of the Optional Header, and some information that further describes the properties of the file.

The Optional Header is optional in the sense that some files, specifically object files, do not require it. For image files however it is mandatory. For image files for Microsoft Windows, the Optional Header consists of three parts, the *Optional Header Standard Fields*, the *Optional Header Windows-Specific Fields* and the *Optional Header Data Directories*.

The Optional Header Standard Fields contain a magic value indicating whether the image file is a PE32, or a PE32+ file. The former is used on 32 bit Windows systems, whereas the latter is used on 64 bit Windows systems. This section will concentrate on PE32, as the majority of Windows installations currently in use are 32 Bit machines and both file formats differ only slightly. The most important other field contained in the Standard Fields is the entry point, i.e., the address of the first instruction executed when the image has been fully initialized by the operating system.

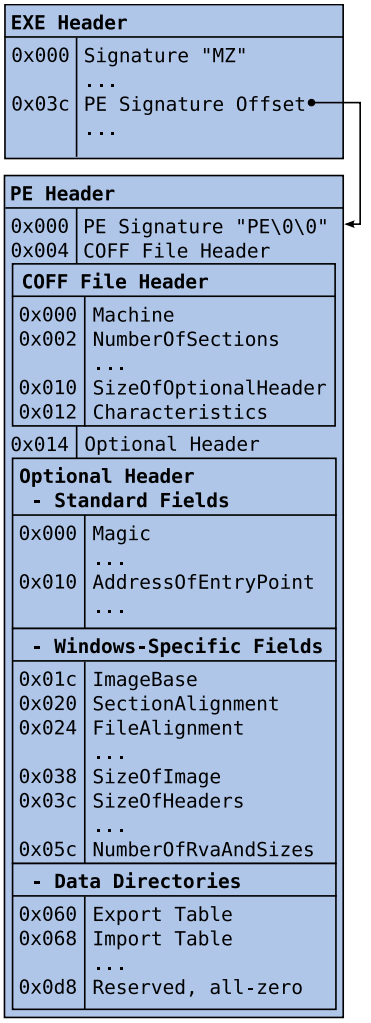


Figure 2.1 – PE Headers

The Optional Header Windows Specific Fields contain information that is required by the image loader and dynamic linker in Windows. Relevant to this thesis are the following fields:

- *ImageBase* – the preferred start address of the image in memory, which must be a multiple of 65536. For executables, typical values are `0x01000000` and `0x00400000`.
- *SectionAlignment* – sections loaded into memory are aligned on multiples of this value. It defaults to to the page size of the architecture the file is intended to be used on.

- *FileAlignment* – raw data of sections in the image file on disk is aligned on multiples of this value. It must be smaller than or equal to the section alignment. If the section alignment is smaller than the architecture's page size, file alignment and section alignment must be equal.
- *SizeOfImage* – the size of the image when it is fully loaded into memory. This value must be a multiple of SectionAlignment.
- *SizeOfHeaders* – the sum of the sizes of the MS-DOS section, the PE header and the section headers, rounded up to a multiple of FileAlignment.
- *NumberOfRvaAndSizes* – the number of entries in the Optional Header Data Directories.

The Optional Header Data Directories contain *Relative Virtual Addresses* (RVA; offsets relative to the image base) of several important data structures, most notably of the export table and import table, which are described in more detail in sections 2.3.1.3 and 2.3.1.4.

2.3.1.2 Section Table

The Section Table immediately follows the Optional Header. None of the header fields contain a pointer to it, so the only way to locate it is by determining the offset of the first byte after the Optional Header.

The entries of the Section Table are called *Section Headers*. They describe how the contents of the image file should be mapped into memory by the Windows loader. Each section header describes a single *section*, i.e., a continuous piece of code or data sharing certain characteristics.

The NumberOfSections field of the COFF File Header determines how many Section Headers there are in the Section Table. The Windows loader is limited to a maximum of 96 Section Headers.

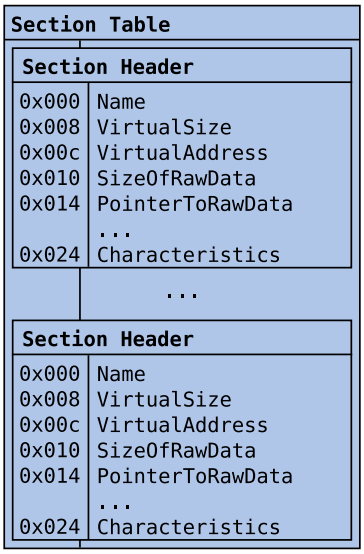


Figure 2.2 – Section Table

The *Name* field of a section header can be used to name a section. It is supposed to be an UTF-8 encoded string, zero-padded to a maximum length of 8 bytes, but the Windows loader does not validate whether this field contains valid UTF-8 data. In practice, it can contain arbitrary data up to a size of 8 bytes. Common section names are ".text" for a section containing program code and ".data" for a section containg program data.

The *VirtualSize* field determines how much memory a section occupies. It should be a multiple of the SectionAlignment field from the optional header. If it is not, the

Windows loader rounds this value up to the next multiple of SectionAlignment. The *SizeOfRawData* field on the other hand determines the size of the section on disk. It must be a multiple of the FileAlignment field from the Optional Header and less than or equal to the value of VirtualSize. If it is smaller than VirtualSize, the Windows loader pads the difference with zero-bytes.

VirtualAddress contains the address of the first byte of the section when loaded into virtual memory, relative to the image base. *PointerToRawData* contains the offset of the first byte of the section within the image file on disk.

Characteristics is a bit field that is used to describe the properties of the section, e.g., whether is readable, writeable or executable, whether it contains code or data (initialized or uninitialized).

Figure 2.3 shows how headers and sections are mapped into memory and zero-padded to fulfil the alignment requirements.

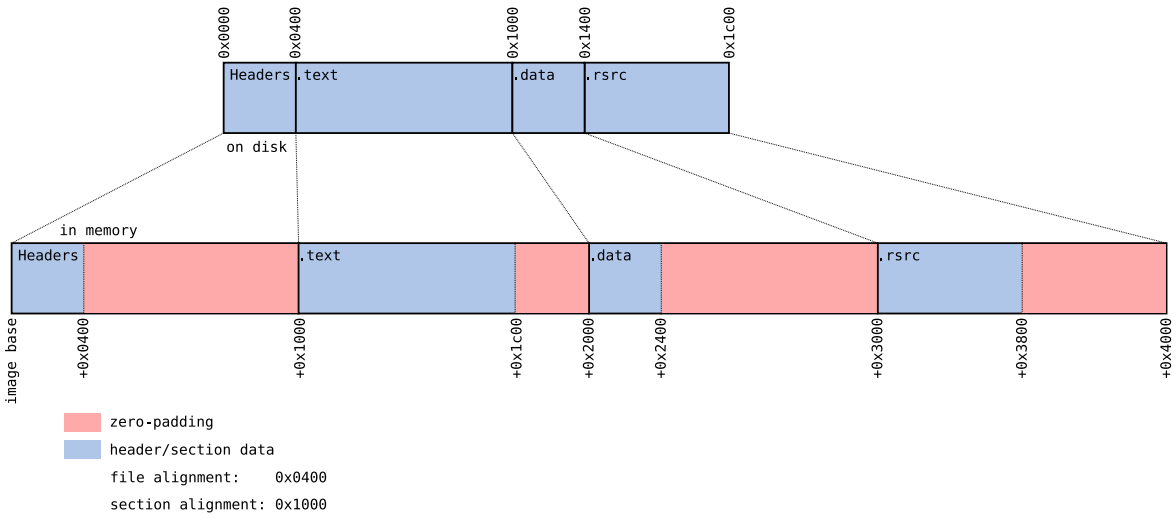


Figure 2.3 – Section expansion

2.3.1.3 Export Data Section

Dynamic link libraries normally *export* symbols (i.e., functions or variables) for other PE images to *import*. Importing these symbols generally involves loading the respective library into memory and obtaining a reference to (i.e., the virtual address of) the symbol.

The data structures describing exported symbols are stored in the export data section. While the PE specification describes it as a section named ".edata", few PE images today actually contain a section with that name. Out of more than 1200 DLLs from the Windows system directory, less than 20 contain an ".edata" section. Export data can be stored in any section in an image that contains initialized data and is marked as readable. Each exported symbol is associated with an *ordinal* number, that can be used

to locate the symbol within the exporting image, for importing the symbol *by ordinal*. For developers' convenience, DLLs typically also associate their exported symbols with an ASCII string, so that they can also be imported *by name*. Furthermore, DLLs can also *forward* symbols to symbols in another DLL, essentially creating an alias for the symbol forwarded to. The *Export Table* data directory in the Optional Header contains a pointer to the export data, and its size. The export data begins with the *export directory table*, which is a data structure that contains, among others, the following fields (see figure 2.4):

- *Name RVA* – the RVA of an ASCII string containing the DLL's name.
- *Ordinal Base* – the lowest ordinal value of a symbol exported by the DLL. When importing a symbol from the DLL by ordinal, this value is subtracted from the ordinal and the result is used as an index into the export address table to acquire the symbol's address.
- *Address Table Entries* – the total number of exported symbols.
- *Number of Name Pointers* – the number of entries in the export name pointer table and in the export ordinal table, i.e., essentially, the number of symbols exported by name.
- *Export Address Table RVA* – the RVA of the *export address table*, i.e., the table that contains the RVAs of all exported symbols
- *Name Pointer RVA* – the RVA of the *export name pointer table*, i.e., a table of RVAs of symbol names. Its entries are in lexicographical order.
- *Ordinal Table RVA* – the RVA of the *export ordinal table*. The export ordinal table contains 16-bit wide indexes into the export address table.

The export name pointer table and the export ordinal table are used as a single table for locating a named symbol's address. To locate a symbol by name, a binary search is performed on the export name pointer table until the name is found. The position of that name within the export name pointer table is then used to index into the export ordinal table to yield the symbol's ordinal (offset by the ordinal base). That can then be used to index into the export address table and locate the symbol's RVA.[1] If that RVA points to *within* the export section (as specified by the export data directory), it is a *forwarder RVA*, i.e., it is an alias for a symbol in another DLL. In that case the RVA points to a specially formatted ASCII string that starts with the other DLL's name (without the ".dll" filename extension), followed by a dot, and the name of the symbol within that DLL. For example, the `HeapAlloc` symbol exported by `kernel32.dll` is forwarded to the symbol `RtlAllocateHeap` within `ntdll.dll` by using the forwarded name `NTDLL.RtlAllocateHeap`.

[1] According to the PE specification, the ordinal base needs to be subtracted from the value within the export ordinal table to yield an index into the export address table. However, experiments and review of some open source PE parsers [Car07, WB07] showed that this is not the case.

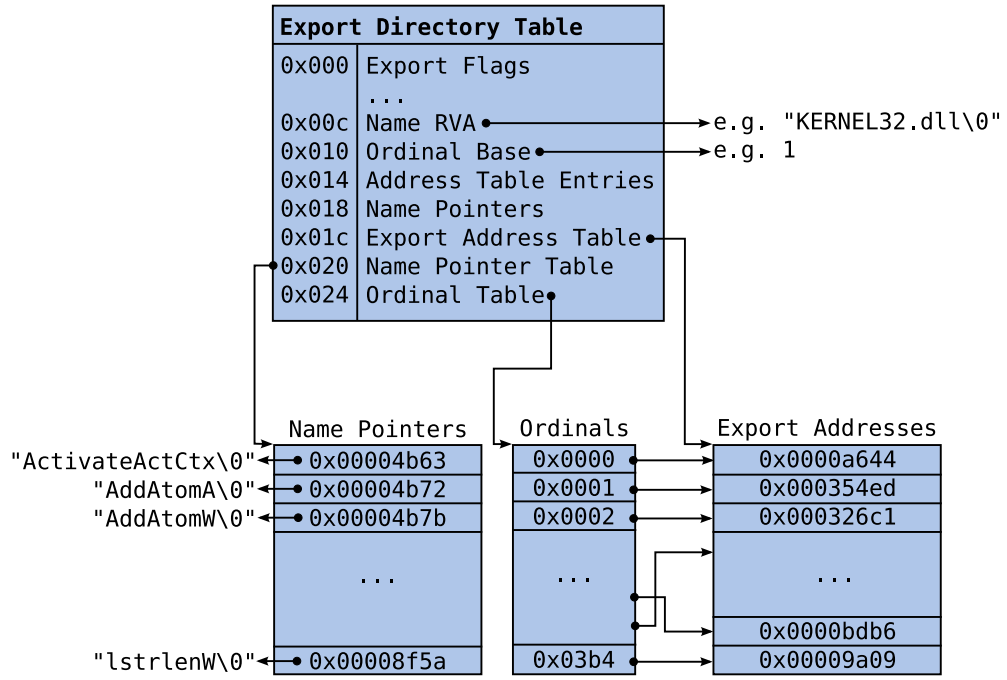


Figure 2.4 – Export Data

2.3.1.4 Import Section

Image files, i.e., both executable files and dynamic link libraries, can import symbols that are exported by other images files. The *import section* details which symbols an image file wants to import. While the PE specification describes the import section as a single section named ".idata", only 20 out of almost 1500 image files from a default Windows XP SP2 installation actually contained a section of that name. In fact, import data can reside in any section that contains initialized data and is marked as read- and writeable.

The import table data directory from the optional headers specifies the location and size of the *import directory table*. Each entry within the import directory table relates to one imported DLL and contains, among other data, the RVAs of the DLL's name (e.g., "kernel32.dll"), its *import lookup table* and its *import address table*. The import directory table is terminated by an all-zero entry.

The import lookup table and the import address table are arrays of 32-bit integers that describe which symbols to import from the corresponding DLL. If an entries' most significant bit is `1`, the symbol is imported by ordinal, and the 16 least significant bits specify the ordinal value. Otherwise the symbol is imported by name and the 31 least significant bits specify the RVA of a *hint/name table entry*. Hint/name table entries consist of a 16-bit hint, a zero-terminated ASCII string containing the symbol name, and a zero byte, should the symbol name not be 16-bit aligned. The hint is used as an index into the imported DLL's export name pointer table and a first match with the imported symbol's name is attempted using the export name at that position. Should

the match fail, a binary search is performed on the export name pointer table to locate the symbol.

The import lookup table and the import address table are terminated by an all-zero entry. Both tables essentially contain the same data, but the import address table entries are overwritten with the actual imported symbol's virtual addresses when the image is bound. Binding is typically performed by the Windows loader when the image is loaded into memory, but an image can also be bound explicitly. Imported functions are typically called using indirect branch instructions that reference entries from the import address table.

Note that PE files typically still execute correctly without an import lookup table, as long as the import address table is present in its unbound form, or if the DLLs it was originally bound to have not been modified since.

2.3.2 PE Packers

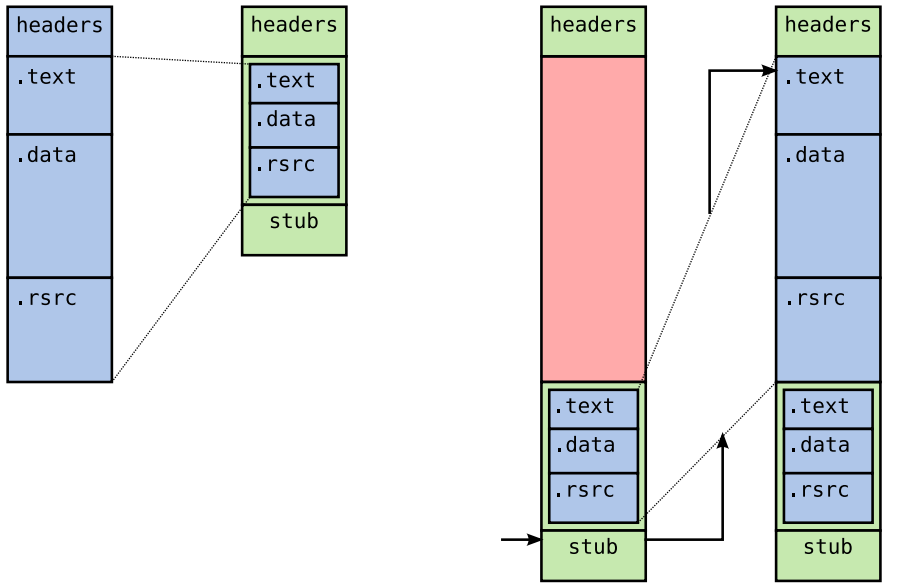
In general, runtime packers compress the original executable and attach an unpacking stub to it. Upon execution of the packed executable, the stub unpacks the original code (and data) and transfers control to it. PE Packers typically follow that scheme as well. This section will elaborate on some of the details of the operation of PE Packers in particular. Note that deviations from the described behaviour are possible.

As depicted in figure 2.5(a), 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 (see figure 2.5(b)), the unpacking stub is executed which unpacks the packed code into the empty section reserved for it. Then (not depicted), 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.

2.3.2.1 Protection

Some runtime packers not only compress or encrypt executable files, but also try to protect the unpacking stub from reverse engineering attempts, or even obfuscate parts



(a) compression – file view (b) decompression – memory view

Figure 2.5 – Typical operation of PE packers

of the original code. Such packers are typically called *executable protectors*. Protection of the unpacking stub is typically attempted by employing code obfuscation and anti-debugging techniques, as presented in section 2.2. Some executable protectors rely on more advanced schemes should the protections of the unpacking stub fail.

Stolen Bytes To obfuscate the original entry point, or sometimes also entry points of API or other functions, some protectors employ a method called *stolen bytes*. While the origin of this term is unknown, it is commonly used in unpacking tutorials posted on reverse engineering community websites. This method works by copying a number of bytes representing an instruction sequence from a branch target to another location, and typically overwriting the original bytes with other data. Then, all branches to the original code are redirected to the copied instructions, and a branch to to the first instruction beyond the original instruction sequence is appended to the copied instruction sequence. Sometimes the copied instruction sequence is further obfuscated by transforming it into a different, yet equivalent instruction sequence, e.g., by inserting `NOP` instructions or junk bytes, by reordering independent instructions, or by modifying register usage.

This obfuscation method serves two purposes. Firstly, it makes it harder to identify the original entry point, as well as calls into API functions. Secondly, it moves parts of the original code to other locations, which makes it harder to acquire a contiguous dump of the original code.

Shifting Decode Frame A technique dubbed *shifting decode frame* in [QV07] does not unpack all the original code at once. Instead it unpacks only a single basic block of

code at a time, executes it, and discards (e.g., by overwriting) it before unpacking and executing the next block. This will lead to only a fraction of the original code being in memory at any point of time so that approaches that try to recover the original code by simply dumping select parts of the virtual address space fail. Additionally, this method only unpacks those parts of the original code that are actually executed, so that methods that rely on dynamic analysis to trace execution of the unpacking process and try to dump parts of the unpacked code at appropriate times cannot guarantee completeness (see also section 2.2.3).

Virtualization A rather extreme obfuscation technique that is employed by some executable protectors (e.g., by Themida [Ore07], a commercial executable protector), is *virtualization* of parts of the original code. When protecting an executable, such protectors transform parts of the original instructions into equivalent instructions for a virtual CPU. This virtual CPU is implemented by the unpacking stub, it executes these new instruction sequences when the protected executable is run, and thus, the original code is never seen in the clear during execution of the protected executable. Static analysis of executables protected with virtualization is hard, as it typically involves reverse engineering the virtual CPU and its instruction set, and then reverse engineering the instruction sequences that are being executed on it.

2.3.3 Unpacking

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 runtime-packers and executable protectors try to prevent these unpacking methods by hardening their unpacking stubs against reverse engineering, e.g., by using anti-debugging and code-obfuscation techniques.

It should be noted that some general-purpose executable packers, e.g., UPX, even support unpacking out of the box.

2.3.4 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. While determining whether

a program contains hidden, packed code or not is undecidable in the general case (see also [RHD+06]), there are heuristics that give good estimates.

Signature-based packer detection Just like malware can be detected using signatures (see section 2.1.3), the same holds for the unpacking stubs used by executable packers. There exist a variety of tools for this purpose, but the most popular is PEiD [pei07], a signature-based packer identifier with plugin support.

Entropy Analysis In [LH07], the authors propose to use an entropy-based metric to detect packed executables. Entropy, or information density, is a term from information theory describing the amount of information that each symbol within a series of symbols carries.

In the simplest case, PE files (or parts thereof) can be represented as byte strings, so that a common approach is to use individual bytes as symbols, meaning that there are $2^8 = 256$ individual symbols, representing values from 0 to 255. Within a series M of N such symbols, each symbol X has an *absolute frequency* of P_X and a *relative frequency* of $p_X = \frac{P_X}{N}$. The *self-information* $I(X)$ of a symbol X is defined as

$$I(X) = log(\frac{1}{p_X}) = -log\ p_X$$

Self-information is a measure describing how much information a symbol carries. The less likely it appears in a message, the higher its self-information, and vice versa. If a message contains only repetitions of the same symbol X ($p_X = 1$)), it does not carry much information ($-log\ 1 = 0$), however, if a certain symbol appears rarely within a message, it carries much more information ($p_X \to 0 : -log\ p_X \to \infty$). Self-information is measured in *bits* if the binary logarithm used. The *entropy* of a series M of bytes is defined as the average self-information across all symbols.

$$H(M) = \sum_{X=0}^{255} p_X\ I(X) = -\sum_{X=0}^{255} p_X\ log\ p_X$$

Like self-information, it is usually measured in bits, depending on the base of the logarithm used. It is highest, when all symbols appear equally often within M, so that for byte sequences, the maximum entropy is 8 bit:

$$\begin{aligned} H_{max} &= -\sum_{X=0}^{255} p_X\ log_2\ p_X \\ &= -\sum_{X=0}^{255} \frac{1}{256}\ log_2\ \frac{1}{256} \\ &= -256\ \frac{1}{256}\ log_2\ \frac{1}{256} \\ &= log_2\ 256 \\ &= 8 \end{aligned}$$

Both packing and encryption transform one byte sequence into another, where the new byte sequence typically has a 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.

The authors of [LH07] measured entropies of plain text files and native, packed, and encrypted Windows executables. They split each sample file into 256 byte sized blocks and recorded its average block entropies and its highest block entropy. They then analysed the aggregated entropy scores statistically and found significant differences between files of different types leading them to the conclusion that entropy metrics are indeed a valid heuristic to determine whether an executable has been packed or encrypted. Experiments showed that a similar metric, based on the maximum entropy of sections within PE files, yields similar differences between unpacked and packed or encrypted files.

Note that 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.

Execution A behaviour-based approach for determining whether an executable is packed is monitoring the execution of a binary and checking whether it at some point executes memory that was previously modified, or, as presented in [RHD+06] (see also section 2.6), whether it executes code that does not exist in a static model of the binary. While this seems a valid method at first glance, it cannot distinguish between programs that contain packed code, and programs that contain self-modifying code.

The *pefile* [Car07] Python library implements signature-based and entropy-based approaches for packer detection. It can match PE files against PEiD-compatible signature databases and it has built-in functionality to calculate the entropy of individual sections within PE files.

2.4 x86 Memory Management

The Intel x86 architecture (also called IA-32 architecture) is the most common architecture used in personal computers today. It provides two memory management facilities, namely *segmentation* and *paging*.

Segmentation can be used to divide a processor's *linear address space* (on the x86 architecture: a 32-bit addressable memory space), into several, possibly smaller, isolated address spaces, to isolate code, data and stack of a single program from another, or to isolate different programs from one another, or both.

Paging provides a mechanism to provide *virtual memory* to the operating system and programs. The basic idea of virtual memory is to provide each program a large, contiguous address space, that can even exceed the amount of available physical memory. Paging can also be used to provide some isolation between individual applications.

The following two sections describe segmentation and paging in greater detail. Note that some intricacies that are not relevant to this thesis have been omitted, while others that are not strictly relevant but are essential to understanding the presented concepts have been included.

More information about the technical details of memory management on the x86 architecture can be found in [Int06b], more information about the underlying concepts and algorithms in [Tan01].

2.4.1 Segmentation

Segmentation is a mechanism to divide a processor's linear address space into smaller address spaces called *segments*. The original idea of segmentation was to be able to separate code, data and stack of a program from another, and also separate different programs from one another. The CPU enforces that no program can write to another program's segments, and it can also limit the types of memory access permitted to a segment, i.e., reading, writing, or execution.

When accessing memory within a particular segment, this memory is addressed using a *logical address*, consisting of a 16-bit *segment selector* which uniquely identifies a particular segment, and a 32-bit offset. The offset is added to the segment's *base* address within the CPUs linear address space to form a *linear address*. The CPU makes sure that the attempted access stays within the segment's bounds and conforms to the segment's permissions.

To store information about segments, the CPU utilizes two kinds of tables, the *global descriptor table* (GDT), and *local descriptor tables*, which each store up to 8192 *segment descriptors*, the GDT storing ones that are global to all processes, an LDT storing ones that are local to a certain process. The location of the GDT is stored in the *global descriptor table register* (GDTR) and the location of the LDT is stored in the *local descriptor table register* (LDTR).

Segment descriptors contain the following information (see figure 2.6, some flags have been omitted here for brevity):

- The *base* field is a 32-bit field containing the linear address of the first byte within the segment.

- The *limit* field is a 20-bit field specifying a segment's size. If the *granularity* (G) flag is clear, it is interpreted as is, for sizes between 1 byte and 1 MByte. If the granularity flag is set, the value of the limit field is multiplied by 4096, to form actual limits between 4096 bytes and 4 GBytes, in 4096-byte increments.

  For *expand-up* segments, the offset of a logical address may be in the range from 0 to the segment limit. Offsets greater than the segment limit cause the CPU to generate a general protection exception.

  For *expand-down* segments, the offset of a logical address may be in the range from the segment limit to `0xffffffff` (4 GBytes).

- The *descriptor privilege level* (DPL) specifies the segment's *privilege level* as a value from 0 (most privileged) to 3 (least privileged). Privilege levels will be explained in section 2.4.1.2
- When the *descriptor type* (S) flag is , the segment descriptor is for a system segment, otherwise it is for a code or data segment.
- The *type* field is dependent on the descriptor type flag and determines the type of segment, what kind of access it permits, and the direction of growth. More detail on this field will be given in section 2.4.1.1.

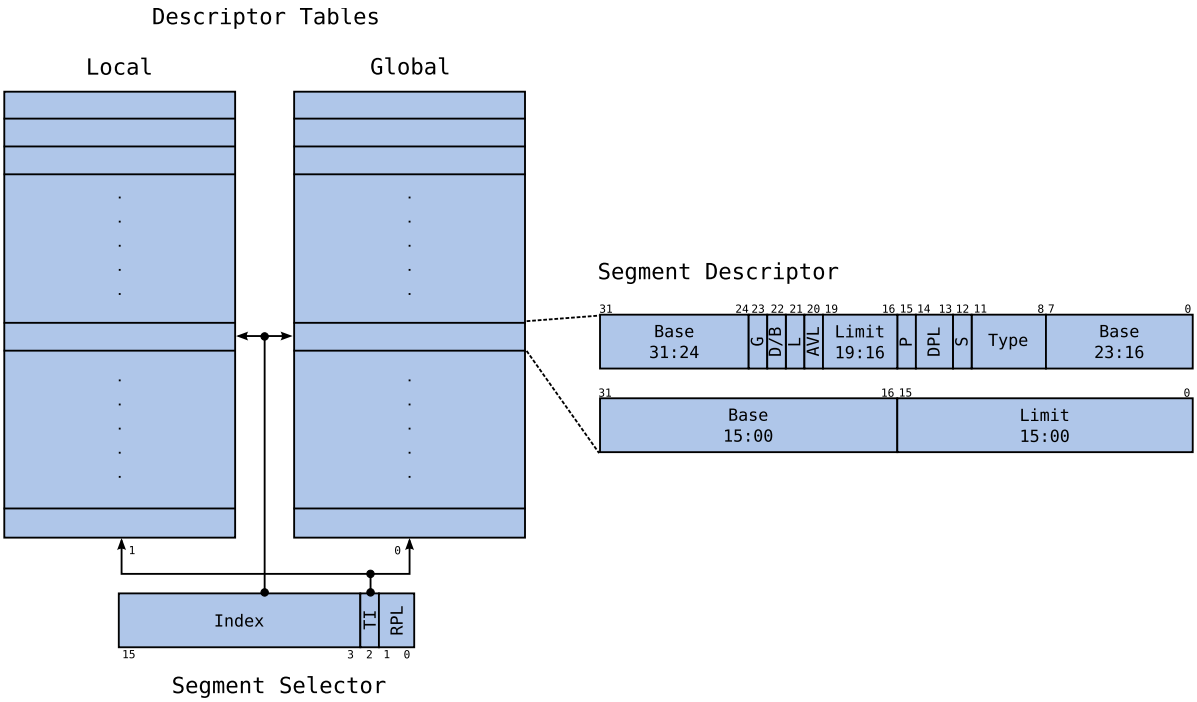


Figure 2.6 – Segmentation on the x86 architecture

Segment selectors are used to locate a segment descriptor in one of the descriptor tables. A segment selector's *table indicator* (TI) flag selects the GDT if clear and the LDT if set. The upper 13 bits index into the selected table. More detail on the *requested privilege level* (RPL) field will be given later. Segment selectors are utilized by loading them into one of 6 *segment registers*. Normally, at least the CS (code segment), DS (data segment) and SS (stack segment) registers need to contain valid segment selectors on the x86 architecture. Three additional registers, named ES, FS and GS, are available so that a maximum of six segments can be used at a time. When a segment selector is loaded into a segment register, the CPU fetches the corresponding segment descriptor and stores the information contained therein in a "hidden" part of the segment register (the *descriptor cache*) to speed up translation from logical to linear addresses.

2.4.1.1 Segment Types

A segment descriptor's type field is interpreted depending on the value of the descriptor type (S) flag. If the latter is clear, the descriptor is for a system segment as detailed in section 2.4.1.3. If the descriptor type (S) flag is set, the descriptor is for a *code* or *data* segment and the type field is interpreted accordingly.

If bit 11 is clear, the descriptor describes a data segment. Bit 10 is then interpreted as the *expansion-direction* (E) flag, bit 9 as the *write-enable* (W) and bit 8 as the *accessed* (A) flag. The flags' names are pretty much self-explanatory. The accessed flag is set by the CPU whenever a segment is accessed. It can be cleared by operating system software. The write-enable flag specifies whether the segment can be written to (the flag is set), or whether it is read-only (the flag is clear). The expansion-direction flag indicates whether the segment is expand-up (the flag is clear) or expand-down (the flag is set) and effects the interpretation of the limit fields.

If bit 11 is set, the descriptor describes a code segment. Bit 10 is then interpreted as the *conforming* (C) flag, bit 9 as the *read-enable* (R) flag and bit 8 as the *accessed* (A) flag. The accessed flag works just like it does for data segments. When the read-enable flag is clear, a code segment can be executed only, i.e., the segment cannot be read, except implicitly by the CPU fetching a new instruction, if the flag is set, the code segment can be read as well. A code-segment is said to be conforming if the corresponding flag is set, otherwise it is said to be nonconforming. The implications of this are tightly interwoven with the concept of privilege levels which will be presented in the following section (2.4.1.2).

2.4.1.2 Privilege Levels

As mentioned earlier, the x86 architecture knows 4 different *privilege levels*. These privilege levels are commonly also called *protection rings*, with ring 0 (most privileged) being innermost and ring 3 (least privileged) being outermost.

Most operating systems today do not use rings 1 and 2, so that ring 0 is used for the operating system kernel and ring 3 for applications. Segmentation and privilege levels are tightly interwoven on the x86 architecture.

The *current privilege level* (CPL) is, as the name suggests, the privilege level the CPU is operating at at a certain point in time. It is stored in the two low-order bits of both the CS and SS segment registers.

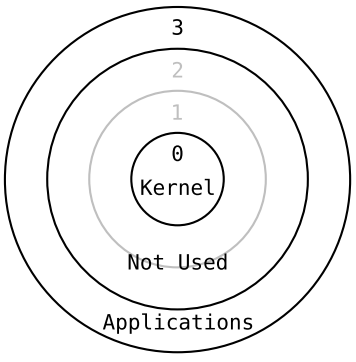


Figure 2.7 – Protection Rings

The *descriptor privilege level* (DPL) is the privilege level of a segment and is stored in its segment descriptor's DPL field. For a data segment, the DPL indicates the minimum privilege level needed for accessing it, e.g., if the DPL of a data segment

is 3, code running at CPL 0 to 3 can access it, whereas if it is 0 it can only be accessed by code executed at CPL 0.

For a conforming code segment, the DPL specifies the most privileged CPL at which it can still be accessed. If, for example, a conforming code segment has a DPL of 3, it can only be accessed if the CPL is 3, if its DPL is 0, it can be accessed regardless of the CPL. For nonconforming code segments, the DPL mandates at which CPL the segment can be accessed.

The two low-order bits of a segment selector contain a *requested privilege level* (RPL). When using a RPL that is not equal to the CPL, the least privileged of both is used for privilege checks when accessing the segment. This mechanism can be used by higher-privileged operating system services that need to access data structures on behalf of less privileged code, to guarantee that the less privileged code is allowed to access that segment.

2.4.1.3 System Segments and Gate descriptors

When the descriptor type flag of a segment descriptor is clear, it is a *system descriptor*. There are two types of system descriptors, *system-segment descriptors* and *gate descriptors*.

System-segment descriptors either describe local descriptor table (LDT) segments or task state segments (TSS). As the name suggest, an LDT segment holds a local descriptor table. It can be used by loading the associated segment selector into the local descriptor table register (`LDTR`), using the `LLDT` instruction. Task state segments store information associated with a hardware *task*, i.e., "a unit of work that a processor can dispatch, execute and suspend" [Int06b, section 6.1]. The x86 architecture provides facilities for saving a task's state and switching between tasks. At least one task must be defined when the CPU is operating in protected mode. Most current x86 operating systems seem to use the architecture's hardware task switching mechanisms only sparingly, e.g., for handling some exceptions, but not for switching between user-mode processes.

There are four different kinds of gate descriptors, *call gates*, *trap gates*, *interrupt gates* and *task gates*. Call gates allow a program to transfer control between different privilege levels in a controlled manner. A call gate contains the selector of the target code segment, an entry point as an offset within that segment, the minimum privilege level for the caller to be allowed to use the call gate, some information related to stack management and whether the call gate is valid.

Task gate, trap gate and interrupt gate descriptors are used within the interrupt descriptor table (IDT) to specify what action to take should an exception occur. Similar to call gates, trap and interrupt gates specify a target code segment and an entry point into that segment to which control is transferred in the event of an exception. Task gates on the other hand contain the selector to a task state segment describing a task that handles the exception.

2.4.1.4 Segmentation in Windows XP

The properties that have been described in [Sch01] for Windows 2000, were verified to hold for Windows XP as well. This is not very surprising as the latter is a close successor to the former. Windows XP essentially uses a flat virtual address space in both user and kernel mode, meaning that the default code, data and stack segments have a base of `0` and a limit of `0xffffffff`. The only variation on this theme are the segments typically accessed through the `FS` register, which are used to hold important data structures that are related to thread and process management, namely the *thread environment block* (TEB) when in user mode (privilege level 3, segment selector `0x0038`) or the *processor control region* (PCR) when in kernel mode (privilege level 0, segment selector `0x0030`).

2.4.2 Paging

The concept of *virtual memory* was originally introduced to be able to run applications that are larger than the available physical memory. Modern multitasking operating system extend this concept to allow the sum of the sizes of all applications to exceed available physical memory and to provide isolation between individual applications, and between applications and the operating system kernel.

The technique used to implement virtual memory on most operating systems for the x86 platform is called *paging*. With paging it is possible to provide each process with its own virtual address space, spanning the whole 32-bit-addressable linear address space (from 0 to $2^{32} - 1$, i.e., four gigabytes). This linear address space is divided into *pages*, which are normally four kilobytes (4096 bytes) in length, although sizes of two or four megabytes are also possible. To use an individual page it must reside in physical memory, in a so-called *page frame*.

When a program tries to access a memory location either implicitly, e.g., when an instruction is fetched, or explicitly, e.g., when instructions specify a memory location for one of their operands, it needs to be determined whether the page is currently mapped into physical memory. If it is, the linear address is translated into a physical address, otherwise, a *page fault exception* is generated that needs to be handled by the operating system, which must either load the missing page into physical memory, or signal the to the faulting process that an error occurred.

2.4.2.1 Address Translation

Address translation is normally handled by a *memory management unit* (MMU), which, on the x86 architecture, is integrated into the CPU. To translate a linear address to a physical address, the MMU uses a data structure called page table. Each *page table entry* (PTE) contains information about a single page, e.g., whether it is currently in memory, and if so, which physical page frame it is mapped to, whether it has been recently accessed or modified, and whether read and/or write access is permitted to that page. Page tables are indexed by page number, i.e., the linear address of a page, divided by the page size.

Assuming a page size that is a power of two, e.g., the typical page size of $4096 = 2^{12}$ bytes on the x86 architecture, the page number can also be determined by just taking the $32 - 12 = 20$ high-order bits of a 32-bit linear address. The remaining 12 low-order bits of the linear address determine the offset into the page, as well as the offset into the page frame.

A single page table covering the whole four gigabyte virtual address space would, using a page size of 4096 bytes, need to contain $2^{32}/4096 = 1048576$ entries. At four bytes per page table entry, this would amount to four megabytes for a single page table that would need to be kept in memory at all times. However, as the linear address space is usually only sparsely populated, such a large page table would occupy more memory than needed. As each individual process typically occupies its own virtual address space, and thus needs its own page table as well, the concept *multi-level paging* was introduced to work around this limitation. It introduces another level of indirection in address translation the *page directory*. The page directory is very similar to a page table, with the exception that it does not map linear addresses to page frames, but linear addresses to page tables, so that apart from the 4096 byte sized page directory only those page tables need to be kept in memory that describe parts of a process's address space that are actually in use.

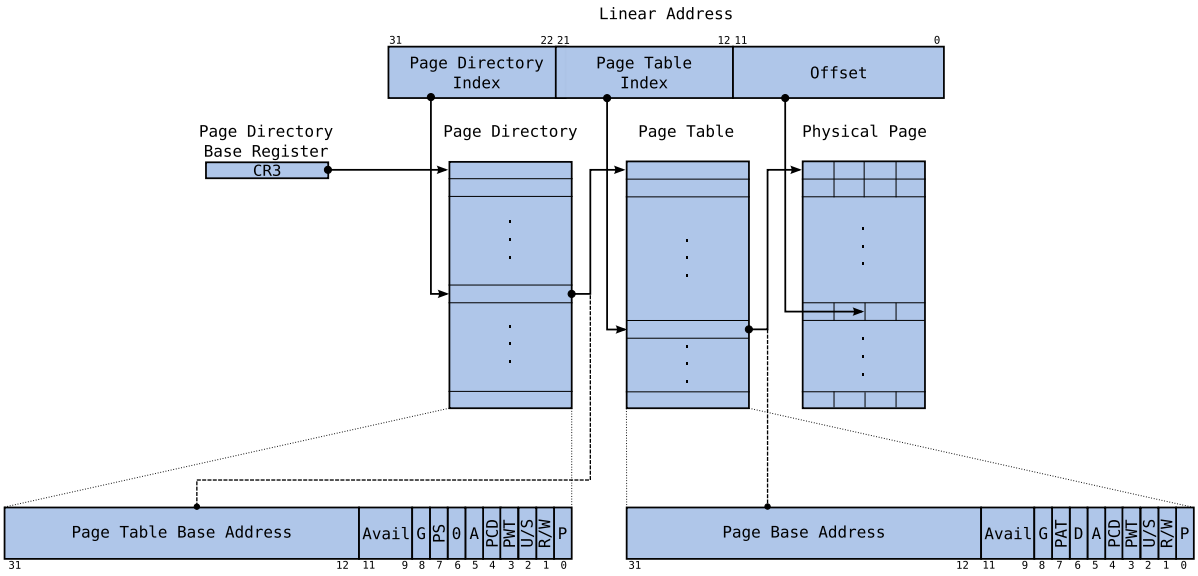


Figure 2.8 – Paging on the x86 architecture

On the x86 architecture, the *control register 3* (CR3), also called *page directory base register*, holds the physical address of the page directory. The page directory contains $2^{10} = 1024$ *page directory entries* (PDE), and the high-order 10 bits of linear addresses are used to index into it. Each page directory entry holds the upper 20 bits of a page tables' address (the lower 12 bits are assumed to be zero, thus aligning each page table on 4096-byte boundaries), along with Page tables contain 1024 entries as well, and are

indexed into by bits 12 to 21 of the linear address, leaving bits 0 to 11 to index into individual pages and page frames (see figure 2.8). Each page table entry describes the properties of a 4096 byte page, so that a single page table can be used to address four megabytes of memory.

Address translation would require many memory references to the page directory and page tables to locate page table entries on each memory access. To remedy this, the CPU caches page table entries in a fast associative memory called *translation lookaside buffer* (TLB). Upon each memory reference, the TLB is first searched for a matching PTE for the requested page. Only when that search fails (called a *TLB miss*) are page directories and page tables walked to locate the matching page table entry, cache it in the TLB and perform the actual memory access.

The following flags in page directory entries and page table entries affect address translation (see also figure 2.8):

- First and foremost, the *present* (P) flag (bit 0 of a PDE or PTE) determines whether the desired page table or page is currently in physical memory or not. If it is not, address translation fails and all other flags are ignored and the operating system is free to use bits 1 to 31 of the PDE or PTE for any purpose.

- The *read/write* (R/W) flag specifies whether the page (for a PTE) or group of pages (for a PDE) can be written to (when the flag is set), or not (when the flag is clear).

- The *user/supervisor* (U/S) flag controls whether access to the page or group of pages is allowed from user mode (flag is set) or kernel mode only (flag is clear).

- The *accessed* (A) flag is set by the CPU whenever a page or page table is accessed. It can be used by an operating system to keep track of which pages are heavily used. The *dirty* (D) flag is set by the CPU when a page is written to. It is used by operating systems to keep track of which pages can be safely discarded to make room for a new page to be brought into memory, and which pages need to be written back to disk. These flags are never automatically cleared by the CPU.

- Instead of pointing to a page table, a PDE can also point to a single four-megabyte page, when its *page size* (PS) flag is set. This is the same size that can be covered by a page table. Operating systems can place frequently used operating system code and data in four megabyte sized pages. This reduces the number of page table entries that need to be cached in the TLB, and thus the number of TLB misses, for better system performance.

2.4.2.2 Page Fault Handling

Address translation can fail for a number of reasons. The simplest case is when the requested page is not currently in physical memory. Other reasons may be writing to a read-only page, or trying to access a supervisor page when the CPU is in user mode.

When address translation fails, the CPU generates a *page fault exception* (or just page fault), which needs to be processed by the operating system:

1. The CPU locates descriptor 14 in the *interrupt descriptor table* (IDT) which is pointed to by the *interrupt descriptor table register* (IDTR). This descriptor normally contains the address of a low-level operating system routine and the privilege level at which it is to be executed.

2. If the privilege level is lower than the privilege level the CPU is operating at when the fault occurs, the CPU switches to a kernel-mode stack segment and stack pointer and pushes the old stack segment and stack pointer onto the new stack.

3. The CPU saves the current CS, EIP and EFLAGS registers on the stack.

4. The CPU saves an error code on the stack. This error code contains several flags that detail the circumstances of the page fault, e.g. whether the it was caused by an attempt to access a not-present page, whether the attempted access was a read or a write operation, or whether the CPU was operating in user or supervisor mode.

5. The CPU saves the linear address that caused the page fault in *control register 2* (CR2).

6. The CPU transfers control to the operating system routine mentioned above. This routine typically saves some more state on the stack and invokes a high-level operating system routine, the *page fault handler*.

7. The page fault handler processes the page fault. It evaluates the conditions that led to the page fault and decides what course of action to take. If the page fault was caused by attempted access to a non-present page that is paged out to disk, it schedules that page to be brought into physical memory. This might involve evicting a page from physical memory and writing it back to disk if it was modified to make room for the missing page.

   If the page fault was caused by access to a non-present page that has no backing store, or by an access conflicting with a page's protection settings, the page fault handler normally sends signals to offending processes, or just terminates it.

8. If a missing page was successfully brought into memory, the operating system can update the faulting process's page tables, schedule it for execution, eventually restoring its state and returning control to it.

2.4.2.3 Paging in Windows XP

In its default configuration, Windows XP splits the 4 GB virtual address space available on the x86 architecture evenly between user-mode processes and the kernel, reserving the lower half of the virtual address space (addresses from `0` to `0x7fffffff`, dubbed user

space) for applications and the upper half (from `0x80000000` to `0xffffffff`, dubbed kernel space) for the operating system. While the user-mode portion of the virtual address space differs from process to process, most of the kernel-mode portion is mapped into all processes' virtual address space.

If a system has more than 256 MB of physical memory, Windows XP uses large (4 MB) pages for some parts of the kernel space so that fewer TLB lookups (and thus, fewer TLB misses) are needed when accessing frequently used kernel memory.

More information can be found in [RS05] and [Sch01].

2.5 Containment

As mentioned in section 2.2, malware analysis should be performed in isolated environments, to contain a malware specimens' malicious effects and prevent it from doing any damage to computer systems that are in production use. Today, the most common way to provide such isolation is by analysing malware within virtual machines as these offer some advantages over real machines. Virtual machines can usually be backed up, restored, or cloned, so that researchers can quickly set up a pristine analysis environment, which can be disposed of after performing analysis on hostile code.

When referring to virtual machines, the system that the virtual machine implementation runs on is commonly called the *host system*, the virtual system is commonly called the *guest system*. Virtual machines' popularity for analysis of hostile code makes it interesting for such code to try and attack virtual machine implementations. Such attacks can generally fall into one of the following categories:

- *Detection* – Hostile code can attempt to detect the presence of a virtual machine and, upon detection, modify its behaviour to hamper analysis. That alternate behaviour could be refusing to run at all, or exercising an innocuous code path that does not reveal malicious functionality contained within the program. There are detection methods for most virtual machines, as their focus is usually on providing an environment that runs most of the software designed for the mimicked platform as fast as possible, and not on mimicking a certain platform as perfectly as possible.

- *Denial of Service* (DoS) – DoS attacks are an attempt to make a resource unavailable to legitimate users. In the context of virtual machines, this can be achieved by exploiting bugs in the implementation that either cause the virtual machine to terminate, or cause it to consume so many resources that its performance degrades so much that it becomes useless.

- *Escape* – It could be possible for hostile code to exploit bugs in a virtual machine implementation that would allow it to execute code outside of the virtual machine. This is the most dangerous kind of attack, as it not only prevents successful analysis of hostile code, but also lets that code escape containment, potentially allowing it to compromise the system hosting the virtual machine, or other systems within the same network.

The following section will give a brief overview about virtual machine implementations and highlight some methods to detect these. A more comprehensive list of known attacks against virtual machines has been documented in [Fer07].

2.5.1 Types of Virtual Machines

Ferrie [Fer07] differentiates three main classes for virtual machines for the x86 architecture, *pure software*, *reduced privilege guest* and *hardware-assisted* virtual machines. Pure software virtual machines, often also called *emulators*, emulate every instruction that is executed on the guest system in software. Reduced privilege guest virtual machines execute (most) guest code directly on the host CPU, albeit at lower privilege levels than usual. Hardware-assisted virtual machines require CPU-facilities that allow running guest code at the same privilege level as it would on a real system while still allowing the host to retain control of the real machine.

2.5.1.1 Pure Software

An advantage of pure software virtual machines is that they are independent of the host architecture, so that they can be ported to different platforms to allow for cross-platform virtualization. However, they tend to be slower than other virtual machine implementations, mainly due to the fact that each instruction executed on the guest CPU needs to be emulated using several machine instructions on the host CPU.

Two free and open-source pure software virtual machines are Bochs [Boc07] and QEMU [Bel05, Bel08], which both have been ported to a variety of host platforms, e.g., Microsoft Windows running on the x86 platform, or Linux and BSD systems running on x86,x86-64, PowerPC or other architectures.

Bochs is able to emulate a complete PC system, including a CPU that can be chosen from a variety of CPUs supporting the x86 and x86-64 instruction set architectures, and additional hardware commonly found in PC systems, such as disks, graphics-, sound-, and network devices. While Bochs can execute most current x86 operating systems as a guest, it offers rather poor performance, sometimes to the point where the guest system cannot be used interactively. Bochs offers an instrumentation interface and a built-in debugger to aid developers in debugging system code.

QEMU also strives to provide full system emulation, although it not only emulates CPUs and additional hardware commonly found in x86-based PC systems, but also other hardware platforms, such as ones using ARM, SPARC, PowerPC or MIPS CPUs. QEMU takes a different approach to CPU emulation than Bochs. It uses dynamic translation to transform entire basic blocks of guest CPU code into equivalent blocks (called *Translation Blocks*) of machine code for the host CPU that can be executed efficiently. Translation blocks are stored in a cache to avoid retranslation of frequently executed basic blocks. This mechanism is optimized for emulation performance and results in QEMU offering much better performance than Bochs.

When emulating x86 and x86-64 systems on host systems of the same architecture

and running Windows, Linux or FreeBSD, QEMU can also act as a reduced privilege guest virtual machine with the help of a kernel module.

2.5.1.2 Reduced Privilege Guest

Reduced privilege guest virtual machines execute most guest code directly on the host CPU. This is in general not a problem for guest code being executed at ring 3 (see section 2.4.1.2), which is executed at the same privilege level on the host CPU. To prevent the host operating system from manipulation by privileged guest code, code that normally runs at privilege level 0, it is executed at a lower privilege level, e.g., 1 (unused by most x86 operating systems). There are however some deficiencies in the x86 architecture that were analysed in detail by Robin and Irvine in [RI00], that pose some problems to implementing a reduced-privileged guest virtual machine for that architecture.

Despite the problems involved, reduced privilege guest virtualization has been very successful in recent years, probably because it offers superior performance over pure-software virtual machines. Some implementations of reduced privilege guest virtual machines are various products offered by VMware, Microsoft VirtualPC, Parallels, VirtualBox, Xen and some versions of QEMU.

2.5.1.3 Hardware-Assisted

Hardware-assisted virtualization was recently introduced into the x86 architecture by by AMD and Intel to remedy the problems faced when virtualizing the it. This new feature adds a new, more privileged ring below the existing 4, as well as new, virtualization-specific instructions to control the guest systems. Using this method, guests can run at their natural privilege levels while allowing the more privileged virtual machine implementation to stay in control.

2.5.1.4 Virtual Machine Detection

Because virtual machines are widely used for analysis of malicious code, such code sometimes tries to detect that it is run within a virtual machine and, upon detection, refuses to run or exercises different code paths to prevent analysis

An obvious method for detecting the presence of a virtual machine stems from the fact that they typically only emulate a very limited set of hardware devices, so that malicious code can query the operating system for the devices that are present and compare their types against a list of hardware devices known to be typically emulated by certain virtual machine implementations.

There are also other methods for detecting the presence of virtual machines. While pure software VMs could in theory implement perfect emulation of a CPU and other hardware devices, this is not the case in practice. The complexity of current CPUs, undocumented bugs and features, and the sheer number of different CPUs makes an identical implementation in software unlikely. Current implementations of QEMU and

Bochs contain bugs in the implementations of several instructions that can be used to identify their presence.

Reduced privilege guest virtual machines suffer from problems inherent in the x86 architecture (described in detail in [RI00]). More specifically, the x86 architecture contains privileged (or sensitive) instructions that reveal information about the hardware state, that can not be properly virtualized, i.e., they can be executed while the CPU is not operating at the highest privilege level without causing an exception, meaning that the virtual machine cannot easily intercept them. The most well-known such instructions are `SIDT` and `SGDT` [Rut04, Kle03, Int06a] that can be used to store the location of the interrupt descriptor table and the global descriptor table table to a memory location. This is sensitive information because those tables cannot be shared between the host and the guest operating system, so that reduced privilege guest virtual machine implementations have to relocate them for the guest system. This causes the aforementioned instructions to return the addresses of the relocated tables which typically differ significantly from locations in non-virtualized environments and can thus be used to detect virtual machines.

Most difficult to detect are hardware-assisted virtual machines. They offer an environment to guest code that most closely resembles the non-virtualized case. Detecting hardware-assisted virtual machines typically involves timing attacks on the execution of instructions known or strongly suspected to be implemented in software by the virtual machine. These techniques become even harder in the absence of (reliable) external timing sources.

Note that some virtual machine implementations even provide an API that allows guest code to communicate with the virtual machine implementation, or include virtual-machine specific drivers to install within the guest operating system to improve performance and whose presence can easily be detected. These kinds of virtual machines assume non-hostile guest code and value flexibility and performance over absolute security.

2.6 Related Work

There is currently a lot of research being done on malware analysis techniques in general, as well as specifically on automated malware unpacking. While many academic researchers have published papers detailing some of their work on unpacking, there is only very little source code publicly available.

2.6.1 Renovo

Renovo [KPY07] is built on top of TEMU, a dynamic analysis component of the BitBlaze binary analysis platform that was developed at the University of California, Berkeley Within the scope of Renovo, instructions that perform memory writes, as well as control transfer instructions are instrumented. Renovo maintains a page-table like data structure (called “shadow memory”) to store flags marking individual bytes within an observed

process's address space as clean (unmodified) or dirty (modified). Renovo inserts a kernel module into the guest operating system which registers some callbacks that notify it about process-related events, like process creation or termination, or insertion of new modules into a process's virtual address space. This information is reported back to the analysis engine. No details are given about the communication method.

Whenever a new process is created, Renovo records the process's name and the value of the CR3 (page directory base) register. The CR3 register's value uniquely identifies individual processes on current versions of the Microsoft Windows operating systems. This property is used by Renovo to enable instrumentation only when the current value of the CR3 register matches that recorded for a process that is to be monitored.

During execution of a monitored process, Renovo instruments memory writes and notes their target addresses by marking them as dirty in the shadow memory. Whenever a new module is inserted into a monitored process's virtual address space, the corresponding memory range is marked as clean.

Renovo detects execution of modified memory on the basic-block level. When the execution flow enters a new basic block, its start address is noted. When leaving that basic block Renovo checks whether any memory within the basic block is marked dirty and in that case records the start of the basic block as the original entry point and dumps memory pages containing modified memory. To unpack multiple packing layers, Renovo then marks all memory as clean and continues execution. Renovo stops execution after a configurable time-out.

The authors highlight two methods that malware could use to evade unpacking by Renovo, virtual machine detection, and exploiting the time-out by staying inactive for long enough. Renovo can work around some of the methods for virtual-machine detection by instrumenting problematic instructions like `SIDT` (see section 2.5.1.4). The authors plan to implement heuristics based on the number of instructions executed to detect malware trying to exploit the time-out by sleeping or busy looping.

The authors evaluated Renovo's performance by using it to unpack a sample executable packed with 14 different packers and comparing the original sample executable's .text section to the memory pages dumped by Renovo. The default configuration was used for 13 packers, while two samples were generated using the 14th packer (Themida [Ore07]), one with instruction virtualization (see section 2.3.2.1) and one without. Renovo was able to successfully unpack 12 of the 15 total samples. One of the samples that could not be unpacked was packed with Themida and had instruction virtualization enabled. However, Renovo was able to unpack some parts of the original executable. Some hidden layers could be extracted from the other two samples that could not be unpacked, however, the original code could not be restored. The authors suspect that this is due to the executables not being compatible with Renovo's emulation engine. The authors also tested Renovo on a number of malware samples that the signature-based packer identifier PEiD [pei07] determined to be packed. Most of the samples used were classified as bots by Norton Anti-Virus. Renovo was able to identify most of the malware samples as packed, confirming PEiD's results. Due to the lack of the original unpacked executables, a different method was chosen to verify the correctness of the extracted code and data. As most of the chosen samples were identified as bots, they were searched for IRC com-

mands commonly used by bots. The majority of extracted code and data were found to contain these.

The authors compared Renovo's unpacking results to results by PolyUnpack and the Universal Unpacker Plugin that comes with the commercial disassembler IDA Pro [Dat05, Dat]. For both experiments, PolyUnpack and UUnP returned only about half as many results as Renovo. Neither Renovo nor TEMU are currently publicly available.

2.6.2 Saffron

Saffron [QV07] is a generic automated malware unpacker that comes in two incarnations. One employs dynamic instrumentation based on Pin [Pin], a dynamic instrumentation tool that 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. If execution branches to previously written memory it marks the branch target as a candidate original entry point and dumps the modified memory range to a file. Saffron was able to successfully unpack 4 out of a total of 5 packers that were tested. The packer for which unpacking failed uses a checksum to verify the integrity of the processes' address space before transferring control to the unpacking code. This highlights one of the drawbacks of automated unpacking using Pin, namely that it is easily detectable and that it modifies the instrumented processes' address space. The authors also note other disadvantages of this method, e.g., that it is fairly slow (however, no numbers are given), and that standard anti-debugger (see section 2.2.3.1) techniques cause problems with Pin as well. Source code of the dynamic instrumentation method is available at `http://www.offensivecomputing.net/bhusa2007/saffron-di.cpp`.

The other method presented in [QV07] 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 by usermode code.

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 analysed, e.g., by using a real machine as a sacrificial lamb (see section 2.2.3). 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. The authors plan to release source code for the page fault handler debugging method eventually. To date it is not yet available.

2.6.3 PolyUnpack

In [RHD+06] the authors highlight the need for methods that *automatically* unpack packed malware due to the sheer volume at which new packed malware appear in the wild. They then present their method *PolyUnpack* that strives to automatically unpack Microsoft Windows PE executables.

PolyUnpack works based on the assumption that packed malware hides malicious code from static analysis, to prevent detection of its malicious intent. Thus, any code being executed that is not part of a static model of a malware instance must have been generated at runtime. PolyUnpack starts by building a static model of a malware instance by disassembling the malware instance and partitioning it into a set I of instruction sequences, and a set D of non-instruction data sequences. PolyUnpack then executes the malware instance one instruction at a time in a sterile and isolated environment, after each instruction comparing the current instruction sequence to the set of instruction sequences in the static model. If the current instruction sequence is not a subsequence of one of the members of I, it must be unpacked code being executed. Execution is then halted and a text representation, a binary dump, or an executable version of the unpacked code is stored to disk. PolyUnpack makes use of the Microsoft Windows Debugging API for single-step executing malware instance and is thus be susceptible to evasion techniques targeting that API (see section 2.2.3.1). PolyUnpack attempts to unpack malware that are packed more than once by first removing the outermost compression layer and creating a new executable from the result, repeating the process until no further compression layers are detected. The authors note that PolyUnpack is fairly slow, e.g., it takes about 17 minutes on average to extract a single packed executable. An implementation of PolyUnpack as a plugin for the Olly Debugger [Yus] is available at `http://polyunpack.cc.gt.atl.ga.us/polyunpack.zip`. Source code has not been made available.

2.6.4 Malware Normalization

The method for *malware normalization* proposed in [CKJ+05] attempts – among other normalizations – to unpack packed malware. The presented method makes two assumptions about the nature of the packed code to be normalized. Firstly, it assumes that no generated (unpacked) code overwrites existing code, or in other words, that code-generation (unpacking) completes before control is transferred to the generated code. Secondly, it assumes that code generation is independent of inputs or the runtime environment. This is generally a valid assumption to make as malware is normally self-contained and designed to automatically run and unpack itself on a variety of different victim systems. An example of malware that violates this assumption is malware that prior to unpacking attempts to detect virtual machines or other malware analysis mechanisms, and, upon detection, stops the unpacking process.

The authors state that they modified QEMU [Bel05, Bel08] to trace memory writes and monitor code execution of packed programs within a Windows 2000 guest system. As soon as previously modified memory is the target of a control-transfer instruction, that target address is noted and execution is halted and an unpacked program is generated from the captured data and the contents of the virtual memory range covering the original in-memory image of the packed program. The unpacked program's entry point is set to the target address of the control transfer to modified memory. This process is then iteratively applied to the new program to remove all additional packing layers.

No implementation details about the modifications that were made to QEMU are given and the software has not been made available in binary or source code form.

Despite stating earlier that multiple unpacking layers are removed by regenerating a new executable and restarting the unpacker on it [CKJ+05, p. 8], the authors later indicate that the presented method does not produce runnable executables [CKJ+05, p. 11] and that the presented method does not attempt to recover import information, as the focus is on aiding (signature-based) malware detection for which import reconstruction is not deemed important. The authors also note that the OEP of packed executables is not always correctly determined, proving some difficulty to malware-detectors employing entry-point based signatures.

3 Implementation

With the threat of malware increasing in numbers, variety and sophistication, researchers need to be able to efficiently analyse malware to find its weaknesses and to devise ways to effectively combat it. Unfortunately, the majority of malware today is runtime-packed, requiring researchers either to rely on mostly dynamic analysis methods, or to unpack malware specimens manually before performing low-level static analysis on them. The presented solution, dubbed *Pandora's Bochs*, strives to reduce some of the complexity by using dynamic instrumentation to automatically extract the hidden code from runtime-packed malware so that it can be subjected to subsequent static analysis.

This chapter will describe the implementation of Pandora's Bochs. It will first list the design goals that drove development of Pandora's Bochs, detail some of the assumptions on the behaviour of runtime packers it is expected to deal with and finally describe the overall structure and implementation.

3.1 Design Goals

Prior to starting work on the implementation of Pandora's Bochs, the following design where formulated to drive its development:

Unobtrusiveness An automated unpacker should be unobtrusive, i.e., the environment it presents to a program that is to be unpacked should be as close to a real environment as possible, to make it hard for the specimen to detect that is being analysed. More specifically, it should not make use of debugging facilities offered by analysis operating system to prevent detection by common anti-debugging techniques as described in section 2.2.3.1 and it should run as little additional code on the analysis platform as possible.

Isolation It is important that malware cannot interfere with the analysis platform. While unobtrusiveness should prevent the malware from detecting the analysis environment, isolation should prevent the specimen from tampering with it. More specifically, the unpacker should not rely on debugging facilities provided by the analysis OS so that it is unaffected by a specimen using them for its own purposes.

Genericity A first implementation of an automated unpacker should be as generic as possible, i.e., it should avoid integrating packer-specific knowledge that might lead to loss of genericity. While this might prevent successful unpacking of programs packed by some packers, it is important to create a generic foundation that can be experimented

with and upon which more specific unpackers can be build easily by integrating packer-specific algorithms.

Portability Analysing malware on a platform other than the platform it targets is assumed to lessen the risk of damaging the analysis environment. Furthermore, every researcher has his or her own preference of operating system and hardware platform. An automated unpacker should therefore work on a variety of different platforms.

Freedom While a lot of research is currently being done on malware analysis and automated unpacking, it seems as if very little of that research and the tools involved are being made publicly available. The author believes that collaboration of academic, corporate and independent researchers is important to successfully fight malware. The automated unpacker that is to be developed should therefore be free for anyone to use, modify and distribute. More specifically, it should make use only of software that is available under a free software license and it should be made available to anyone under such a license.

3.2 Scope

The following assumptions were made about the runtime packers that Pandora's Bochs should unpack:

- *Unpacking does not cross process boundaries* – Unpacking stubs could spawn another process, unpack their code to that new process's virtual address space and then redirect that process's control flow to the unpacked code. Another possibility would be unpacking the original code to an image file on disk and executing it.

  Although it is possible to capture these kinds of behaviours behaviour by monitoring the respective operating system services, an implementation would be fairly complex and exceed the time available for developing a working prototype. Additionally, the majority of runtime packers appear not to use this technique.

- *Unpacking does not happen in kernel mode* – It is possible for an unpacker to come with its own kernel-mode driver that manipulates the address space of a process to be unpacked. Again, it might be possible to distinguish this behaviour from standard operating system services writing to the processes' address space, but it was decided not to implement such measures due to time constraints and the majority of runtime-packers not implementing such behaviour.

- *The unpacked code stays within the image boundaries* – Code can generally be unpacked to arbitrary memory locations, e.g., into part of the executable image, the heap, or the stack. However, executables typically expect to be loaded to a known location in memory, as specified by the image base field in the PE headers, which is normally not a problem as executable files are the first thing that are

mapped into a new processes' (empty) address space. Unpacking stubs must take care to either honour this requirement for the unpacked code, or relocate it by manually adjusting memory references within the unpacked code.

The simplest way to accomplish this is for the packed executable to ask to be mapped so that it contains the memory range expected by the original code and unpacking to that range, which should normally succeed. Another method would be to ask the operating system to allocate memory for the process to use at the unpacked code's original memory range, but while there is a good chance that the memory can be allocated, this is not guaranteed. The majority of packers experimented with do indeed unpack the original code to within the image. There was one packer that doesn't (Morphine), but which manipulates the image list so that the unpacked code lays within an "image" of the same name as the packed executable.

As such, this assumption seems valid for the majority of runtime-packers. In light of some runtime-packers also unpacking helper code to the heap and executing it, it has the additional advantage of limiting locations for the potential OEP, reducing some of the complexity involved in correctly detecting it.

- *The OEP is reached by a branch instruction* – While it is theoretically possible for the unpacking stub to unpack to an adjacent memory range, essentially extending a last basic block within the unpacking stub, then transferring control to that basic block, and letting execution proceed to executing the unpacked code, this is unlikely. The entry points of most PE executables are not at the start of all executable code, but usually embedded into other code. Therefore the described scenario is unlikely to happen in practice and it is valid to assume that control is transferred to the original entry point by means of a branch instruction.

- *No obfuscation of the original code* – While the unpacker is expected to extract packed code regardless of code obfuscation techniques that may have been used on it, it cannot be expected to automatically and generically reconstruct the original, unobfuscated code.

3.3 The Unpacker

The unpacker part of Pandora's Bochs is, as the name suggests, based on Bochs [Boc07], a portable x86 emulator. It was chosen as a basis for an automated unpacker for several reasons:

Firstly, it is a *pure software* virtual machine, meaning that it is not subject to some of the design-inherent flaws of reduced-privilege guest virtual machines for which well-known and simple detection methods exists, e.h., *Red Pill* [Rut04] or *scoopy doo* [Kle03]. While there exist some methods to detect the presence of Bochs as well, these 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. More on this subject can be found in section 2.5 and [Fer07]. Additionally, Bochs is *free software* and licensed under the GNU Lesser General Public License (LGPL), version 2.1 [Fre99]. Its source code is available on the Internet for download and anybody is allowed to use and modify it, and even to distributed derivative works, on the condition of adhering to the LGPL. This makes it an ideal candidate for a project designed with freedom in mind, as described in section 3.1. Furthermore, Bochs provides a built-in mechanism for instrumenting code running on the emulated CPU. This is a unique feature that sets it apart from the other well-known and free x86 emulator, QEMU, and makes it highly useful for dynamically analysing malicious software.

However, there is a disadvantage to using Bochs as well. For one it is quite slow when compared to reduced privilege guest virtual machines or QEMU in pure software mode. While these are generally able to run Windows XP guest systems with acceptable performance on current hardware, Bochs is subjectively too slow to use a Windows XP guest system interactively. This was a bit of an inconvenience when installing the analysis system, and later turned out to have some trouble dealing with unpackers of high algorithmic complexity.

Most of the unpacker's functionality, as well as the PE reconstructor were written in the Python [pyt07] programming language. Python is a powerful interpreted high-level programming language supporting the functional, object oriented and imperative programming paradigms. Using Python was considered after first experiments implemented in C++ yielded rather cumbersome code where a lot of effort went into correctly creating and maintaining complex data structures, a task that Python handles automatically most of the time. Python thus allows for writing more concise code that is easier to read and maintain. Python comes with an extensive standard library [Ros06b] and powerful built-in types, e.g., strings, lists, sets and dictionaries. Its type system provides extensive introspection and reflection capabilities which make it an ideal choice for reading and manipulating complex data structures like those used within the Windows kernel or in PE images. Python can easily be extended by creating *extension modules* in C or C++, and it can also be embedded into C or C++ applications by linking in the Python interpreter and writing some wrapper code to interact with it [Ros06a, Ros06c]. The version of Python used in Pandora's Bochs (Python 2.5) is licensed under the Python 2.5 license [Pyt06] which grants anyone the right to use Python and to create and distribute derivative works.

Python was chosen for Pandora's Bochs because of its ease of use, its powerful standard library, its flexible type system, it's embedding and extension capabilities, and its permissive license. Furthermore, changes to the Python code to not require recompilation of Bochs to be effective, thus saving some development time.

SQLite [sql07a] is a library implementing an SQL database engine that can easily be embedded into C, C++ or Python applications. It does not use a database server and thus requires no configuration or setup. Its authors claim it to be faster than popular client/server database engines such as MySQL and PostgreSQL. SQLite has

been released to the public domain [sql07b], meaning among other things that anybody is free to use and distribute SQLite and to create derivative works and distribute them.

SQLite is used in Pandora's Bochs to store instrumentation events and memory dumps for later analysis. It was chosen because of its availability of libraries for Python and C, its ease of use and its permissive license.

3.3.1 Preparation

In preparation for experimentation and building the unpacker, a guest system had to be created. Windows XP SP2 was chosen as the guest operating system due to it being the major target for malware today.

First, a *sparse* disk image was created for use with Bochs using Bochs's `bximage` utility:

```
bximage -hd -mode=sparse -size=20000 winxpsp2sparse.img.0
```

Advantages of these kinds of disk images are that uninitialized parts of the disk image don't use up disk space on the host system, and that Bochs supports *stacking* them, i.e., that it is possible to use one disk image for a while, and then create another disk image as a layer on top of it, and writes will always only affect the topmost layer.

For installing the guest operating system, Bochs was built to offer maximum performance during the interactive installation process, particularly without the built-in debugger or instrumentation support, using the following invocation of the build configuration script:

```
./configure --enable-clgd54xx --enable-vbe --enable-pci --enable-acpi \
            --enable-pnic --enable-ne2000 --enable-x86-debugger \
            --enable-disasm --enable-readline --with-x11 --with-nogui \
            --enable-all-optimizations
```

Bochs was then started using the configuration file from appendix A and Windows XP SP2 was installed. After installation, another disk image was created as described earlier, named "winxpsp2sparse.img.1" and the configuration file was changed to use that file as a disk file, resulting in Bochs using the earlier image as a basis and writing changes only to the new image. Then, the guest operating system was updated over the Internet using Microsoft's update service.

Then, another disk image was created and stacked on top of first two images, the guest operating was booted, and a snapshot of the machine status was created once system bootup was complete. The topmost disk image layer and the saved machine status were then backed up to be used later as a pristine state from which to start the unpacking process. Finally, Bochs was built using the following arguments to the build-configuration script:

```
./configure --enable-clgd54xx --enable-vbe --enable-pci --enable-acpi \
            --enable-pnic --enable-ne2000 --enable-x86-debugger \
            --enable-disasm --enable-readline --with-x11 --with-nogui \
```

```
--enable-guest2host-tlb --enable-icache \
--enable-fast-function-calls --disable-repeat-speedups \
--enable-instrumentation=instrument/unpacking
```

The major differences between this and the build configuration used earlier are that instrumentation support was added, and that the "repeat speedups" optimization was disabled (and all other optimizations enabled manually), as it breaks proper instrumentation of memory writes performed by repeated string operations using the `REP` prefix [Int06a, p. 4-212], which are typically used to copy large amounts of data in memory, e.g., by the `strcpy()` and `memcpy()` C functions.

3.3.2 Instrumentation Interface

Bochs provides an interface that allows for fine-grained instrumentation of guest systems via a variety of callback functions. Some of the events that can be instrumented are execution of branch instructions, CPU exceptions or memory accesses.

3.3.2.1 Convenience Functions

Some functions were needed for easier access to the emulator core's state. Most of these functions are modified versions of functionality already contained in Bochs. Some of the modifications that were needed were the removal of instrumentation callbacks, as these functions are to be called from within instrumentation callback functions and should thus not generate additional instrumentation events, removal of side-effects on Bochs's internal state, and adding the possibility of choosing a particular virtual address space by supplying the address of a page directory.

`bool logical2linear( Bit16u sel, Bit32u ofs, Bit32u *laddr)`
`bool logical2linear( Bit16u sel, Bit32u ofs, Bit32u *laddr, Bit32u pdb)`
These functions translate a logical address consisting of a segment selector and an offset into a linear address (see section 2.4.1). The additional argument `pdb` accepted by the second function allows specifying a page directory base, so that the virtual address space of a particular process can be selected. The first function just calls the second, passing the first three arguments unmodified, and using the current value of the emulated CPU's CR3 register as the page directory base.

`bool linear2physical( Bit32u laddr, Bit32u phy)`
`bool linear2physical( Bit32u laddr, Bit32u phy, Bit32u pdb)`
These two functions translate a linear to a physical address. Again, the second function accepts a page directory base address in the `pdb` argument to select a specific virtual address space, and the first function performs address translation by invoking the second one and using the emulated CPU's current current page directory base.

`bool pmem_read(Bit32u addr, Bit32u len, Bit8u *buf)`
This function reads as many bytes as specified by `len` from *physical* address `addr` into

the memory buffer pointed to by `buf`.

`Bit32u vmem_read(Bit32u addr, Bit32u len, Bit8u *buf)`
`Bit32u vmem_read(Bit32u addr, Bit32u len, Bit8u *buf, Bit32u pdb)`
These two functions can be used to read `len` bytes at *linear* address `addr` into a `buf`. As with the other functions described before, the second function accepts a page directory base address in the `pdb` argument to select a particular virtual address space described by the page directory at that address. The first function calls the second one with on the first three arguments unmodified, and passing the CPU's current page directory base address as the `pdb` argument.

`void page_fault( Bit32u laddr)`
This function can be used to simulate a page fault caused by a read access to a non-present page, from code running at a privilege level of 3. To that end it loads the linear address passed as the `laddr` argument into the emulated CPU's CR2 register, resets some of the CPU's internal state and calls the `interrupt` method of Bochs's CPU object to prepare transferring control to the guest system's exception handler. Due to the way exception handling works in Bochs, the `page_fault()` function then jumps back into Bochs's main emulator loop by means of a long jump. Because this means that the `page_fault()` function does not return, extra care needs to be taken when calling this function. Pandora's Bochs uses this function to coerce a Windows XP guest system's demand-paging mechanism into bringing pages that are of interest to the analysis environment into memory. It should only be called when the CPU is currently operating at privilege level 3 to avoid interrupting the guest operating system kernel while performing critical tasks in kernel mode.

3.3.2.2 Instrumentation Callbacks

Bochs's instrumentation interface is implemented as a series of C++ preprocessor macros which can be defined to call instrumentation callback functions whenever an event of interest occurs. Instrumentation can be enabled at build configuration time by passing the switch `--enable-instrumentation=<directory>` to the Bochs's build-configuration script. This switch causes the script to set the C++ compiler's include path to the specified directory, makes the build infrastructure build a library containing definitions of instrumentation callback functions in that directory and defines the `BX_INSTRUMENTATION` macro in Bochs's `config.h` to 1. The `config.h` header file is included by the `bochs.h` header which is included by most of Bochs's source files. The `bochs.h` header in turn includes the `instrument.h` header from the instrumentation directory. The `instrument.h` header uses the `BX_INSTRUMENTATION` macro to decide at compile time whether to define the instrumentation macros to be empty or to wrap instrumentation callbacks.

The following instrumentation macros are used within Pandora's Bochs:

`BX_INSTR_INIT( cpu)`

This macro is called whenever Bochs initializes a CPU object, i.e., usually when the emulator is started. In Pandora's Bochs it's defined to call the `bx_instr_init()` function which initializes the Python interpreter and the Python instrumentation code, and creates and initialises an SQLite database file.

`BX_INSTR_ATEXIT()`
This macro was added to Bochs to be called when Bochs is about to exit. It call the `bx_instr_atexit()` function that calls the Python instrumentation code's `shutdown()` function and then cleanly shuts down the embedded Python interpreter.

`BX_INSTR_TLB_CNTRL( cpu, what, new_cr3)`
This macro is called by Bochs whenever an operation is performed that flushes the TLB, or when the `INVPLG` instruction is executed to invalidate a TLB entry. The type of event is passed in the `what` argument, a possibly new value for the `CR3` register is passed in the `new_cr3` argument. The only event of importance to Pandora's Bochs is an explicit modification of the `CR3` register which signifies that the Windows XP guest operating system has just switched from the virtual address space of one process to the virtual address space of another. This macro wraps the `bx_instr_tlb_cntrl()` function which checks whether the event is an explicit modification of the `CR3` register and in that case passes control to the `ev_mod_cr3()` function which in turn invokes a python function with the same name and sets a global variable `bx_instr_enabled` to the result. That global variable is used to toggle fine-grained instrumentation on only for those processes that are to be monitored, for better performance.

`BX_INSTR_CNEAR_BRANCH_TAKEN( cpu, new_eip)`
`BX_INSTR_CNEAR_BRANCH_NOT_TAKEN( cpu)`
`BX_INSTR_UCNEAR_BRANCH( cpu, what, new_eip)`
`BX_INSTR_FAR_BRANCH( cpu, what, new_cs, new_eip)`
These macros are called by Bochs whenever fine-grained instrumentation is enabled for a process (i.e., when the value of `bx_instr_enabled` is not zero) and the emulated CPU executes a branch instruction of some kind. These macros wrap the `ev_branch()` function that passes the branch source, branch target and branch type to a Python function of the same name and logs the branch to the SQLite database if the Python function returns a non-zero result.

`BX_INSTR_LIN_ACCESS( cpu_id, lin, phy, len, rw)`
This macro is called whenever fine-grained instrumentation is enabled for a process, and an instruction accesses memory using a linear address. It is defined to call the `ev_write()` function whenever the `rw` argument indicates a memory write. It passes the linear address and the length (1, 2 or 4 bytes) of the write to a Python function of the same name and logs the write to the database if that function returns a non-zero result.

`BX_INSTR_BEFORE_EXECUTION( cpu_id, i)`
This macro is called by Bochs whenever fine-grained instrumentation is enabled for the

current process and the emulator is about to execute a new instruction. It wraps the `fetch_pending_page()` function. That function checks a global flag named `bx_instr_pending_page` which can be set from Python code to indicate that at least one page needs to be paged in. It then asserts that the CPU is currently operating at privilege level 3, and invokes the `pending_page()` Python function that returns a pages' linear address and invokes the `page_fault()` function to simulate a page fault at that address to make the guest operating system's demand-paging mechanism load the page into memory.

3.3.2.3 Python Interface

The Python interface needs to fulfil two tasks. It needs to make the emulator's internal state accessible to instrumentation code written in Python, and it must provide mechanisms to call Python instrumentation callbacks from Bochs, which was written in C++. The first task is what the author of the Python programming language calls *extending*, the second is what he calls *embedding* Python. An introduction into both subjects is given in [Ros06a].

Python provides an extensive C API that programmers can use to interface with the Python interpreter, which is documented in great detail in [Ros06c]. Python objects are presented to C code as variables of a type `PyObject`, which encapsulates information such as the Python object's type, its value, or its reference count. The type can be any of Python's built-in types or a user-defined type, and it determines what kinds of values the object can hold. The reference count of an object is used by Python's garbage collector which destroys an object when its reference count reaches zero. When interfacing with Python, care must be taken to correctly maintain the reference counts of objects to avoid accidental destruction of objects that are still in use, and to prevent objects that are not in use anymore from unnecessarily using memory.

Calling Python Functions From C The Python/C API provides several functions for calling a Python function from C code. Of those, Pandora's Bochs uses

```
PyObject* PyObject_CallFunction( PyObject *callable, char *format, ...)
```

which expects a reference to a callable Python object in the `callable` argument, a format string describing the kinds of arguments passed to the Python function in the `format` argument, followed by arbitrary many arguments that are converted to Python objects as specified by the format string. The `PyObject_CallFunction()` method returns `NULL` if an exception was raised by the Python function, or a pointer to a `PyObject` containing the Python function's return value. References to Python functions can be obtained by using the

```
PyObject* PyObject_GetAttrString( PyObject *o, const char *attr_name)
```

method that returns a pointer to a `PyObject` if an attribute of the `PyObject` referenced by `o` with a name specified in `attr_name` was found. The object referred to by `o` is typically a Python module that was imported using the

```
PyObject* PyImport_ImportModule( const char *name)
```

function with the module's name passed as the `name` argument. Pandora's Bochs expects instrumentation code to reside in a Python module called `PyBochs`, tries to import that module upon initialisation and tries to obtain references to the following functions:

`init( samplename)`
This function is called upon emulator initialisation. It accepts a filename as an argument, which indicates the name of the sample executable to unpack.

`shutdown()`
This function is called when the emulator is about to shut down. It can perform some last-minute actions if needed. In Pandora's Bochs's current form, it does nothing.

`ev_mod_cr3( new_cr3)`
This function is called whenever the `CR3` register of the emulated CPU changes, i.e., whenever the guest operating system switches from the virtual address space of one process to the virtual address space of another. The `ev_mod_cr3()` function is expected to return `True` if fine-grained instrumentation should be enabled for the newly active process, otherwise it should return `False`.

`ev_branch( source, target, type = None)`
This function is called whenever fine-grained instrumentation has been enabled for the current process, and the emulated CPU executes a branch instruction. The function accepts the branch source, the branch target, and a branch type (e.g., a function call, function return, or a conditional or unconditional branch) as arguments. It is expected to return `True` if the branch is to be logged to the SQLite database, otherwise it should return `False`

`ev_write( addr, size)`
This function is called whenever fine-grained instrumentation has been enabled for the current process, and the emulated CPU executes a memory write. It accepts the linear address of the memory write, and its size(i.e., either 1, 2, or 4 bytes) as arguments. It is expected to return `True` if the write should be logged to the SQLite database, or `False` otherwise.

`pending_page()`
This function is called whenever the emulated CPU is about to execute an instruction at privilege level 3, fine-grained instrumentation is turned on for the current process, and the Python instrumentation code has signalled that it would like to have a page brought into virtual memory. It is expected to return a linear address within the page to be brought in.

Additionally a reference to the `PageFaultException` class is obtained, which is used to signal to the Python instrumentation code that some virtual memory could not be read.

Calling C Functions From Python To be able to call C function from Python code, a Python module must be created, the C functions must be registered with that module, and the module in turn needs to be registered with the Python interpreter. The functions that are to be made available need to all follow the same C prototype:

```
static PyObject* func( PyObject* self, PyObject* args)
```

The `self` argument is only relevant when the C function implements a method belonging to some Python class or object, to pass a reference to the object on which the method is invoked. It is present in regular functions as well so that the Python interpreter won't have to deal with two different kinds of C functions. The `self` argument is not used in Pandora's Bochs.The `args` argument references a `PyObject` that represents a Python tuple containing all arguments that were passed from Python to the function. The

```
int PyArg_ParseTuple( PyObject *args, const char *format, ...)
```

Python/C API function can be used to convert those argument tuples to C data types as specified by the format string `format`. `PyArg_ParseTuple()` expects pointers to variables of the correct types to be passed after `format` and returns `true` on success. On failure it returns `false` and raises an appropriate exception.

Pandora's Bochs creates a module named `PyBochsC` and registers it with the embedded Python interpreter. That allows the Python instrumentation code running within the interpreter to import the module by that name and access the functions it exports. These function are mostly used to make the emulated machine's state accessible to Python instrumentation code:

```
logical2linear( selector, offset, pdb)
vmem_read( address, length, pdb)
pmem_read( address, length)
```

These function wrap the corresponding convenience functions described earlier in this chapter. The `vmem_read()` function can be used to read the contents of a virtual address space identified by the page directory base passed in the `pdb` argument. The `pmem_read()` function can be used to read the contents of physical memory. Logical addresses consisting of a segment selector and an offset can be converted to linear addresses with the `logical2linear()` function.

```
registers()
eip()
```

`genreg( register)`, `sreg( register)`, `creg( register)`, `dreg( register)`
These functions allow Python instrumentation code to retrieve information about the

emulated CPU's registers from Bochs. The `registers()` function returns a Python dictionary containing all the values of all general purpose registers, the segment registers, the flags register, the instruction pointer and all control registers, indexed by their name in lower case. The `eip()` function returns the instruction pointer's current value, the `genreg()`, `sreg()`, `creg()` and `dreg()` functions can be used to query the values of a general purpose, segment, control or debug register respectively.

`set_eip( new_eip)`
`set_genreg( register, new_value)`
These functions can be used to set the values the instruction pointer or a general purpose register.

`emulator_time()`
Bochs internally maintains a timer that counts the number of "clock ticks" since the emulator was booted. This timer is used as a frame of reference for instrumentation events. Its value can be queried using this function.

`shutdown()`
This function can be used by Python instrumentation code to indicate to the emulator core that it should shut down.

`pending_page( pending)`
This function can be used by the instrumentation code to notify the emulator core whether there are currently any pages pending to be brought into the linear address space. When the flag is set, the emulator queries the Python instrumentation code for a new page to be brought in (by simulating a page fault at a linear address within that page) whenever it starts execution of an instruction in privilege level 3. This was made to depend on aforementioned flag for performance reasons, to avoid frequent calls from the emulator into Python instrumentation code when there are no pages pending to be brought in.

`sqlite3_insert_process( pdb, pid, ppid, filename`
`sqlite3_insert_image( pdb, base, size, basedllname, fulldllname)`
`sqlite3_insert_image_dump( time, pdb, base, type, score, tag, dump)`
These functions can be used by instrumentation code to log information about processes, memory images of PE files, and dumps of memory ranges to the SQLite database. The actual logging is performed from C++ code, as simultaneous access to the same database from both C++ and Python code lead to locking conflicts and logging of frequent events such as branches and writes is performed from C++ code because the C/C++ SQLite library offers superior performance over the Python SQLite module.

3.3.2.4 Logging

Pandora's Bochs collects a lot of information during execution of a malware sample, such as all memory writes and all branches. This information is useful in several ways. For one it is used during the reconstruction phase 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. A relational database was chosen to store this information in an easily accessible and searchable way.

3.3.3 Accessing Structured Data in Virtual Memory

As one of the design goals of Pandora's Bochs is unobtrusiveness, it was decided to not run any code apart from a standard operating system and a program to be analysed within the guest system. While other automated unpackers (see section 2.6) rely on guest operating system services to gather information about the operating system state, processes, or PE files, Pandora's Bochs needs to gather this information from the emulator's point of view, solely by reading from virtual memory and inspecting the emulated CPU's state. As most of this information is stored in data structures managed by the operating system kernel, the instrumentation code must be able to locate that data, and to correctly parse it.

3.3.3.1 Locating Kernel Structures

To unpack a binary it is executed within Pandora's Bochs which in turn monitors its execution. To be able to do that, Pandora's Bochs, having only a very low-level view of the emulated machine and the code being executed, must be able to identify the currently running process and its properties. As Windows XP provides each process with its own virtual address space, a process can be uniquely identified by the page directory describing that virtual address space, which in turn is uniquely identified by its location in memory. The page directory base register `CR3` holds the start address of the page directory of the currently active virtual address space, so that once a mapping from page directories to processes is created, the currently operating process can be identified by the current value of the `CR3` register.

In the Windows NT family of operating systems, e.g., on Windows NT, Windows 2000, or Windows XP, processes are represented by *executive process blocks* (`EPROCESS` blocks) [RS05]. An `EPROCESS` block is a data structure that resides in kernel space and which contains a lot of the information the Windows kernel needs for managing the associated process, as well as containing pointers to other equally important data structures. While there appears to be no official documentation about how to obtain a reference to the `EPROCESS` block of the currently running process, this information can be pieced together from a variety of sources, such as third-party books on the undocumented internals of Windows operating systems [Sch01], books [HB05] and articles [bs05] on rootkits, as well as community websites on rootkits (`http://rootkit.com`) or

reverse engineering (`http://www.openrce.org`). That information was verified using WinDbg from Microsoft's Debugging Tools for Windows [Mic07] and by experimenting with Bochs's built-in debugger.

Windows uses the x86 architecture's segmentation mechanism to provide a convenient way to access certain important data structures, i.e., by placing them at the beginning of the segment (normally) selected by the `FS` register. In user mode, the `FS` register normally contains segment selector `0x0038` referencing a segment that contains the *thread environment block* (TEB), in kernel mode it contains segment selector `0x0030` referencing a segment that contains the *kernel's processor control region* (KPCR). The latter contains important data about the current thread as well as pointers to other data structures which eventually lead to the `EPROCESS` structure of the currently active process as depicted in figure 3.1. Some noteworthy fields within the `EPROCESS` structure are the

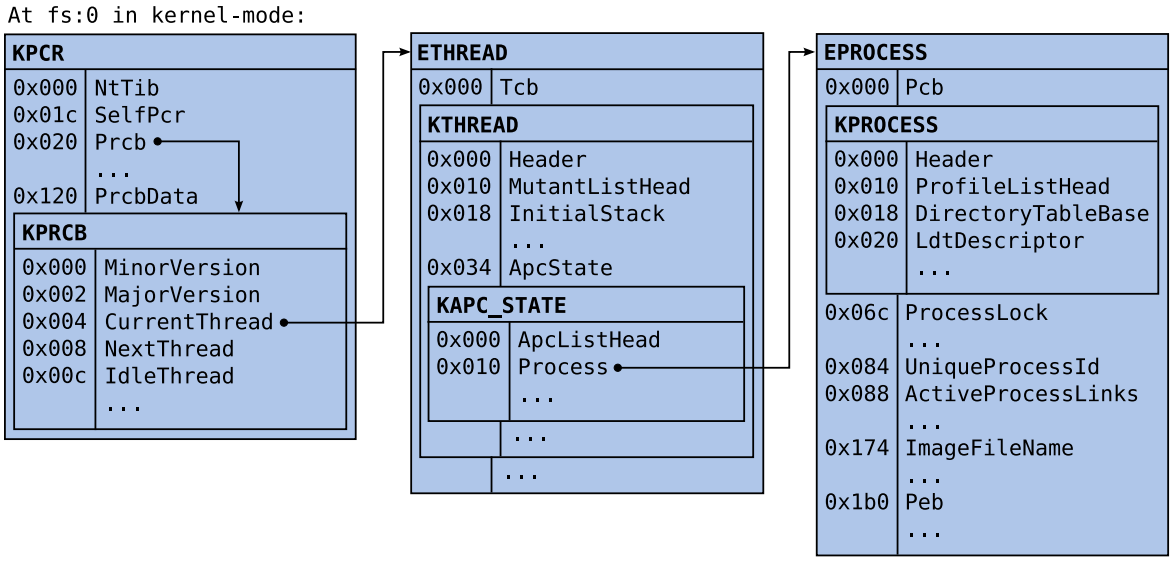


Figure 3.1 – Finding the EPROCESS structure

`UniqueProcessId` field which contains a unique identifier the operating system associates with that process, the `ImageFileName` field, which contains the first 16 bytes of the process's executable file name and is used by Pandora's Bochs to identify the process to be instrumented, and the `Peb` field which contains a pointer to the *process environment block* (PEB), a data structure residing in user space that contains and references additional information about the process. Among the information accessible through the PEB is a list of which modules (typically the executable file and several dynamic link libraries) have been loaded into the process's virtual address space. It is stored in the *module list* which is a doubly linked list of `LDR_DATA_TABLE_ENTRY` structures that contain a module's base address in the `DllBase` field, its size in the `SizeOfImage` field, and its full path and filename in the `FullDllName` and `BaseDllName` fields (see figure 3.2). The instrumentation code uses this information to determine which DLLs

are mapped into the instrumented process, so that it can extract more information from them, such as which symbols they export.

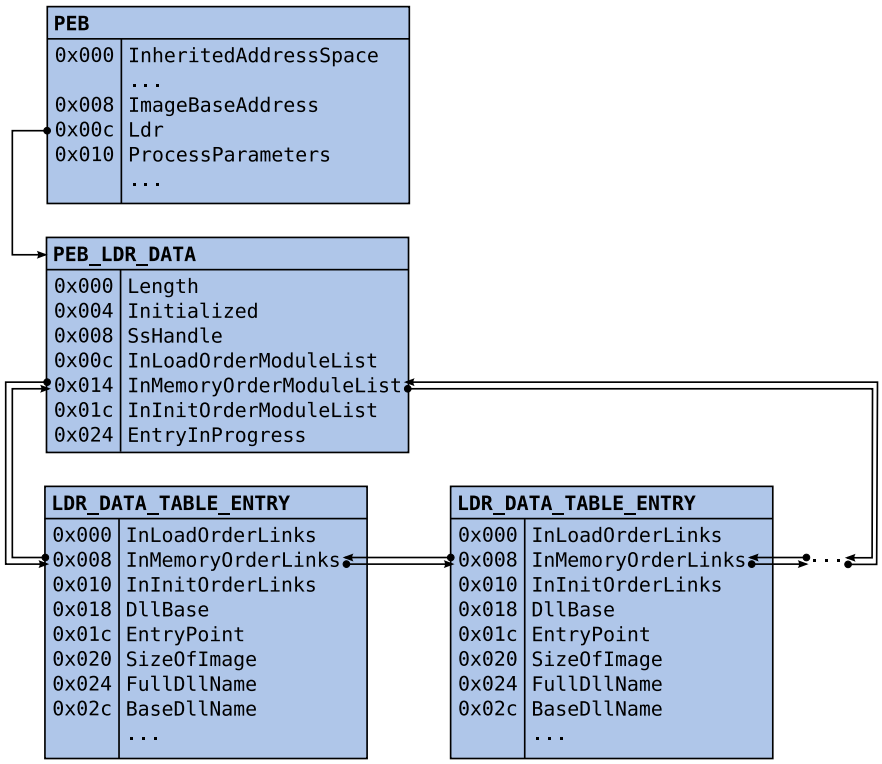


Figure 3.2 – Data structures describing PE images in a process's address space

3.3.3.2 Parsing Data Structures

Most of the data structures referred to in the previous section are not well documented and some also differ between different versions of Windows. It was therefore important to find a way to obtain correct information for the version of Windows used as a guest system in Pandora's Bochs (Windows XP SP2). Fortunately, Microsoft provides WinDbg as part of the Debugging Tools for Windows [Mic07]. WinDbg is a debugger that can access debugging symbols provided by Microsoft to dump type information about many data structures that are not very well documented otherwise. Example 3.1 shows WinDbg being used to dump type information for different structures. It shows names and types of structure members as well as offsets of those members within their parent structures.

To utilize this information from instrumentation code written in Python, several Python classes were created that provide easy access to complex data structures residing in a byte string, or the emulated virtual memory. The core functionality is provided by the `StructuredData` class which implements some simple mechanisms to operate on structured data that is made accessible through an instance of a class implementing the

Example 3.1 Listing type information in WinDbg

```
lkd> dt _KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY
   +0x010 Process          : Ptr32 _KPROCESS
   +0x014 KernelApcInProgress : UChar
   +0x015 KernelApcPending : UChar
   +0x016 UserApcPending   : UChar
lkd> dt _LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
lkd>
```

`Backend` interface (see figure 3.3). The `VMemBackend` class is used for read-only access to data within the emulated virtual memory (see example 3.2). Note that the base and limit attributes are set to cover the whole 32-bit addressable memory space by default.

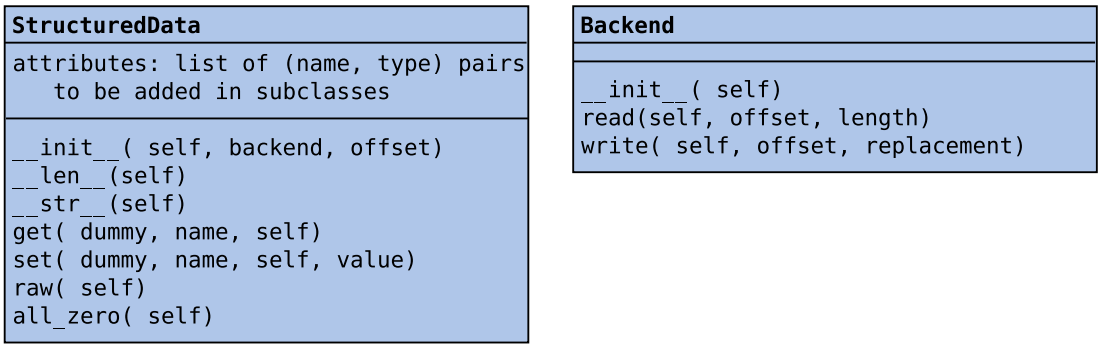


Figure 3.3 – StructuredData and Backend class interfaces

The `StructuredData` class is meant to be subclassed by other classes that should have an attribute `attributes` which should be a list of `(name, type)` tuples like in example 3.3. That list describes the names and types of member attributes of the structure the subclass will wrap.

Such subclasses inherit `StructuredData`'s `__init__()` method which is called when they are instantiated. It walks the attribute list, calculates the indicated type's size, and cumulates those sizes to calculate an attribute's offset within the structure. Using this information it creates a dictionary mapping attribute names to `(type, offset, size)` tuples and adds appropriate attributes to the subclass:

- If `type` is a string, it is assumed to be a format string for Python's *struct* module ([Ros06b, chapter 4.3]) that can be used to convert data between C-style structures that are stored in raw byte strings and Python.

  An attribute of the specified name is then created as a Python *property* [Ros02] with getter and setter functions obtained by using the *functools.partial()* method

Example 3.2 The VMemBackend class

```
class VMemBackend( Backend):
    def __init__( self, base = 0, limit = 0x100000000, pdb = None):
        self.base = base
        self.limit = limit
        self.pdb = pdb

    def read( self, offset, length):
        return PyBochsC.vmem_read( self.base + offset, length, self.pdb)

    def write( self, offset, replacement):
        raise Exception( "write() not implemented in Virtual Memory Backend")

    def __len__( self):
        return self.limit - self.base
```

Example 3.3 Subclassing StructuredData

```
class KAPC_STATE( StructuredData):
    attributes = [ ("ApcListHead", "4I"),
                   ("Process", P(EPROCESS)),
                   ("KernelApcInProgress", "B"),
                   ("KernelApcPending", "B"),
                   ("UserApcPending", "B"),
                   ("Pad", "B")
                 ]
```

([Ros06b, chapter 6.6]) to bind the attribute's name to the `name` argument of `StructuredData`'s `get()` and `set()` methods.

These methods essentially use that name to look up the format string, offset and size of the attribute of the specified name within the dictionary created earlier and then use that information for accessing the actual data. The `get()` method reads `size` bytes at the specified `offset` from the structure's backend (relative to the offset of the current structure within the backend), and returns the result of converting that data to a Python data type as specified by `format`, by using `struct.unpack()`. The `set()` method first converts the `value` argument to a byte string using `struct.pack()` and the specified `format`, and then invokes `write()` on the structure's backend to overwrite data at the appropriate offset. Note that the `VMemBackend` class currently does not implement a `write` method.

- Otherwise, `type` is assumed to be a class with an `__init__()` method that accepts a backend and an offset as arguments, e.g., a different subclass of `StructuredData`. It is instantiated with the current subclass's backend, and the sum of the current

subclass's offset within the backend and the argument's offset within the structure as arguments. This is useful for modelling nested types.

Example 3.4 The get() and set() methods

```
def get( dummy, name, self):
    (format, struct_offset, size) = self.attribute_dict[ name]
    offset = self.offset + struct_offset
    retval = struct.unpack( "<" + format, self.backend.read( offset, size))
    if len( retval) == 1:
        return retval[ 0]
    else:
        return retval

def set( dummy, name, self, value):
    (format, struct_offset, size) = self.attribute_dict[ name]
    offset = self.offset + struct_offset
    replacement = struct.pack( "<" + format, value)
    self.backend.write( offset, replacement)
```

To be able to model C-style pointers as well, a class `Pointer` was created that provides a `deref()` method to access the data pointed to and whose `__init__()` method implements the same interface as `StructuredData`, so that it can be used within attribute lists to create nested types with `Pointer` attributes. The `P( datatype, ignore_nulli = 0)` function (which is also used in example 3.3) allows one to dynamically create a typed pointer class. It uses Python's `type()` function to create a new subclass of `Pointer` with an additional class attribute `datatype` holding the type of the data pointed to. This is a very convenient way to pass type information to the `Pointer` class while maintaining its `__init__()` method's generic interface.

In a similar way, a `LIST_ENTRY` class was created to mimic the data type used for doubly linked lists within Windows's internal data structures. As shown in example 3.1, it contains two member attributes, `Flink` and `Blink` that each link to another instance of `LIST_ENTRY`. As depicted in figure 3.2, `LIST_ENTRY` instances contained in structures link back and forth only among themselves, so, to locate the containing data structure in memory, the offset of the `LIST_ENTRY` member within the data structure must be known. The `LIST( datatype, le_member)` function is very similar to the `P()` function in that it can be used to dynamically generate subclasses of `LIST_ENTRY` and at the same time attach much needed information to it, without disturbing its `__init__()` method's simple interface, so that it, too, can be used to created nested `StructuredData` types. In this case, that information is information about the data type the list entries are contained in, and the name of the list entry's member attribute, so that the containing data structures can be properly located.

Using this collection of classes, type information obtained with WinDbg can be easily converted to attribute lists for creating subclasses of `StructuredData` to map data structures residing in virtual memory to Python objects. These can then be used in standard and intuitive ways, just like other Python objects. For example, in an instance `state` of the `KAPC_STATE` class seen above, the attribute `ApcListHead` could be accessed as `state.ApcListHead`, the structure pointed to by the Process attribute by accessing it as `state.Process.deref()`.

These classes are used in Pandora's Bochs for accessing process-management related data structures in kernel memory and for parsing PE files in virtual memory.

3.3.4 Modelling Guest Operating System State

Pandora's Bochs uses several data structures to keep track of the guest operating system's state, e.g., which processes are running within the guest operating system and what their properties are, which image files are mapped into a process's address space, and what memory pages within a processes' virtual address space have been modified.

3.3.4.1 Processes

An instance of the `Process` class keeps information about a single Windows process, such as instances of the `StructuredData` class to map the process's `KPCR` and `EPROCESS` structures, a dictionary that maps base addresses to PE images that have been mapped into the process's address space, or a list of pages that should be brought into the process's address space.

It is initialised with the value of the page directory base register that describes the process's virtual address space. Upon initialisation it locates the `KPCR` structure at the beginning of the segment described by segment descriptor `0x30` as seen in figure 3.1. It then goes on to initialise some member attributes and invokes the class's `update()` method which performs the following tasks:

- Due to some uncertainty about the current state of the guest operating system, especially during initialisation of a new process, care must be taken to ensure that the data that instrumentation code operates on is valid. To that end, instances of the `Process` structure contain a `valid` attribute that is initialised to `False`.

  When a process is being updated and is not yet valid, the `EPROCESS` structure is located in memory by following member attributes of the `KPCR` structure obtained during initialisation as depicted in figure 3.1. Some member attributes of that structure are then checked for values that indicate that the structure (and thus the accompanying process) has been fully initialised by the guest operating system:

  – The process creation time (`CreateTime`), the number of active threads (`ActiveThreads`), the process id (`UniqueProcessId`) and the parent process id (`InheritedFromUniqueProcessId`) should not be equal to zero.

  - The pointer to the process environment block (`Peb`) should point to a memory area where this data structure is typically located, i.e., at addresses between `0x7ff00000` and `0x80000000`.
  - Alternatively, the `EPROCESS` structure could refer to either the *System* or the *Idle* process for which weaker criteria suffice to mark them as valid. The system process's `ImageFileName` is the ASCII string `System`, its process id is 4, and its parent process id is 0. The idle process has the same process ids and its image filename is `Idle`.

  If a process is newly valid, its information is logged to the database.

- If the process is valid, the dictionary of PE images is updated:
  - The module list is obtained by locating the module list as depicted in figure 3.2.
  - That list is then traversed and if an image is already known, i.e. if an instance of the `Image` class is already stored in the image dictionary, its `update()` method is invoked. Otherwise, a new instance of the `Image` class is created and stored in the dictionary.

  For more information see the next section (3.3.4.2) that details the `Image` class.

- It is determined whether the process should be monitored. This is currently just a comparison whether the `ImageFileName` within the `EPROCESS` structure matches the filename supplied during initialisation.

A process's `update()` method is also called once every time the guest operating system switches to it when the process is either being monitored, or when it could not yet be marked valid. These further updates are only performed when a process first enters user space after the process switch, assuming that the kernel-mode data structures that are accessed during an update are then in a stable state.

3.3.4.2 Executable Images

The `Image` class represents the properties of PE image files that have been loaded into a process's virtual address space. It reflects basic information about an image, such as its base address, its size, its name and full path, as well as more specific information, such as a PE image's exports or imports. Upon initialisation an image is passed an instance of an `LDR_DATA_TABLE_ENTRY` structure (see figure 3.2) as well as a reference to the process containing it. The core of the initialisation process is the `update()` method which is also invoked when a process updates it's image dictionary. It performs the following tasks:

- Just like for the `Process` class, the `LDR_DATA_TABLE_ENTRY` structure might be accessed by instrumentation code at a time when it is not yet fully initialised. For that reason, images are checked for validity similarly to processes. If an image is not valid, the `update()` method checks the following conditions and marks the image as valid if they are met:

  - The image base (`DllBase`) and the image size (`SizeOfImage`) must be a multiple of the page size. The image size must not be zero.
  - The image's entry point (`EntryPoint`) must point to a location within the bounds specified by the image base and the image size.
  - The memory range covered by the image must be in user space, i.e. at addresses lower than `0x80000000`.
  - The image's full path is pointed to by the `FullDllName` field. That pointer must point to a page that is currently in memory, most notably, it should not be zero.

- If the image is newly valid, it is logged to the database and it is scheduled for an initial memory dump. Any pages within the image that are not yet mapped into memory are scheduled to be brought in.

- If the image is valid and the process containing it is being monitored, an attempt is made to parse the PE data in virtual memory. If it contains any export data, the exported symbols' names are looked up in a global list of watchpoint definitions and watchpoints added accordingly.

3.3.4.3 Other Structures

ModifiedPage Processes keep a dictionary that maps page numbers to instances of the `ModifiedPage` class. These carry information such as the last time a page was modified, executed, or dumped, and which memory locations within the page were modified since the last dump. This information is stored in per-page data structures, because maintaining such information for individual memory addresses would require a lot more overhead both in storage space and in algorithmic complexity.

Watchpoints Pandora's Bochs supports *watchpoints*, i.e., it allows for associating locations in virtual memory with objects that can invoke some action when control flow reaches that location. Pandora's Bochs uses watchpoints to take note of calls to the `GetProcAddress()` Windows API function that resolves symbols from imported DLLs. Whenever that function is called, the watchpoint obtains the function arguments from the stack and adds another watchpoint that is triggered when the call to `GetProcAddress()` returns. It then records the returned virtual address of the symbol that was resolved and creates another watchpoint that triggers on the first call to the resolved symbol. This is used by Pandora's Bochs to track "liveness" of monitored processes, as described in section 3.3.6.

Helper The `Helper` class is instantiated upon emulator initialisation. It is used to keep track of liveness properties and to maintain a dictionary that maps page directory base addresses to processes.

3.3.5 Unpacking

The unpacking process utilizes Boch's save/restore mechanism that can save and restore the emulator's CPU, memory, and device state, and Bochs's ability to work with stackable disk images where writes only go to the topmost image (see section 3.3.1).

To start a new process within the emulator, Pandora's Bochs relies on the guest system's *AutoRun* functionality that automatically executes commands from a file named `autorun.inf` on CD-ROMs that are inserted into a CD-ROM drive. For that, an ISO9660 CD image file needs to be created, e.g., by using the `mkisofs` tool from the Cdrtools [Sch07] software package. That image should contain the program to be unpacked and a file named `autorun.inf` with contents similar to the following example that will make Windows try to automatically start a program named "sample.exe" when the CD is "inserted" into the emulated CD-ROM drive:

```
[autorun]
open=sample.exe
```

Then, the disk image layer and machine state saved earlier (see section 3.3.1) need to be restored, the saved state's configuration file (following the format of appendix A) has to be pointed to the ISO image, and options related to the unpacker have to be set by adding a line similar to the following:

```
unpacker: executable="sample.exe", duration=3600
```

This line basically tells the unpacker to monitor execution of all processes created by executing an image file named "sample.exe" with an upper bound to the execution time of 3600 seconds. The `duration` field can also be set to 0 for unlimited execution time. Now, Bochs can be invoked by using the command

```
bochs -r saved_state
```

assuming `saved_state` is the directory that contains the saved machine state. This will make Bochs resume from that state, the guest operating system will notice that a CD has been inserted into the emulated CD-ROM drive and it will start executing commands from the `autorun.inf` file.

After invoking Bochs like this, instrumentation code monitors the guest operating system and the processes running within it. Upon switching virtual address spaces, i.e., whenever the `CR3` register is modified, it is checked whether this virtual address space (and thus: the process to which this address space belongs) has already been seen earlier. If that is the case, the associated structure that models the process's properties is fetched from a global data dictionary that maps page directory base addresses to `Process` structures. Otherwise a new `Process` structure is created and inserted into the dictionary. Then, the process's `enter()` method is called that does some bookkeeping, tracks liveness of currently monitored processes, and tells the emulator core to shutdown should certain liveness criteria not be met (see section 3.3.6). Then fine-grained

instrumentation is enabled either if the new process is being monitored, or if it could not yet be marked valid. When fine-grained instrumentation is enabled, Pandora's Bochs instruments branch instructions and memory writes.

After switching to a new process, when executing the first branch instruction to a userspace address, the process structure is updated to check whether the process data is valid, whether the process should be monitored, and to update the process's image list (see section 3.3.4.1).

3.3.5.1 Memory Writes

Each process structure contains a dictionary that maps page numbers to instances of the `ModifiedPage` class (see section 3.3.4.3). Whenever Pandora's Bochs performs a memory write from usermode code, it checks whether such a structure already exists for the target page, otherwise it creates a new one and inserts it into the dictionary. Then, the addresses written to are entered into the structure's set of modified address, and the time of last modification for the page is updated.

3.3.5.2 Branches

All branches whose target is in usermode are instrumented by Pandora's Bochs. If a watchpoint (see section 3.3.4.3) has been registered for the target address, it's `visit()` method is invoked to perform the action associated with it. Then, if the branch target is a memory location that was previously modified, a dump is made of the memory contents around the branch target as follows:

- The *virtual address descriptor* (VAD) tree [DG07] is traversed to find the VAD of the memory range containing the target address. The VAD tree is a data structure that resides in kernel space and describes all valid memory ranges within a process's virtual address space, e.g., the stack, memory allocated on the heap, or memory occupied by files that are mapped into a process's address space, e.g., executable files and dynamic link libraries. The advantage of using this data structure is that it provides a complete view of the address space and that it cannot be tampered with from usermode code. If a VAD describes a file that is mapped into memory, it also contains information such as the file's name.

- An attempt is made to read the memory range indicated by the VAD. If some pages within that memory range are not currently mapped into memory, these pages are filled with zero-bytes.

- If there is a file associated with the VAD, the file's name is noted. Some packers unpack to the heap (meaning there is no file associated with the VAD), but modify the module list (a userspace structure, see section 3.3.4.2) to associate a module with that memory range. To account for this behaviour, if the VAD does not reference a file, it is checked whether the target address falls within the range of a module, and that module's name is noted.

- The memory dump and related information such as a time stamp, whether the memory range could be fully read, and, in the case of files, the filename, are logged to the database.

- Finally, the modified memory range around the branch target is marked clean, i.e., starting from the page containing the branch target, for the contiguous range of modified pages around it, the sets containing modified memory locations are emptied.

3.3.5.3 OEP Detection

Detecting the OEP generically was one of the toughest challenges faced once the instrumentation facilities were implemented. In principle, every branch to a modified memory location is a potential OEP, in practice however, several scenarios need to be considered:

- After branching to a potential OEP, execution tends to stay in within that modified memory region and further branches within that region are unlikely OEP candidates and should not be considered.

  Other automated unpackers commonly account for this by discarding all information about previously modified memory upon visiting a potential OEP. Renovo [KPY07] performs a memory dump, marks all memory as unmodified and continues execution. PolyUnpack [RHD+06] and the malware normalization method [CKJ+05] deal with multiple unpacking layers in a similar way. Upon executing modified memory, they halt execution, generate a new executable containing the modified memory and restart the unpacking process using that new executable.

  However, experiments showed that this method often fails to detect the OEP correctly, as some unpacking stubs branch to memory that was modified along with memory at the OEP before jumping to the real OEP. Some packers seem to do this explicitly to fool unpackers that try to detect execution of modified memory.

- It is common for unpacking stubs to unpack in multiple stages, e.g., by unpacking a first stage, executing it to unpack another, and so on, until control is transferred to the actual original code. Thus, only the final control transfer should be considered as a potential OEP.

  While this behaviour should not pose any problems for Renovo, PolyUnpack and malware normalization attempt to tackle multiple unpacking layers by reconstructing valid executables from memory dumps and executing these under supervision of the unpacker until no further layers are detected. However, reconstructing executables from memory dumps is an error-prone process, especially when unpacking layers are executed on the heap. It is hard, if not impossible, to generate a working executable that takes the state of the whole virtual address space into account.

- Some executable protectors obfuscate calls to API functions by redirecting them to a branch function which generates branch trampolines to those API functions

on the fly and transfers control to them. Of course, these trampolines constitute modified memory, but they should not be considered as candidate OEPs.

The other presented automated unpackers typically assume every branch to modified memory to be a potential OEP and assume the last suchcandidate to be the correct OEP. As such they do not correctly account for executable protectors that employ such obfuscation methods.

Pandora's Bochs takes a slightly refined approach to OEP detection. When branching to modified memory, not all memory is marked clean, but only the contiguous modified memory range containing the branch target. This improved OEP detection for unpacking stubs that prematurely branch to memory that was modified along with the actual OEP, as these false candidates tend to be well away from the real one, and thus not within the same contiguous modified memory range. Another approach that was experimented with was marking only that memory around a candidate OEP as clean that was modified by the same code. This was achieved by recording the source addresses of memory writes with a granularity of 256 bytes and clearing contiguous modified memory ranges sharing the same writers. However, this approach often failed to correctly identify the whole range containing memory semantically similar to the branch target and yielded more false positives than the simpler method.

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. The details will be explained in greater detail in the next section (3.3.6). The last candidate OEP reached during execution of a sample is considered the most likely OEP by Pandora's Bochs.

As explained in section 3.1, it is valid to assume that the final unpacked code is extracted to a location within the bounds of the packed executable image. This assumption reduces the number of potential locations for the OEP and allows Pandora's Bochs to disregard branches to modified heap memory, which might be caused by dynamic branch trampolines even after the actual unpacking is complete.

To account for incorrectly detected OEPs, the corresponding memory range is dumped for all first branches to newly modified memory regions. Along with the execution history, this enables a researcher to investigate the unpacking process of packers outside of the proposed scope (see section 3.2), should the need arise.

3.3.6 Termination

Determining whether a program will unpack additional code and transfer control to it is undecidable in the general case [RHD+06]. 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:

- *Memory Writes vs. Branch Targets* – Pandora's Bochs records recent branch targets and memory writes with a granularity of 256 bytes. This is not done with byte

granularity to keep data structures and insertion times small. Whenever the guest operating system switches to a monitored process, the ratio of recent writes to recent branch targets is calculated, and, if big enough, this is noted as innovation. The rationale behind this is that unpacking typically happens within a tight loop with few branch targets and many memory writes.

- *Dynamic Link Libraries* – Shared libraries contain new functionality for the process to use. Thus, whenever a new dynamic link library is detected within a monitored process's virtual address space, this is considered innovation.

- *Executing modified memory* – Whenever a process executes instructions from a modified memory region, it is likely that new code was unpacked and is now being executed for the first time. Thus, execution of modified memory is considered innovation.

- *Function calls* – PE packers typically strip most imports from the packed file and have the unpacking stub recover them by means of the `GetProcAddress()` API call. Pandora's Bochs monitors the use of `GetProcAddress()` and creates watchpoints for API calls resolved this way. Whenever such a function is called for the first time, this is considered innovation.

Upon each process switch, Pandora's Bochs increments a counter when the new process is not being monitored. The emulator is shut down when that counter reaches a configurable upper bound. If, however, the new process is being monitored, and has shown innovation when it was last executed, that counter is reset to zero. Unfortunately, there are scenarios where the innovation-based approach might fail to stop the emulator even though the monitored program is fully unpacked. This might happen, when a process frequently copies data from one memory location to another, or when it uses a branch function that dynamically resolves each API function that is called prior to transferring control to it. Likewise, there might be cases of processes that are still in the process of unpacking, yet show no innovation as defined above. This could happen in heavily obfuscated decompression routines that execute many branches for each single written byte, or processes that use other means than `GetProcAddress()` to resolve API functions, e.g., by parsing DLLs and their exports in memory.

A common solution to guarantee termination of the unpacking process is to use an upper bound to the unpacker's total run time. Pandora's Bochs, too, has a configurable upper bound for the total run time after which execution is stopped, regardless of prior results.

3.4 Reconstructing PE files

The memory images dumped by the analysis engine are usually not valid PE files. When loading PE files into memory, the Windows PE loader performs several transformations whose effects need to be undone when attempting to generate a valid PE binary from a

memory image. Sections are copied into memory from their on-disk offsets within the PE file to their RVAs and the difference between their on-disk size and their virtual size is padded with zero-bytes. Sections are aligned differently on disk than they are aligned in memory. Imports are resolved by the PE loader, rewriting import address table entries with the linear addresses of imported symbols.

Packers further complicate matters. The unpacked code is usually written to sections formerly marked as containing uninitialized data, with a raw size of 0. The packed binaries' entry point is usually not equal to the original entry point. Packers typically overwrite the original import information and replace it with minimal imports, often just importing the `LoadLibrary()` and `GetProcAddress()` functions. These are then used at runtime to regenerate the original import address table.

To avoid correct parsing and modification by analysis tools, some packers also interleave the PE header with the EXE header or use headers that do not adhere to the PE specification yet do not prevent the rather robust implementation of Windows's PE loader from "correctly" loading a packed file into memory. Some packers also appear to overwrite (parts of) their headers in memory.

In light of these obstacles, automatically reconstructing valid PE files from memory dumps is not a trivial task. However, as one of the goals of Pandora's Bochs is to present useful input to existing analysis tools, reconstruction should be attempted.

To that end, a Python program was developed that attempts to reconstruct valid PE files from the information that was logged to an SQLite database during unpacking. Its operation is detailed in the following sections,3.4.1 and 3.4.2.

3.4.1 Headers

It is attempted to reconstruct valid headers from the PE headers found within the memory dump. To leave most of the original data found within the memory dump unmodified, the original PE headers and section headers are copied and appended to the memory dump, and only the PE signature offset within the EXE header is modified to refer to the copied headers. This makes the PE loader and analysis tools use the copied headers instead. These can then be manipulated to reconstruct a valid PE file while leaving the original dumped headers as is. The copied header is then zero-padded to a multiple of the section alignment.

After setting the entry point field within the optional header to the detected original entry point, an attempt is made to fix the copied section headers. As the sections within the memory dump are aligned as specified by the section alignment field within the optional header, the file alignment field within that header is changed accordingly, so that the memory dump's structure need not be modified. Then, the following operations are performed on every section header:

- The virtual size field is rounded up to a multiple of the section alignment and the field containing the on-disk size is changed to that value.

- The field containing the section's offset within the file is changed to its RVA.

- The section's characteristics field is changed so that the section is marked as containing initialized data and as being read- and writeable. It is also marked as as containing code and being executable if code was executed within that section during unpacking.

After fixing all the sections that were dumped, another section header is appended to the existing ones to describe a new section named *.pandora* with offsets and sizes such that the section contains the copied and modified headers. This section will later be extended to take up reconstructed import data.

3.4.2 Imports

Import data plays an important role in understanding a program's semantics. Disassemblers such as HT Editor [WB07] or IDA Pro [Dat] typically try to parse it and use it to identify and annotate calls to the Windows API in disassembly listings. Such information about API calls provides insight into the higher-level semantics of a program, as API calls are typically well documented and as such useful for determining a program's behaviour. Additionally, API functions' type signatures can be used to identify the data types of memory locations that are used as arguments to these functions. Furthermore, Windows PE loader requires valid import data to correctly load and execute a program.

When reconstructing PE files from a memory dump, an attempt is made to reconstruct import data based on the execution trace stored in the SQLite database as follows:

- To find all API calls, the database is queried for branches from within the dumped memory image into memory occupied by a shared library.

- Because entries in the import address table are typically used as operands to indirect branches, the instructions at the queried branches' source addresses are disassembled, indirect branches identified and their operands stored in a list.

- That list is then sorted to find the lowest address that was used as an indirect operand to a branch instruction. It is assumed that that address is the location of an import address table entry and memory is then searched downwards to find the likely first import address table entry. For that, all import address table entries are expected to point to a symbol exported by a DLL and that the import address tables for all DLLs all reside within the same contiguous memory region. It is also assumed that import address tables are only separated by all-zero double words (as specified by [Mic06]), or values of `0x7fffffff` or `0xffffffff` (values that were found during experimentation). The search terminates when a double word is found which does not point to an symbol exported by a DLL, and which is neither of these values. Then, the location of the previous double word is taken as the start address of the first IAT in the memory dump.

Then, information about assumed import address tables is collected as follows:

- The memory dump is searched upwards for the first double word that is not the name of a symbol exported by a DLL, while saving for every symbol its name and the name of the DLL containing it in a list. The first double word that does not point to a symbol is set to zero to terminate the IAT.

- If not all of the IAT entries point to symbols from the same DLL, the most likely cause is that the DLL holding the majority of symbols forwards the missing ones to another DLL. Thus, the majority DLL is searched for forwarders for the missing symbols, and, if those are found, the corresponding entries in the symbol list are modified to contain the forwarded name and the name of the majority DLL. Should there still be symbols from minority DLLs left, an attempt is made to replace these symbols by symbols with the same name from the majority DLL as a last resort.

- This procedure is repeated for a new assumed IAT at the location of the next double word until no further IATs are found.

The information collected in the previous step is then used to reconstruct the import directory table, the hint/name table and the import address tables:

- For every IAT that was found earlier, space is reserved at the end of the reconstructed image for an import directory entry, followed by space for one all-zero terminating import directory entry.

- For every IAT, a zero-terminated ASCII string representing the name of its majority DLL is appended to the reconstructed image, and the name RVA field of the corresponding import directory entry is set to point to the beginning of that name. Then, a hint/name table is constructed for that IAT, containing all-zero hints and symbol names from the information collected earlier. The IAT entries are then set to the RVAs of these hint/name table entries, and the import directory entry is modified to point to the IAT.

- The section header describing the *.pandora* section that was created earlier is modified to include the data that was appended. Tthe image size is modified in the optional header and the import table data directory is made to reference the newly generated import data.

4 Evaluation and Results

Pandora's Bochs's performance was evaluated in two different ways. Firstly, known executables were packed using a variety of different packers (using their default options) and an attempt was made to uncompress them using Pandora's Bochs. The unpacking results were then compared to the original binary. Secondly, an attempt was made to unpack a collection of 409 malware samples from the RWTH Aachen honeynet and the unpacking results were examined to gauge their quality.

4.1 Synthetic Samples

A test set of packed binaries was generated from two different executables. The first 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. The other executable was a standalone version of *GNU Wget* [wge], a command-line program to download files from the Internet. The version used was 749568 bytes in size. The following runtime packers were used to create samples for the evaluation:

4.1.1 Packers

- ACProtect 2.0, a commercial executable protector. An evaluation version is currently available at `http://www.ultraprotect.com/`
- ANDpakk2 0.18, a freeware runtime packer aimed at compressing demos, currently available at `http://and.intercon.ru/`
- Armadillo 5.20, a commercial executable protector. An evaluation version is currently available at `http://www.siliconrealms.com/`
- 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/`
- ASProtect SKE 2.3, 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/`

- FSG 2.0, a freeware runtime packer, currently available at `http://www.xtreeme.prv.pl/`
- kkrunchy 0.23a, a freeware runtime packer aimed at compressing demos, currently available at `http://www.farbrausch.de/~fg/kkrunchy/`
- MEW 11 SE 1.2, a freeware runtime packer, currently available at `http://northfox.uw.hu/`
- Morphine 2.7, a free and open source executable encryptor, currently available through the Internet Archive Wayback Machine at `http://web.archive.org/web/20051227054026/www.hxdef.org/download.php`
- Molebox Pro 2.2951, a commercial executable protector. An evaluation version is currently available at `http://www.molebox.com/`
- NeoLite 2.0, a (discontinued) commercial runtime packer. An evaluation version is currently available from some software archives, e.g., `http://download.chip.eu/de/NeoLite-2.0_46860.html`
- nPack 1.1.300 beta, a freeware runtime packer, currently available at `http://petools.org.ru/npack.shtml`
- NSPack 2.3, a (discontinued) commercial runtime packer. An evaluation version is currently available through the Internet Archive Wayback Machine at `http://web.archive.org/web/20070206171102/http://www.nsdsn.com/english/nspack.zip`
- Obsidium 1.3, a commercial executable protector. An evaluation version is currently available at `http://www.obsidium.de`
- Packman 1.0, a free and open source runtime packer, currently available at `http://packman.cjb.net/`
- PECompact2 2.79, a commercial runtime packer. An evaluation version is currently available at `http://www.bitsum.com/`
- PELock NT 2.04, a freeware executable protector, currently available on some software archives, e.g. `ftp://ftp.sac.sk/pub/sac/security/pelck204.zip`
- PE-PaCK 1.0, a freeware runtime packer, currently available on some software archives, e.g., `http://ftp.elf.stuba.sk/packages/pub/pc/pack/pepack10.zip`
- PESpin 1.304, a freeware executable protector, currently available at `http://pespin.w.interia.pl`
- Petite 2.3, a freeware executable compressor, currently available at `http://www.un4seen.com/petite/`

- RLPack 1.19 Basic, a free and open source executable protector`http://rlpack.jezgra.net/`
- tELock 0.98, a freeware runtime packer, currently available on some software archives, e.g., at `http://www.softpedia.com/progDownload/Telock-Download-23.html`
- Themida 1.9.3.0, a commercial executable protector. An evaluation version is currently available at `http://www.oreans.com/`
- UPX 3.01w, a free and open source, cross-platform runtime packer, currently available at `http://upx.sourceforge.net/`
- VGCrypt 0.75, a freeware executable encryptor, currently available on some software archives, e.g., at `ftp://ftp.sac.sk/pub/sac/security/vgcrypt.zip`
- WinUPack 0.39, a freeware runtime packer, currently available at `http://wex.cn/dwing/mycomp.htm`
- Yoda's Crypter 1.3, a free and open source executable encryptor, currently available at `http://yodap.sourceforge.net/`
- Yoda's Protector 1.03.3, a free and open source executable protector, currently available at `http://yodap.sourceforge.net/`

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. The above list includes only those packers that could generate a working binary from at least one of the target executables.

4.1.2 Performance

The following tables list the detailed results from executing each synthetic sample under Pandora's Bochs and attempting reconstruction of an unpacked PE image file. 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 *manual* for packers that required user interaction, *liveness* when Pandora's Bochs terminated because it could not detect any progress as detailed in section 3.3.6, or *timeout* if Pandora's Bochs terminated after the user-configurable timeout.
- The *correct OEP* column shows whether the OEP was correctly determined by Pandora's Bochs.
- As a *similarity* measure, the original executables' .text section was compared to the data that the unpacking result contains at the same location. The number of matching bytes was divided by the number of total bytes within that area to obtain the similarity value. It is listed in the *.text similarity* column.
- The *result executes* column shows whether it was possible to execute the reconstructed image file correctly.
- The *unpacking time* column contains the absolute run time of Pandora's Bochs for a given sample.
- The *reconstruction time* column contains the time needed for reconstructing an executable from the information that was dumped to the database.

Note that non-interactive samples were executed on a computer system that used an Intel Pentium D CPU running at 3.4GHz, and 2GB of RAM. The samples that required user interaction were executed on a different system that sported an AMD Athlon64 CPU running at 1.8GHz, and 3GB of RAM. Thus, the unpacking times for interactive and non-interactive samples cannot be directly compared due different system configurations and the time that was needed for interacting with the sample.

Wget

packer	version	working executable	user interaction	termination reason	correct OEP	.text similarity	result executes	unpacking time	reconstruction time
ASPack	2.12	X	-	liveness	X	1.0	X	8:25	0:15
ACProtect	2.0	X	X	manual	X	1.0	-	<15:00	∼0:25
ANDpakk2	0.18	X	-	timeout	-	0.4145[1]	-	1:00:00	-
Armadillo	5.20	X	-	timeout	-	0.4145[1]	-	1:00:00	-
ASProtect	1.35	X	O[2]	liveness	X	0.9984	-	18:54	0:27
ASProtect SKE	2.3	X	O[2]	liveness	X	0.9985	-	16:56	0:25
eXpressor	1.5.0.1	X	-	liveness	X	1.0	X	27:48	0:24
FSG	2.0	X	-	liveness	X	1.0	X	5:58	0:16
kkrunchy	0.23a	X	-	liveness	X	1.0	X	21:22	0:21
MEW 11 SE	1.2	X	-	liveness	X	1.0	X	26:37	0:23
Molebox Pro	2.2951	X	X	manual	X	1.0	-	<25:00	∼1:00
Morphine	2.7	X	-	liveness	X	1.0	X	3:25	0:06
Neolite	2.0	X	-	liveness	X	1.0	X	8:16	0:09
nPack	1.1.300b	X	-	liveness	X	0.9999	X	6:55	0:11
NSPack	2.3	X	-	liveness	X	1.0	X	24:01	0:15
Obsidium	1.3	X	X	manual	-	1.0000	-	<14:00	∼0:45
Packman	1.0	X	-	liveness	X	1.0	X	6:34	0:11
PECompact2	2.79	X	-	liveness	X	1.0	X	13:28	0:11
PELock NT	2.04	X	-	liveness	X	1.0	X	4:55	0:11
PE-PaCK	1.0	X	-	liveness	X	1.0	X	5:15	0:11
PESpin	1.304	X	-	liveness	-	0.9974	-	17:07	0:43
Petite	2.3	O[3]	-	liveness	-	1.0	O[3]	7:17	0:12
RLPack	1.19	X	-	liveness	X	1.0	X	6:23	0:11
tELock	0.98	X	-	liveness	X	1.0	X	9:55	0:28
Themida	1.9.3.0	O[4]	X	manual	-	0.9963	-	<1:00:00	-
UPX	3.01w	X	-	liveness	X	1.0	X	3:56	0:10
VGCrypt	0.75	X	-	liveness	X	1.0	X	2:51	0:07
WinUPack	0.39	X	-	liveness	X	1.0	X	21:30	0:27
yoda's Crypter	1.3	X	-	liveness	X	1.0	-	5:49	0:10
yoda's Protector	1.03.3	O[5]	-	timeout	-	1.0	-	1:00:00	-

[1] memory dump at .text location is all-zero
[2] sometimes displays a dialog box
[3] crashes when invoked with arguments that cause network connectivity
[4] appears to crash in bochs
[5] does not operate normally within Bochs, but appears to unpack the original code

Notepad

packer	version	working Executable	user Interaction	termination reason	correct OEP	.text similarity	result executes	unpacking time	reconstruction time
ASPack	2.12	X	-	liveness	X	0.9863	X	2:57	0:05
ACProtect	2.0	X	X	manual	X	0.9020	-	<6:00	∼0:12
ANDpakk2	0.18	-	-	-	-	-	-	-	-
Armadillo	5.20	-	-	-	-	-	-	-	-
ASProtect	1.35	X	O[6]	liveness	X	0.9814	-	7:52	0:12
ASProtect SKE	2.3	X	O[6]	liveness	X	0.9828	-	11:59	0:13
eXpressor	1.5.0.1	X	-	liveness	X	0.9887	-	3:51	0:06
FSG	2.0	X	-	liveness	X	0.9599	X	2:51	0:05
kkrunchy	0.23a	-	-	-	-	-	-	-	-
MEW 11 SE	1.2	X	-	liveness	X	0.8741	X	3:56	0:05
Molebox Pro	2.2951	X	X	manual	X	0.9873	-	<15:00	∼0:30
Morphine	2.7	X	-	liveness	X	0.9886	X	2:45	0:05
Neolite	2.0	-	-	-	-	-	-	-	-
nPack	1.1.300b	X	-	liveness	X	0.9873	X	2:58	0:04
NSPack	2.3	X	-	liveness	X	0.8743	X	3:52	0:05
Obsidium	1.3	X	X	manual	-	0.8439	-	<9:00	∼0:45
Packman	1.0	X	-	liveness	X	0.9711	X	2:53	0:05
PECompact2	2.79	X	-	liveness	X	0.9622	X	3:19	0:05
PELock NT	2.04	-	-	-	-	-	-	-	-
PE-PaCK	1.0	-	-	-	-	-	-	-	-
PESpin	1.304	X	-	liveness	-	0.8207	-	8:58	0:24
Petite	2.3	-[7]	-	-	-	-	-	-	-
RLPack	1.19	X	-	liveness	X	0.8785	X	2:57	0:04
tELock	0.98	-	-	-	-	-	-	-	-
Themida	1.9.3.0	X	X	manual	X	0.7829	-	<1:00:00	-
UPX	3.01w	X	-	liveness	X	0.8742	X	2:41	0:05
VGCrypt	0.75	-	-	-	-	-	-	-	-
WinUPack	0.39	X	-	liveness	X	0.8742	X	3:42	0:05
yoda's Crypter	1.3	-	-	-	-	-	-	-	-
yoda's Protector	1.03.3	O[8]	-	liveness	-	0.8967	-	8:38	0:07

[6] sometimes displays a dialog box
[7] petite did not attempt compression
[8] does not operate normally within Bochs, but appears to unpack the original code

4.1.3 Analysis

First of all, it is apparent that many packers failed to generate a working, packed version of Notepad. In most cases the operating system failed to initialize a new process from the packed executable. As this is hard to debug, the reasons for this could not be determined. Packing Wget worked much better, most runtime packers that were tried successfully generated working packed executables from it.

Furthermore, .text section similarity for Notepad was generally less than for Wget. The reason for this is simple. While Wget stores import data in a separate section named .rdata, for Notepad it resides in the .text section. For the original Notepad executable, this data includes a pre-bound import address table and an import lookup table that are both 840 bytes in length, the import directory table that is 200 bytes in length, as well as the hint/name table and DLL names that amount to 3379 bytes. In total, 4459 of 30536 bytes in the .text section are taken up by import data, i.e., about 14.6%. For all samples, the import address table was modified during reconstruction. Those packers for which the reconstructed executable's .text section matches about 98% of the original text section left the original import directory table, import lookup table and hint/name table unmodified. Those for which similarity was about 90% left the import directory table and import lookup table unmodified but overwrote the hint/name table with zeroes. Those for which similarity was less than 90% overwrote all import data, some of them also modified other parts of the .text section.

The following section examine some of the reasons why reconstruction failed.

4.1.3.1 ASProtect 1.35

ASProtect 1.35 changes the original executable code such that that some branches to Windows API functions are replaced by calls into a branch function. The following listing shows executable code from the original notepad.exe. It is easy to spot the indirect subroutine calls that reference addresses in the import address table to locate the real branch target:

```
0x010070f1  ff15b8100001   call   [0x010010b8]
0x010070f7  8b75fc         mov    esi, [ebp-4]
0x010070fa  3375f8         xor    esi, [ebp-8]
0x010070fd  ff150c110001   call   [0x0100110c]
0x01007103  33f0           xor    esi, eax
```

ASProtect 1.35 replaced that code by the following instructions:

```
0x010070f1  e80a8f92ff     call    0x00930000
0x010070f6  c9             leave
0x010070f7  8b75fc         mov     esi, [ebp-4]
0x010070fa  3375f8         xor     esi, [ebp-8]
0x010070fd  e8fe8e92ff     call    0x00930000
0x01007102  a7             cmpsd
0x01007103  33f0           xor     esi, eax
```

Obviously, the three indirect subroutine calls have been replaced by direct subroutine calls to one single function at address `0x00930000`. That function resides on the heap. It was also dumped by Pandora's Bochs. It is 404 bytes long and uses code obfuscation techniques to try and prevent static analysis. Example 4.1 shows an excerpt from its beginning. It uses superfluous prefixes to unconditional jumps which are ignored by the CPU, but which might confuse some disassemblers. It inserts junk bytes which are not executed but that might throw off linear sweep disassemblers. Furthermore it calculates and stores values that are later discarded, i.e., simply overwritten, possibly in an attempt to cause a reverse engineer to waste time trying to follow and understand these calculations. This branch function later transfers control to yet another function on a different part of the heap.

It is obviously hard to automatically and generically reconstruct the original executable in light of modification of the original code and complex, obfuscated branch functions. However, Pandora's Bochs succeeded in dumping the modified executable and also the branch functions used so that they can be reverse engineered manually. This can be really helpful in light of heavy use of anti-debugging techniques in the unpacking stub, which Pandora's Bochs is immune to. ASProtect SKE employs similar obfuscation techniques.

4.1.3.2 Petite

OEP detection failed for the Wget sample packed with Petite. This is due to Petite apparently writing the instruction that branches to the real OEP to the same memory region as the actual OEP. However, the real OEP is fairly easy to spot when examining the instructions at that location:

```
0x004c3106 5f           pop edi
0x004c3107 f3aa         repz stosb
0x004c3109 61           popad
0x004c310a 669d         popf
0x004c310c 83c40c       add esp, 0ch
0x004c310f e994aafaff   jmp 0x0046dba8
```

The `popad` and `popf` instructions, followed by an unconditional branch to a fixed address as seen here are a typical example of an instruction sequence transferring control to the OEP. In fact, `0x0046dba8` is the real OEP of the executable. After changing the entry point in the PE header of the reconstructed executable to that value, it can be executed and works as intended.

4.1.3.3 PESpin

OEP detection and reconstruction failed for PESPin because it uses code obfuscation by default, most prominently the stolen bytes technique. Example 4.2 shows this for the entry point. The right column in the example shows the code that resides at the entry point within the original executable, the left column contains the code that was executed

Example 4.1 Start of the ASProtect 1.35 branch function

```
0x00930000 repne   jmp 0x00930004      ; superfluous REP prefix
0x00930003 db      0x69                ; not executed
0x00930004 push    esi                 ;
0x00930005 pushf                       ;
0x00930006 jmp     0x00930009          ;
0x00930008 db      0x0f                ; not executed
0x00930009 xor     esi, [esp+8]        ; irrelevant
0x0093000D sub     esi, 0x2d           ; irrelevant
0x00930010 sub     esp, 0x20           ;
0x00930013 adc     esi, edi            ; irrelevant
0x00930015 sbb     esi, 0x68ea95b8     ; irrelevant
0x0093001B jmp     0x0093001e          ;
0x0093001D db      0x9a                ; not executed
0x0093001E add     esi, ebp            ; irrelevant
0x00930020 lea     esi, [esp+ecx+0x34] ; esi := esp + ecx + 0x34
0x00930024 sub     esi, ecx            ; esi := esi - ecx
0x00930026 jmp     0x00930029          ;
0x00930029 lea     esi, [esi+ebx-0x34] ; esi := esi + ebx - 0x34
0x0093002D sub     esi, ebx            ; esi := esi - ebx
0x0093002F push    ebx                 ;
0x00930030 pop     [esi+0x0c]          ;
0x00930033 jmp     0x00930037
0x00930037 sbb     ebx, esi            ; irrelevant
0x00930039 push    ecx                 ;
0x0093003A pop     [esi+4]             ;
0x0093003D jmp     0x00930040
           ...
```

by the PESpin-packed sample instead. It is easy to see that PESpin executes most instructions as is, albeit in a different location and interspersed with non-conditional jumps. Only some `CALL` instructions from the original executable have occasionally been replaced by equivalent instruction sequences, e.g. the push of a return address, followed by an unconditional branch to the subroutine. Note that non-reachable junk bytes have been omitted in the left column to fit the example to one page. PESpin obfuscates API calls similarly. Another problem with PESpin is that it appears to overwrite the section table during unpacking.

4.1.3.4 Morphine

Morphine works slightly differently from other packers in that the original executable is not unpacked to a memory area within the bounds of the packed image. Instead, it

Example 4.2 PESpin OEP obfuscation using stolen bytes

```
PESpin dump                              | original executable
-----------------------------------------+-----------------------------------------
            ...                          |
0x004c4b26  or     edx, eax              |
0x004c4b28  test   ecx, 0x88b66320       | entry point:
0x004c4b2e  push   18h                   | 0x0046dba8  push   0x0049c7f0
0x004c4b30  jmp    0x004c4b33            |
0x004c4b33  push   0x1394f37e            | 0x0046dbaa  push   0x0049c7f0
0x004c4b38  sub    [esp], 0x134b2b8e     |
0x004c4b3f  push   0x004c4b49            | 0x0046dbaf  call   0x0046fbe0
0x004c4b44  jmp    0x0046fbe0            |
0x004c4b49  mov    edi, 0x94             | 0x0046dbb4  mov    edi, 0x94
0x004c4b4e  jmp    0x004c4b51            |
0x004c4b51  mov    eax, edi              | 0x0046dbb9  mov    eax, edi
0x004c4b53  jmp    0x004c4b56            |
0x004c4b56  push   0x004c4b60            | 0x0046dbbb  call   0x0046a050
0x004c4b5b  jmp    0x0046a050            |
0x004c4b60  mov    [ebp-0x18], esp       | 0x0046dbc0  mov    [ebp-0x18], esp
0x004c4b63  jmp    0x004c4b66            |
0x004c4b66  mov    esi, esp              | 0x0046dbc3  mov    esi, esp
0x004c4b68  jmp    0x004c4b6b            |
0x004c4b6b  mov    [esi], edi            | 0x0046dbc5  mov    [esi], edi
0x004c4b6d  jmp    0x004c4b70            |
0x004c4b70  push   esi                   | 0x0046dbc7  push   esi
0x004c4b71  jmp    0x004c4b74            |
0x004c4b74  call   [0x004c74f7]          | 0x0046dbc8  call   [GetVersionExA]
0x004c4b7a  jmp    0x004c4b7d            |
0x004c4b7d  mov    ecx, [esi+0x10]       | 0x0046dbce  mov    ecx, [esi+0x10]
0x004c4b80  jmp    0x004c4b83            |
0x004c4b83  mov    [0x004c0528], ecx     | 0x0046dbd1  mov    [0x004c0528], ecx
0x004c4b89  jmp    0x004c4b8c            |
0x004c4b8c  mov    eax, [esi+4]          | 0x0046dbd7  mov    eax, [esi+4]
0x004c4b8f  jmp    0x004c4b92            |
0x004c4b92  mov    [0x004c0534], eax     | 0x0046dbda  mov    [0x004c0534], eax
0x004c4b97  jmp    0x004c4b9a            |
0x004c4b9a  mov    edx, [esi+8]          | 0x0046dbdf  mov    edx, [esi+8]
0x004c4b9d  jmp    0x004c4ba0            |
0x004c4ba0  mov    [0x004c0538], edx     | 0x0046dbe2  mov    [0x004c0538], edx
0x004c4ba6  jmp    0x004c4ba9            |
0x004c4ba9  mov    esi, [esi+0x0c]       | 0x0046dbe8  mov    esi, [esi+0x0c]
0x004c4bac  jmp    0x004c4baf            |
0x004c4baf  and    esi, 0x7fff           | 0x0046dbeb  and    esi, 0x7fff
0x004c4bb5  jmp    0x004c4bb8            |
0x004c4bb8  mov    [0x004c052c], esi     | 0x0046dbf1  mov    [0x004c052c], esi
0x004c4bbe  jmp    0x004c4bc1            |
0x004c4bc1  jmp    0x0046dbf7            |
0x0046dbf7  cmp    ecx, 2                | 0x0046dbf7  cmp    ecx, 2
```

asks the operating system to allocate memory at the base address of the original image, unpacks the original code and data to that memory area and finally manipulates the process's loaded module list (see section 3.3.3.1, figure 3.2) to include an entry for the unpacked "image".

Reconstruction of a working executable for Notepad failed at first. It appeared that the Windows loader failed to resolve imports for two dynamic link libraries, even though the import data seemed to be correct. The solution was rather simple. Notepad uses so-called *bound* imports, i.e., its IAT is pre-populated with the actual addresses of imported symbols, instead of RVAs of hint/name table entries (see section 2.3.1.4). Because this information is only valid for specific versions of the imported DLLs, their timestamps are kept in the so-called *bound imports table* that is referenced from the optional header data directories. When such a program is executed and the timestamps of the system DLLs do not match those in the bound imports table, the Windows loader re-populates the IAT by resolving entries from the import lookup table and overwriting IAT entries with the correct virtual addresses. While most other runtime-packers clear the bound imports data directory, Morphine does not. When the presented solution reconstructed the IAT, the actual symbol addresses were overwritten with RVAs of hint/name table entries, effectively un-binding the image. The import lookup table was not reconstructed and the bound imports table was not cleared. When trying to execute the reconstructed image on another Windows system, the timestamps of some system DLLs matched those in the bound import table, so that some entries in the IAT were not properly resolved and attempts to use these entries in indirect branches ended up branching to non-existing memory. This problem was later eliminated by always clearing the bound imports data directory during reconstruction.

4.1.3.5 Other Results

The other packers for which reconstruction failed mostly employed code obfuscation techniques as well:

- Molebox Pro redirects some imports to custom branch functions.
- Obsidium resets section sizes and the image base in the PE header to zero. It also uses stolen bytes to obfuscate the OEP and redirects IAT entries to an obfuscated branch function.
- Themida appears to redirect some IAT entries to a branch function, as well as directly modifying some branch instructions within the original code. It also claims to employ instruction virtualization in its default configuration.
- yoda's Crypter and yoda's Protector both redirect some IAT entries to a branch function. The samples packed with yoda's Protector also appeared to not work properly within Bochs, i.e., while it unpacked the original code, it did not appear to execute any of the original binaries' functionality, even in a non-instrumented version. The reason could not be determined.

It could not be determined why the reconstructions of the samples packed by ACProtect did not execute correctly. Both reconstructions start execution but shortly thereafter terminate without apparent error. Neither could it be determined why reconstruction of the Notepad sample failed for eXpressor. Debugging the reconstructed executable hinted at an incorrect .rsrc section, but this could not be determined with absolute certainty.

As was to be expected, unpacking times for the larger Wget executable, were higher than those for Notepad, due to the greater amount of data that had to be unpacked. For some packers, unpacking could not complete within a hour, highlighting a severe limitation of the current implementation.

Note that Pandora's Bochs was able to extract hidden code from most samples, even if working, executable images could not always be generated. The only exceptions were Armadillo and ANDpakk2 which did not complete unpacking within the time limit of one hour.

4.2 Malware Samples

For evaluating Pandora's Bochs on malware, a set of 409 malware samples that were collected by the RWTH Aachen honeynet over the course of one month was used. Some of the samples' properties were:

- The samples had an average size of 100 kB.
- ClamAV [Sou07] identified 379 files as infected, and assumed most of them to be bots.
- PEiD [pei07] identified 239 of the samples to be compressed, 126 as not compressed (or rather: not containing known packer signatures) and failed to process 44 samples. It detected 203 samples to be packed by UPX. UPX, a runtime packer that can also decompress files it created, determined 161 samples to not have been packed by UPX, assumed 114 samples to have been packed by modified versions of UPX (which it cannot decompress), failed decompression of 104 files for other reasons (e.g. wrong checksums, or other indicators of file corruption) and unpacked 30.

Executing the malware samples within Pandora's Bochs yielded the following results:

- In 361 cases, the guest operating system started executing the sample. The average execution time of Pandora's Bochs for these samples was 7 minutes and 21 seconds.
- 343 samples executed modified memory within the image (see section 3.3.5.3) at least once and were thus considered packed (see figure 4.1).
- From these 343 samples, Pandora's Bochs was able to generate 332 unpacked images.

- In 152 cases, the observed processes spawned an instance of *dwwin.exe*, Windows's error reporting tool, signifying that the observed process experienced an error. Further investigation showed that these samples also crashed in non-instrumented instances of Bochs, but appeared to work within VMWare. However, the reason for these crashes could not be determined, in part due to Bochs's poor performance that rendered it unusable for interactive debugging. It appears that these crashes typically happen after unpacking is complete during execution of the original code.

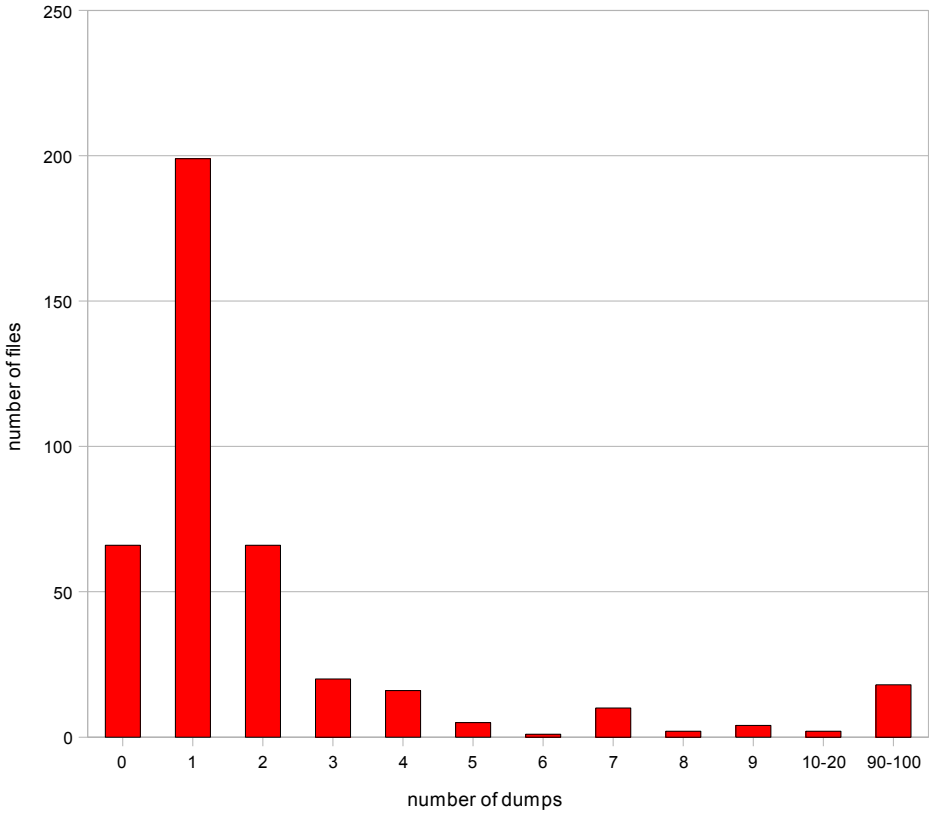


Figure 4.1 – Dump frequencies

As the original images of these executables are not available, it is difficult to evaluate the quality of the extracted images. Some of the approaches that were taken will now be documented in greater detail.

String Matching Similar to the method presented in [KPY07], the extracted binaries and the corresponding original samples were checked for strings that are used in the IRC protocol [OR93], a protocol commonly used by malicious bots to communicate with their master. Pattern matching was done using the *pcregrep* utility which is distributed along with the PCRE library [Haz07]:

```
pcregrep -i -c \
"(PRIVMSG|USER|NICK|PING|PONG|MODE|JOIN|PART|TOPIC|NAMES|LIST\
|INVITE|KICK|NOTICE|MOTD|WHO|WHOIS|WHOWAS|DCC SEND|DCC CHAT)(\x00|\s|%)"
```

The results can be seen in figure 4.2. It is noteworthy that about half of the original samples that could be unpacked contained one or two of the strings searched for. This might be an indication of these files having been compressed by packers using dictionary coding algorithms, e.g., by UPX. The extracted files typically contained more of the strings that were searched for. Those were also subjectively of better quality, i.e., they seemed more complete and more likely to represent zero-terminated ASCII strings that would be used by C programs to communicate with IRC servers.

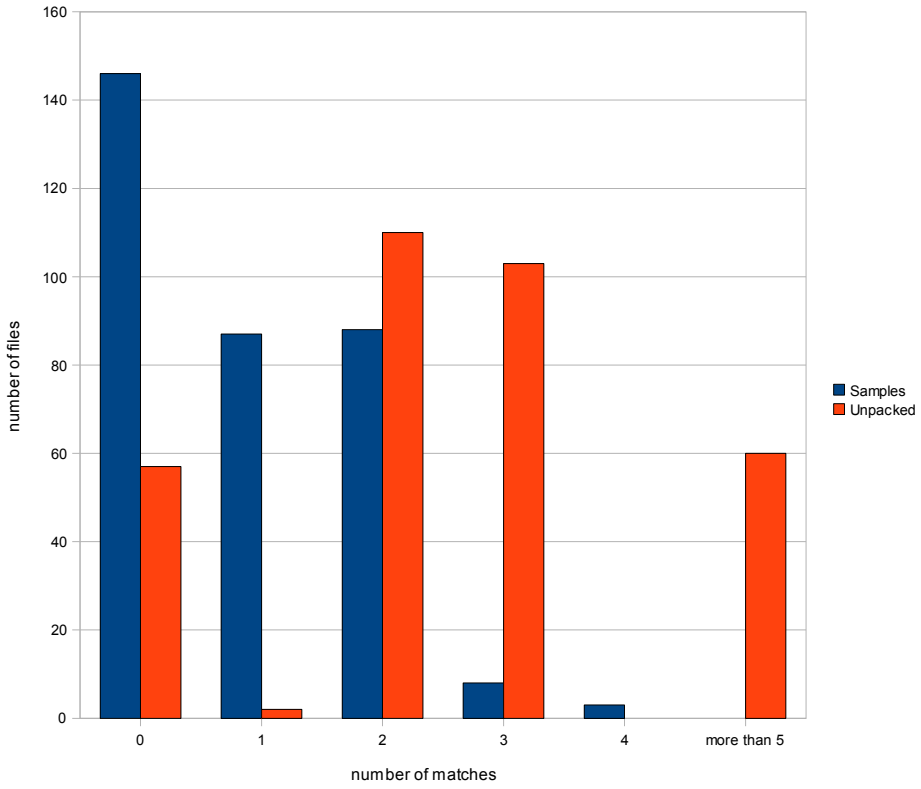


Figure 4.2 – IRC protocol string frequencies

Additionally, the unpacked binaries and their original samples were searched for Windows registry keys using the following command:

```
pcregrep -i -c \
'(SOFTWARE|HARDWARE|SAM|SECURITY|SYSTEM|HKEY(_[A-Z])+)\\[A-Z0-9_\\]+\x00'
```

Registry keys are commonly queried and manipulated by Windows applications. As can be seen in figure 4.3, only 2 of the original samples appeared to contain registry keys, whereas 250 of the unpacked samples contained at least one. Of these, 120 contained the string `SOFTWARE\Microsoft\Windows\CurrentVersion\Run\end` that is a registry key than applications can add entries to so that the operating system executes them automatically during system startup or when a user logs in. Adding an entry to this registry key enables malware to regain control when the infected system is rebooted.

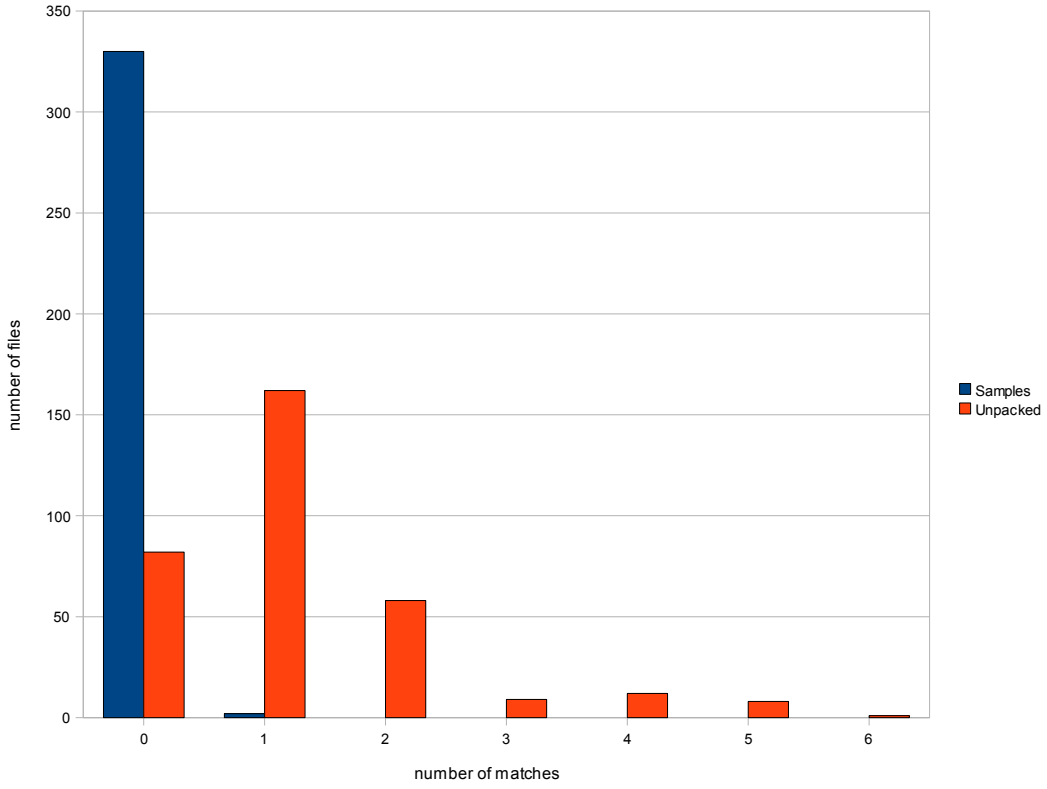


Figure 4.3 – Registry key string frequencies

OEP Detection While analysing the unpacked executables, it became apparent that there were two kinds of assumed OEPs that dominated over others. The candidate OEP in example 4.3 appears to first obtain a handle to the image file that was used to create the current process. It then deletes a file named "ftpupd.exe", quite likely to erase traces of itself from the file system and then calls some subroutines before terminating execution. This could well be a valid entry point for an executable. Another candidate OEP, shown in example 4.4, appears to start with a valid function prologue that saves the current frame pointer on the stack, sets up a new stack frame, and reserves space

Example 4.3 Assumed OEP #1

```
0x314324ef  push    0               ; lpModuleName
0x314324f1  call    GetModuleHandleA
0x314324f7  push    offset FileName ; "ftpupd.exe"
0x314324fc  mov     [0x31435040], eax
0x31432501  call    DeleteFileA
0x31432507  call    0x314320B6
0x3143250c  push    offset aUterm20 ; "uterm20"
0x31432511  call    0x314320e4
0x31432516  pop     ecx
0x31432517  mov     hObject, eax
0x3143251c  call    GetLastError
0x31432522  cmp     eax, 0xb7
0x31432527  jnz     short 0x31432531
0x31432529  push    1               ; uExitCode
0x3143252b  call    ExitProcess
```

Example 4.4 Assumed OEP #2

```
0x00410e3c  push    ebp
0x00410e3d  mov     ebp, esp
0x00410d3f  add     esp, 0x0fffffed0
0x00410e45  xor     eax, eax
0x00410e47  mov     [ebp-0x0130], eax
0x00410e4d  mov     [ebp-0x012C], eax
0x00410e53  mov     [ebp-0x0128], eax
            ...
```

for local variables and sets them to zero. In total, for 93 of the extracted files the detected OEP was similar to example 4.3, for 179 it was similar to example 4.4, and for an additional 39 the detected OEP started with a minimal function prologue consisting of only the first two instructions depicted in example 4.4 that set up a new stack frame. While it is is likely that the OEPs were detected correctly for these files, this cannot be said with absolute certainty.

Of the 21 remaining reconstructed executables, 14 appeared to have been packed with Themida (i.e., they contained a section named “Themida”) and the code at their detected OEPs appears to be obfuscated, i.e., it is unlikely that the OEP was correctly detected in these cases. The other files were detected by PEiD to have been packed by Yoda’s Protector, ASProtect SKE, or were not detected. For none of them the assumed OEP appeared to be correct.

Signature-Based Malware Detection Out of the 332 samples that could be extracted, the ClamAV [Sou07] malware scanner determined 324 to be known malware. After unpacking, only 284 were detected any longer, a result that appeared to be discouraging at first. However, further investigation yielded interesting results.

The 179 samples whose unpacked version sported an OEP like the one depicted in example 4.4, were originally detected by ClamAV to be various variants of the *Trojan.Gobot* family or as *Trojan.Downloader.Delf-35*. After unpacking, however, *all* were detected as *Trojan.Gobot-11*. This highlights some of the disadvantages of signature-based detection. The signatures that matched the packed executables largely target specific locations within the unpacking stub or the packed data, i.e., they are easily rendered ineffective by using a different runtime packer. The signature for Trojan.Gobot-11 does not target a specific location, but matches a byte string that only appears in the unpacked, original binary. Note that ClamAV detected 29 of the original samples as Trojan.Gobot-11, despite the corresponding signature not appearing in those files. This is likely due to the fact that ClamAV is able to unpack files packed by some runtime packers prior to signature matching, e.g., UPX, FSG, MEW, Petite or NsPack. It appears to have failed to unpack the other 150 Gobot samples correctly, however.

The 14 samples that appeared to have been packed with Themida were detected as *Trojan.Wootbot-17* after unpacking. ClamAV's signature matches a byte string that is related to the malware offering functionality for stealing copy-protection keys of a popular game. Prior to unpacking, the samples were detected as various kinds of malware, either by signatures that match less than 100 bytes at the entry point of the unpacking stub, or by matching a checksum against the first section. At least the former do not appear to be very good signatures, as they at best just identify the unpacking stub. Why section checksums match remains unclear.

For the 93 samples which contained an OEP similar to the one in example 4.3, the situation is less clear. Prior to unpacking they are detected mostly as instances of the *Worm.Korgo*, *W32.Virut* and *Worm.Padobot* families. After unpacking, detection rates for Worm.Korgo increased from 18 to 34, for W32.Virut they decreased from 34 to 21, for Padobot they decreased from 39 to 0 and overall detection decreased from 93 to 55.

These results show that runtime-packers and executable protectors can be used by malware authors to easily evade signature-based detection. The signatures used by ClamAV appear to often match parts of unpacking stubs or packed data. While unpacking cannot improve detection rates for malware identified by such signatures, this is well possible for malware that is detected by signatures matching the original, unpacked code or data contained in malware.

Execution To further test reconstruction quality, an attempt was made to execute the reconstructed images within Pandora's Bochs. For 316 out of the total 322, the guest operating system started execution, i.e., it created andinitialized a process for them and executed an instruction at the specified entry point. None of these images executed modified memory locations. In 195 cases, Windows's error reporting tool was executed. It is unclear how to interpret these numbers in light of the frequent crashes also experienced during unpacking.

5 Conclusion

This last chapter shall highlight some of the lessons learned during the implementation of Pandora's Bochs, illustrate some of the limitations of the presented solution and show what future work could address these issues. It will conclude by recapitulating the achievements, comparing the presented solution to related work and some closing remarks.

5.1 Accomplishments

The presented solution succeeds at solving the tasks that were set. Pandora's Bochs can identify individual processes without calling upon functionality offered by the Windows XP SP2 guest operating system. Instead it locates the data structures Windows itself relies on for process management, and parses their contents. Similarly, Pandora's Bochs can obtain information about image files mapped into a process's address space, e.g., their base addresses, sizes and file names, and parse these images, e.g., to obtain information about exported and imported symbols.

Pandora's Bochs calls upon Bochs's instrumentation facilities to trace execution of monitored processes at the basic block level. Furthermore, memory writes can be recorded and branches to memory regions that have been modified are detected. All branches and writes executed by a monitored process are also logged to an SQLite database. When Pandora's Bochs detects a branch to previously modified memory, this is indicative of the completion of an unpacking stage, and the target memory range (e.g., the containing image, or, a memory block allocated on the heap) is dumped to the SQLite database, select parts of memory surrounding the branch target are marked as clean, and execution continues so that additional stages can be unpacked, if present. Pandora's Bochs stops execution either after it determines that unpacking of additional code is unlikely, by tracking "innovation" of monitored processes, or, after a configurable timeout.

To acquire pages that the guest operating system's demand-paging mechanism has not yet mapped into memory, Pandora's Bochs can simulate page faults to coerce the guest operating system into bringing them in from disk. Pandora's Bochs is immune to anti-debugging techniques, as it operates independently of the emulated CPU's and the guest operating system's debugging facilities and is thus unaffected by attempts to manipulate or abuse these.

Once unpacking is complete, an attempt is made to reconstruct a valid executable image. Currently, this works well in many cases, but fails in light of executable protectors that heavily manipulate the packed images' structure or manipulate and obfuscate API

calls and/or the original code. However, Pandora's Bochs usually succeeds in extracting hidden code from runtime packed executables that can then be subjected to static analysis, even if it cannot generate a fully working executable.

Pandora's Bochs can also aid signature-based malware detection, depending on the nature and quality of the signatures.

5.2 Limitations

The results of the evaluation that were presented in chapter 4 highlight some of the limitations of Pandora's Bochs's current implementation.

Speed Although unpacking speed was not a major design goal, it has become an issue during development, experimentation and evaluation. While Pandora's Bochs performs adequately on small executables, or executables packed with rather simple packers, the emulator's performance degrades for more advanced executable protectors. This is not only an issue for unpacking as such, it is also a severe limitation when experimenting with new (e.g., packer-specific) algorithms.

Compatibility As seen in sections 4.1 and 4.2, a high number of malware samples failed to execute properly, despite being able to do so on other platforms. While this did not appear to prevent successful unpacking, this is an issue that surely needs to be addressed.

OEP detection While Pandora's Bochs succeeds at correctly detecting the OEP for most packers that were experimented with, it doesn't always do so. This was often due to OEP obfuscation by executable protectors, but in some cases also due to deficiencies in the detection algorithm.

Recovering Imports As seen in section 4.1.3 import information was not always recovered correctly. Sometimes this was due to the runtime packers using a simple redirection layer to obfuscate function calls, in some cases it was due to more elaborate obfuscation schemes, involving obfuscated redirection layers and stolen bytes.

Protection As expected (see section 3.2), and analysed in section 4.1.3, the obfuscation methods employed by some executable protectors severely lessen the chance of the presented solution reconstructing a working executable image. Stolen bytes are an issue for both OEP detection and import recovery, IAT-based API redirection and branch functions, as well as modification of the original code pose additional challenges. Elaborate schemes like the shifting decode frame method or instruction virtualization are well beyond the scope of Pandora's Bochs.

5.3 Future Work

To address some of the deficiencies that were highlighted, the following tasks are presented as potential future work:

Speed and Compatibility While Pandora's Bochs's unpacking performances is adequate for simple packers, it is far from satisfactory for more advanced packers of higher algorithmic complexity. Profiling revealed that Pandora's Bochs spent about equally as much time in its emulation code as it spent in Python instrumentation code.

To address emulation speed, it might be worthwhile to explore the option of using a different, better performing virtual machine as a basis for dynamic instrumentation. One such option would be to use QEMU, another open source virtual machine, whose author claims it to be 30 times faster than Bochs [Bel05, section 5]. While it currently lacks a publicly available instrumentation interface, other researchers have successfully added such functionality to it [KPY07, CKJ+05]. Using QEMU could also benefit compatibility, as QEMU is currently being used by a much wider audience than Bochs so that it sees much more real-world testing.

While Python, an interpreted language, does not offer as much performance as compiled languages, its benefits (especially ease of development and flexibility, see section 3.3.5) outweigh this deficiency. It would not be advisable to replace the Python instrumentation implementation with an implementation in a compiled language (e.g., C or C++) while development is still ongoing. It might however be possible to create new Python modules in a compiled language that exports optimized implementations of some of the data types and algorithms that are used.

OEP detection Most binaries that are runtime-packed were originally generated by a standard compiler. These usually include standard startup routines at the entry points of such executables, which are typically linked in from standard libraries. For Microsoft's Visual C compilers, these routines are normally `mainCRTStartup`, `wmainCRTStartup`, `WinMainCRTStartup`, or `wWinMainCRTStartup` from the C runtime library (CRT).

Building a database containing fingerprints of these standard startup routines could be helpful for improving OEP detection mechanisms. Such a database could even help correctly detecting the OEP for runtime-packer that use (non-obfuscated) stolen bytes for OEP obfuscation.

Reconstruction Reconstructing working PE executables did not always succeed. The major obstacles faced were related to import recovery and import obfuscation. Import recovery for simple redirection layers could be improved by trying to follow the control flow through those layers until a symbol exported by a dynamic link library is reached. On the other hand, generic reconstruction for more elaborate import obfuscation techniques does not appear to be feasible.

Executable Protectors In light of the seemingly arbitrary amount of complexity that can be introduced into executable protectors, it is doubtful whether attempting to create a generic solution to unpack packed and heavily protected executables is a viable approach, especially when the goal is to manually analyse the result afterwards. In retrospective, a more practical approach to unpacking protected executables would be addressing the deficiencies of the tools that are currently used for manual unpacking, interactive debuggers in particular. Their main weakness is that they typically rely on CPU and operating-system facilities that can both be detected and abused by the debuggee to render debugging ineffective or cumbersome.

An obvious solution to these problems would be separating the debugger from the operating system and the CPU, so that both can operate unaffected by its presence. While this would be hard to implement on real hardware (although probably possible using hardware-assisted virtualization) free and open source virtual machines present powerful technology to which such functionality could be added without affecting the environment they present to guest operating systems and applications. The techniques presented in this thesis could be used by such a debugger to gain an understanding of some of the guest operating system's semantics without making the guest OS aware of the debugger's presence.

Such a debugger should, in addition to other standard debugging features, provide "break on execute" facilities similar to the ones presented in section 3.3.5 to enable it to break on control flow reaching potential OEPs. It should be able to record an execution history so that control flow obfuscation can be analysed conveniently, and it should allow an analyst to dump memory regions and provide tools to analyse and manipulate them to create viable inputs for other static analysis tools.

5.4 Conclusion

This thesis set out to solve the task of automatically unpacking executable files that have been packed using runtime-packers or executable protectors, a strategy commonly employed by malware to escape signature-based detection and prevent static analysis.

The presented solution, dubbed *Pandora's Bochs*, succeeds at unobtrusively monitoring execution of runtime-packed binaries and it can extract hidden code and data from a wide variety of runtime packers and executable protectors. In many cases it can also successfully generate valid and executable PE images from the extracted information, which can be analysed using common static analysis tools. Pandora's Bochs does not need to run any additional code on the guest system and is thus transparent to the guest operating system and applications, rendering it largely immune to anti-debugging techniques that are employed by some unpacking stubs.

Compared to other automated unpackers, the presented solution offers some advantages. Unlike Pandora's Bochs, Renovo [KPY07] relies on a kernel module that is inserted into the guest operating system to make instrumentation code aware of operating system semantics. Renovo also does not appear to generate valid PE images from the unpacked code and data. Renovo does however offer much better performance, it typi-

cally needs less than a minute to unpack a packed binary. PolyUnpack [RHD+06] relies on correct disassembly to build a static model of an executable that is to be unpacked, and it uses the Windows debugging API for instrumentation. Unlike Pandora's Bochs, these methods are easily thwarted by code obfuscation and anti-debugging techniques. Saffron's [QV07] Pin-based method modifies the address space of the instrumented executables and is thus easily detected by unpacking stubs that employ integrity-checking mechanisms. Saffron's page fault handler debugging is a powerful technique, but it only works on real machines, so that it cannot profit from the easier management of virtual machines. The malware normalization method presented in [CKJ+05] does not attempt reconstruction of working executables and implementation details are scarce.

It is apparent that Pandora's Bochs offers some advantages over other automated unpackers, but during evaluation, some of its deficiencies were also discovered, namely its subpar emulation performance and compatibility, as well as the presented solution's inability to correctly reconstruct valid PE images from obfuscated data. Some options to remedy this were proposed. They pose interesting challenges and will be pursued as future work.

To help remedy the scarcity of freely available tools that aid malware analysis, and to facilitate further research by academic, corporate and independent researchers, the source code of Pandora's Bochs will be released under a free software license.

A .bochsrc

```
config_interface: textconfig
display_library: x
megs: 256
romimage: file="BIOS-bochs-latest", address=0x000f0000
vgaromimage: file="VGABIOS-lgpl-latest-cirrus"
boot: cdrom, disk
floppy_bootsig_check: disabled=0
floppya: 1_44="/dev/fd0", status=ejected
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, mode=sparse, translation=auto,
             path="winxpsp2sparse.img.0",
             cylinders=40634, heads=16, spt=63,
             biosdetect=auto, model="Generic"
ata0-slave: type=cdrom, path="winxpsp2.iso",
            status=ejected, biosdetect=auto, model="PSEUDO"
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata2: enabled=0
ata3: enabled=0
parport1: enabled=1, file="parport.out"
parport2: enabled=0
com1: enabled=1, mode=null, dev=""
com2: enabled=0
com3: enabled=0
com4: enabled=0
usb1: enabled=0
i440fxsupport: enabled=1, slot2=cirrus, slot3=ne2k
vga_update_interval: 300000
vga: extension=cirrus
cpu: count=1, ips=10000000, reset_on_triple_fault=0
text_snapshot_check: enabled=0
private_colormap: enabled=0
clock: sync=realtime, time0=local
ne2k: enabled=1, ioaddr=0x240, irq=9, mac=fe:fd:de:ad:be:00,
      ethmod=tap, ethdev=tap0
pnic: enabled=0
sb16: enabled=0
log: bochsout.txt
```

```
logprefix: %t%e%d
debugger_log: -
panic: action=ask
error: action=report
info: action=report
debug: action=ignore
pass: action=fatal
keyboard_type: mf
keyboard_serial_delay: 250
keyboard_paste_delay: 100000
keyboard_mapping: enabled=0, map=
user_shortcut: keys=none
mouse: enabled=0, type=ps2
```

B Graphs

The following pages contain some graphs of the unpacking phases of some of the runtime packers that were experimented with. Memory writes are marked in red, code execution is marked in green. For greater clarity, memory writes cast a "shadow" to the right so that code execution in modified memory is easier to identify. Subsequent writes to the same location are displayed with increased brightness. Note that events outside of the initial image, e.g., on the heap or within dynamic link libraries, are not visualised.

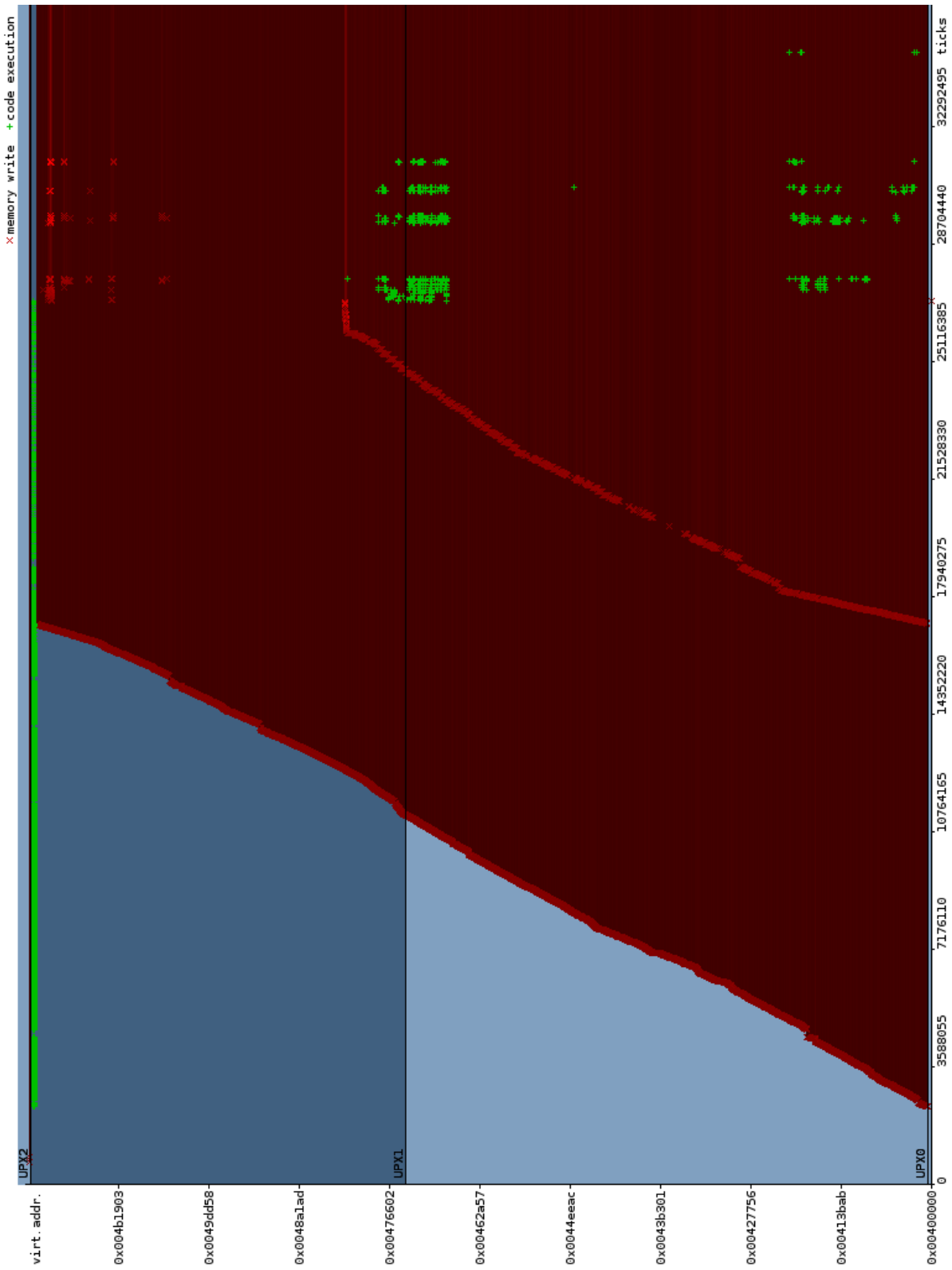


Figure B.1 – Unpacking UPX 3.01w

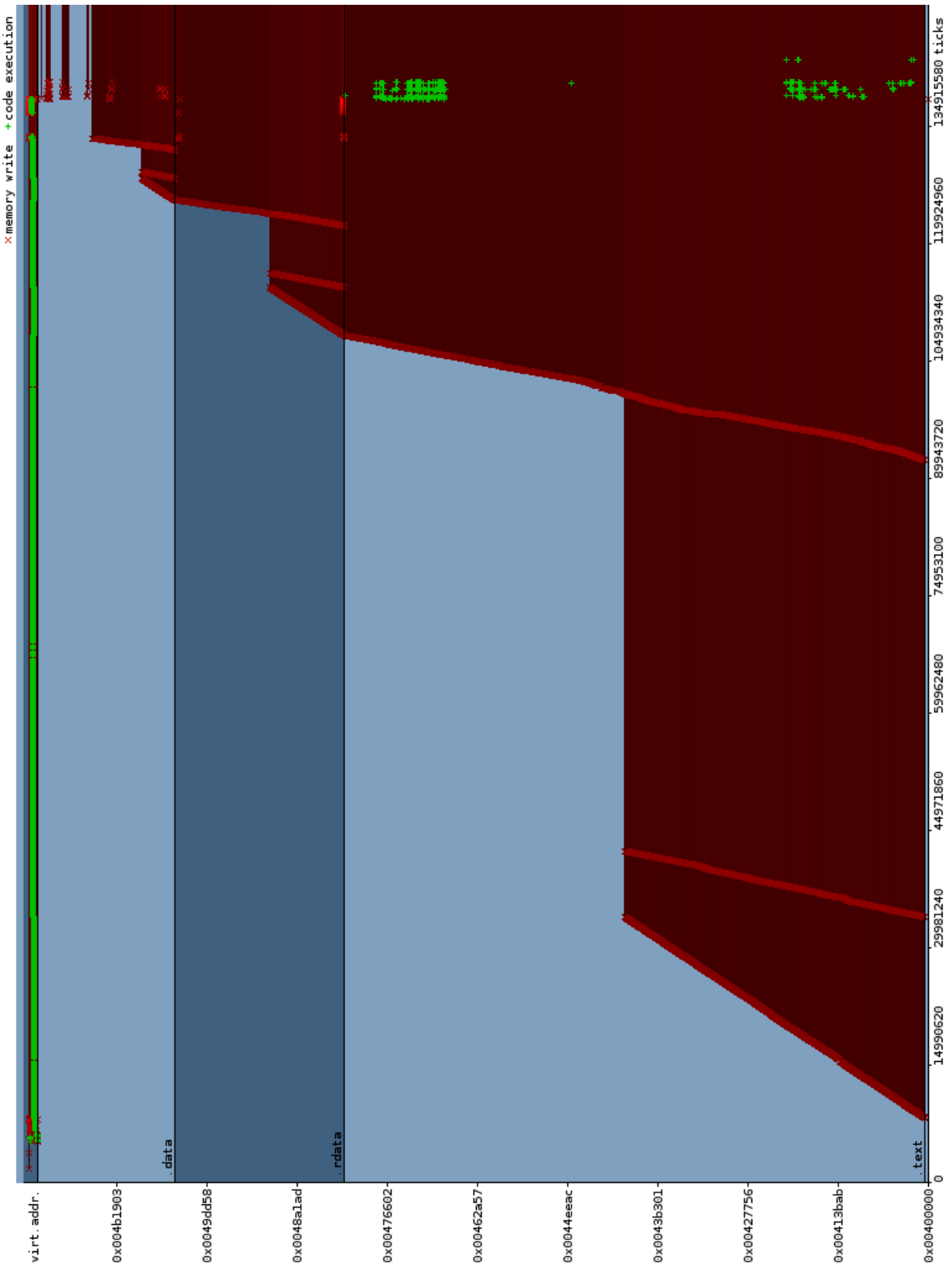


Figure B.2 – Unpacking tELock 0.98

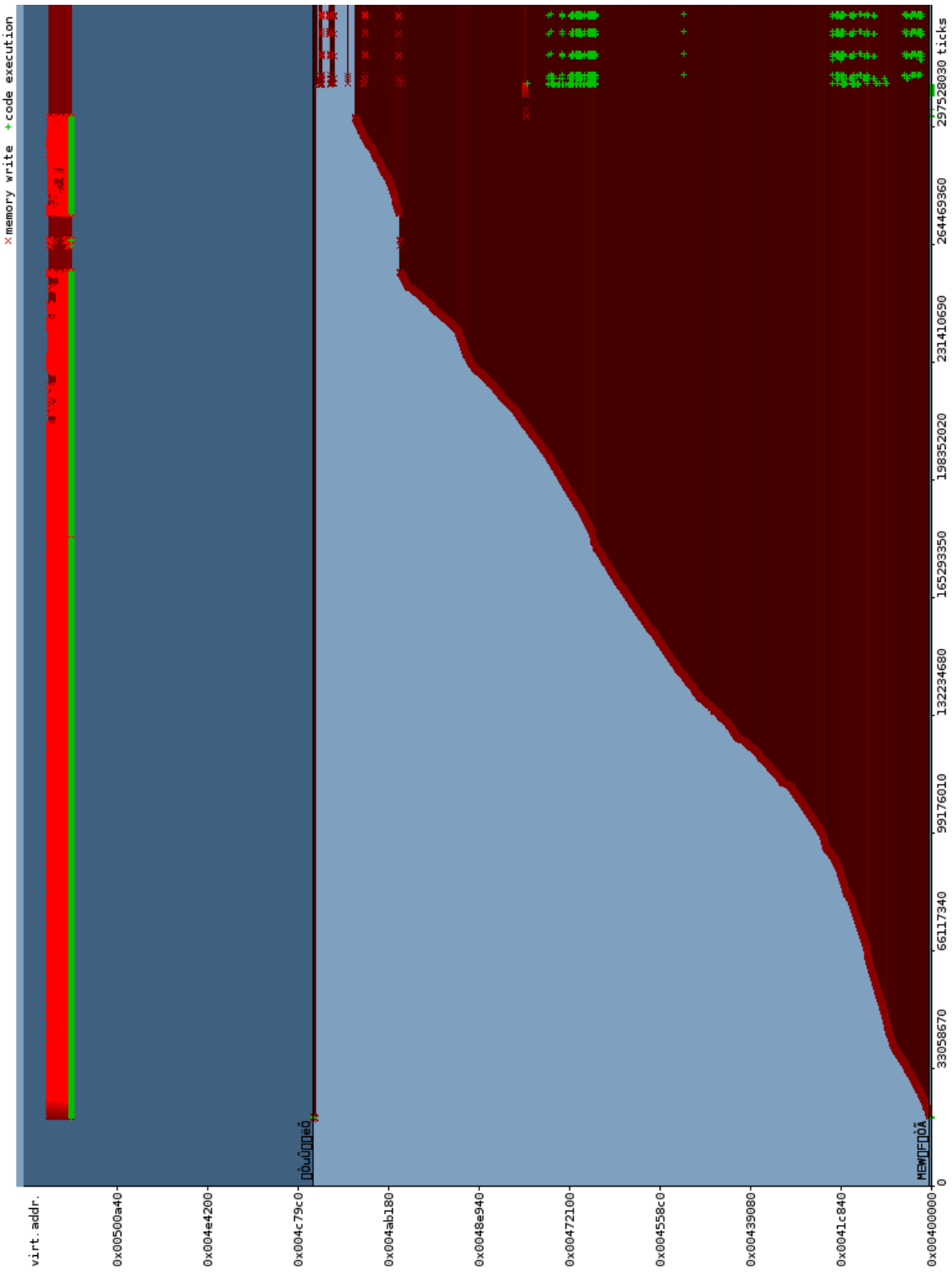


Figure B.3 – Unpacking MEW 11 SE 1.2

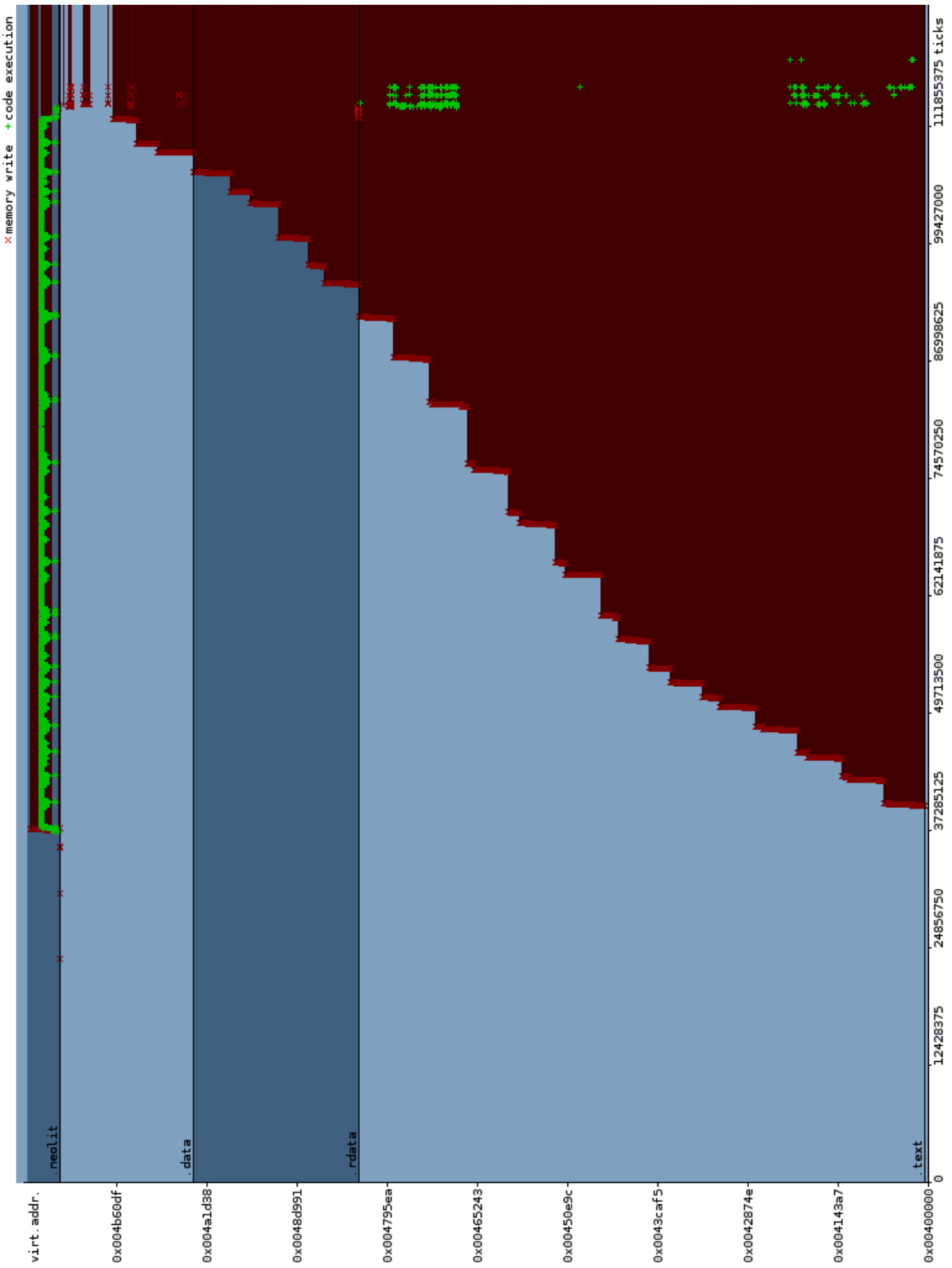


Figure B.4 – Unpacking Neolite 2.0

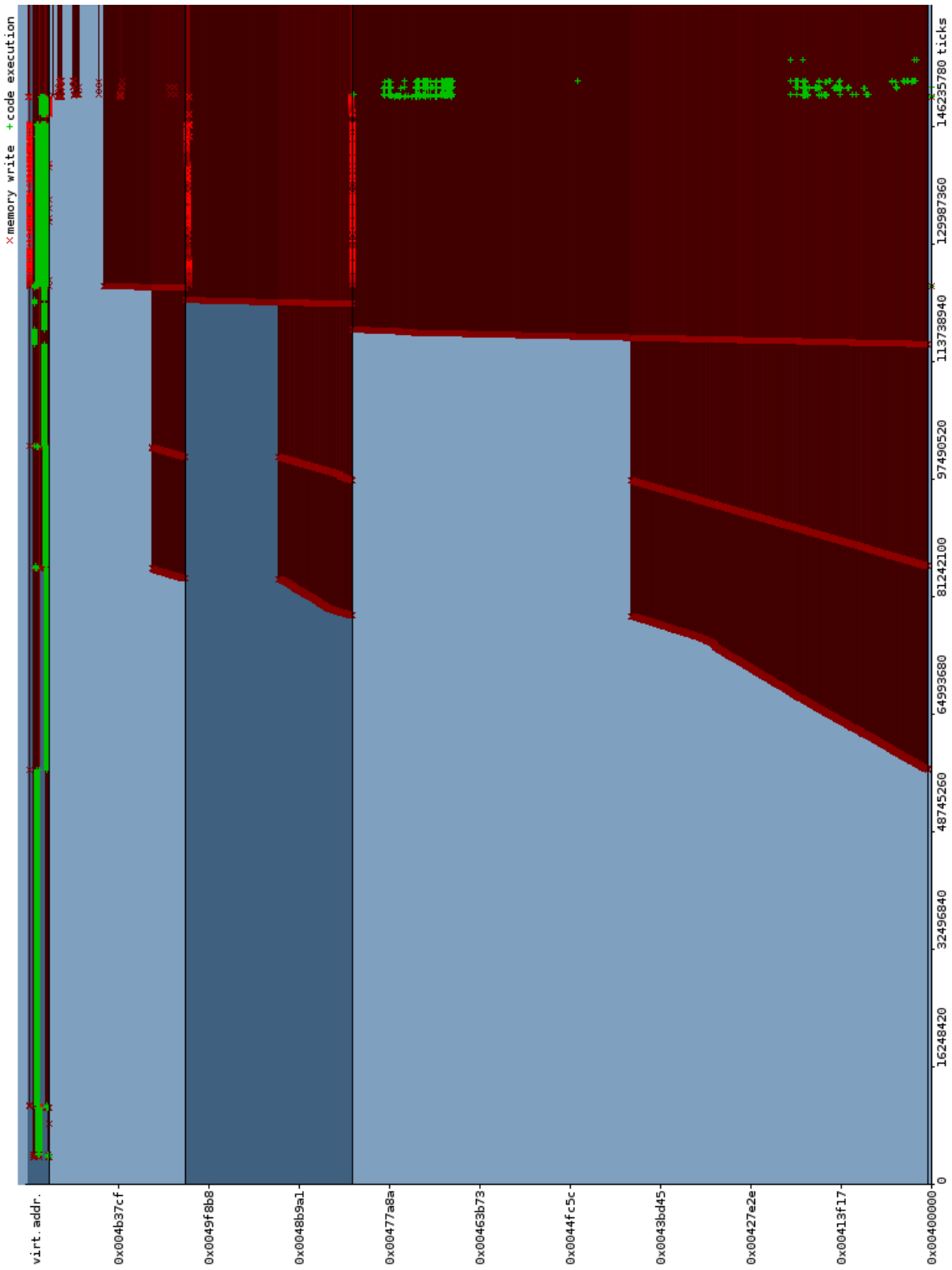


Figure B.5 – Unpacking PESpin 1.304

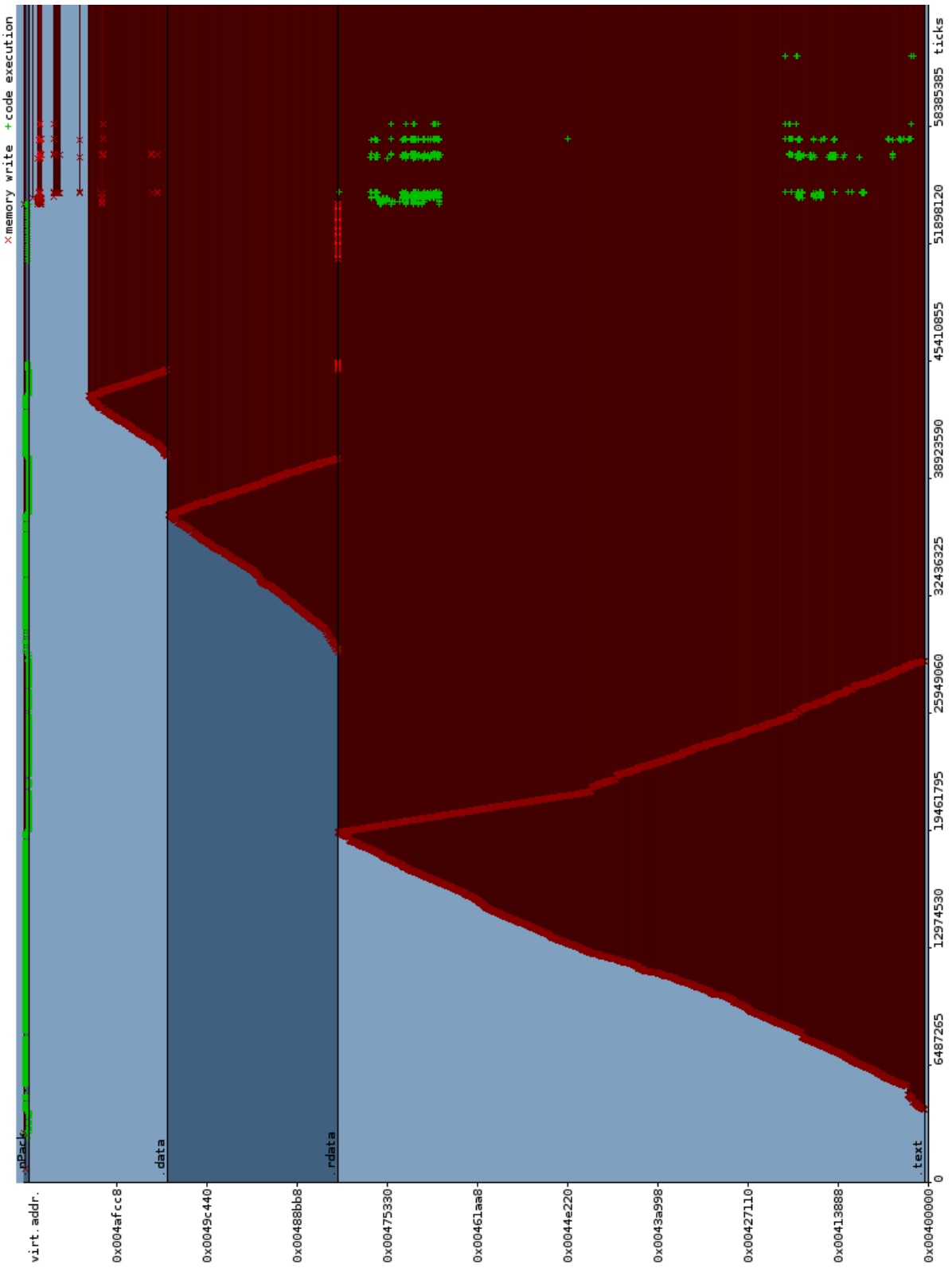


Figure B.6 – Unpacking nPack 1.1.300 beta

Bibliography

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 41–46, 2005.

[Bel08] Fabrice Bellard. QEMU, 2008. `http://fabrice.bellard.free.fr/qemu/`.

[BHKW05] Paul Bächer, Thorsten Holz, Markus Kötter, and Georg Wicherski. Know your enemy: Tracking botnets, March 2005. `http://www.honeynet.org/papers/bots/`.

[Boc07] The Bochs Development Team. Bochs - The Cross Platform IA-32 Emulator, 2007. `http://bochs.sourceforge.net/`.

[bs05] bugcheck and skape. Windows kernel-mode payload fundamentals. *Uninformed*, Vol 3, 2005. `http://www.uninformed.org/?v=3&a=4&t=sumry`.

[Car07] Ero Carrera. pefile, 2007. `http://code.google.com/p/pefile/`.

[CERL02] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical report, Sun Microsystems Laboratories, University of Queensland,Harvard University, January 2002.

[CKJ+05] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.

[Coh84] Fred Cohen. Computer viruses: Theory and experiments. In *7th DoD/NBS Computer Security Conference*, pages 240–263, 1984.

[CWS] CWSandbox Webinterface. `http://www.cwsandbox.org/?page=submit`.

[Dat] DataRescue. The IDA Pro Disassembler and Debugger. `http://www.datarescue.com/idabase/`.

[Dat05] DataRescue. Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executable, 2005. `http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf`.

[DG07] Brendan Dolan-Gavitt. The VAD tree: A process-eye view of physical memory. *Digital Investigation*, Volume 4, Supplement 1:62–64, September 2007.

[Eil05] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing Inc., 2005.

[Fal07] Nicolas Falliere. Windows Anti-Debug Reference, September 2007. `http://www.securityfocus.com/infocus/1893`.

[Fer07] Peter Ferrie. Attacks On More Virtual Machine Emulators, April 2007. `http://pferrie.tripod.com/papers/attacks2.pdf`.

[Fre99] Free Software Foundation. GNU Lesser General Public License v2.1, February 1999. `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`.

[FV04] Dan Farmer and Wietse Venema. *Forensic Discovery*. Addison-Wesley, December 2004.

[Haz07] Philip Hazel. PCRE - Perl Compatible Regular Expressions, 2007. `http://www.pcre.org/`.

[HB05] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[Hon04] The Honeynet Project. *Know Your Enemy - Learning About Security Threats*. Addison-Wesley, 2nd edition edition, 2004.

[Hon07] The Honeynet Project & Research Alliance. Know your enemy: Fast-flux service networks, July 2007.

[Int06a] *Instruction Set Reference, N-Z*, volume 2B of *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, March 2006.

[Int06b] *System Programming Guide Part 1*, volume 3A of *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, March 2006.

[ISO06] ISO/IEC. *Common Criteria for Information Technology Security Evaluation. Part 1: Introduction and general model*. September 2006.

[Kle03] Tobias Klein. scoopy doo - VMware Fingerprint Suite, 2003. `http://www.trapkit.de/research/vmm/scoopydoo/index.html`.

[KPY07] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, pages 46–53, New York, NY, USA, 2007. ACM.

[KRVV04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, August 2004.

[LD03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, October 2003.

[LH07] Robert Lyda and James Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security & Privacy*, 5(2):40–45, March 2007.

[Mic06] Microsoft Corporation. *Visual Studio, Microsoft Portable Executable and Common Object File Format Specification*, May 2006.

[Mic07] Microsoft Corporation. Debugging Tools for Windows, 2007. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245. IEEE Computer Society Press, May 2007.

[mwc07a] mwcollect Alliance. Honeybow, 2007. `http://honeybow.mwcollect.org/`.

[mwc07b] mwcollect Alliance. nepenthes, 2007. `http://nepenthes.mwcollect.org/`.

[OC07] Offensive Computing: Community Malicious Code Research and Analysis, 2007. `http://www.offensivecomputing.net/`.

[OR93] J. Oikarinen and D. Reed. Internet relay chat protocol, May 1993. `http://www.ietf.org/rfc/rfc1459.txt`.

[Ore07] Oreans Technologies. Themida: Advanced windows protection system, 2007. `http://www.oreans.com/`.

[pei07] PEiD, 2007. `http://www.peid.info/`.

[Pie02a] Matt Pietrek. An in-depth look into the win32 portable executable file format. *MSDN Magazine*, February 2002.

[Pie02b] Matt Pietrek. An in-depth look into the win32 portable executable file format, part 2. *MSDN Magazine*, March 2002.

[Pin] Pin Team. Pin - A Dynamic Binary Instrumentation Tool. `http://rogue.colorado.edu/Pin/`.

[Pyt06] Python Software Foundation. Python 2.5 license, 2006. `http://www.python.org/download/releases/2.5.1/license/`.

[pyt07] The Python Programming Language, December 2007. `http://www.python.org`.

[QV07] Danny Quist and Valsmith. Covert Debugging: Circumventing Software Armoring Techniques. In *Black Hat Briefings USA 2007*, August 2007.

[RHD+06] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[Ros02] Guido van Rossum. Unifying types and classes in python 2.2, 2002. `http://www.python.org/download/releases/2.2.3/descrintro/`.

[Ros06a] Guido van Rossum. Extending and embedding the python interpreter, September 2006. `http://docs.python.org/ext/ext.html`.

[Ros06b] Guido van Rossum. Python Library Reference, September 2006. `http://docs.python.org/lib/`.

[Ros06c] Guido van Rossum. Python/C API Reference Manual, September 2006. `http://docs.python.org/api/api.html`.

[RS05] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals*. Microsoft Press, Redmond, WA, USA, 4th edition, 2005.

[Rut04] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction, November 2004. `http://www.invisiblethings.org/papers/redpill.html`.

[Sch01] Sven B. Schreiber. *Undocumented Windows 2000 Secrets - A Programmer's Cookbook*. Addison-Wesley Professional, May 2001.

[Sch07] Jörg Schilling. Cdrtools, 2007. `http://cdrecord.berlios.de/private/cdrecord.html`.

[SF01] Péter Ször and Peter Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, September 2001.

[Sou07] Sourcefire, Inc. Clamav, 2007. `http://www.clamav.net/`.

[sql07a] SQLite, 2007. `http://www.sqlite.org`.

[sql07b] SQLite Copyright, 2007. `http://www.sqlite.org/copyright.html`.

[Sym07] Symantec Corporation. Symantec internet security threat report: Trends for january-june 07, September 2007. `http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_internet_security_threat_report_xii_09_2007.en-us.pdf`.

[SZ03] Ed Skoudis and Lenny Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, November 2003.

[Szö05] Péter Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, second edition edition, 2001.

[VT] VirusTotal - Free Online Virus and Malware Scan. `http://www.virustotal.com/`.

[WB07] Stefan Weyergraf and Sebastian Biallas. HT Editor, 2007. `http://hte.sourceforge.net/`.

[wge] Gnu wget. `http://www.gnu.org/software/wget/wget.html`.

[Wil06] Carsten Willems. Automatic Behavior Analysis of Malware. Diploma thesis, RWTH Aachen, July 2006.

[Yas07] Mark Vincent Yason. The Art of Unpacking. In *Black Hat Briefings USA 2007*, August 2007.

[Yus] Oleh Yuschuk. OllyDbg. `http://www.ollydbg.de/`.

[ZHH+07] Jianwei Zhuge, Thorsten Holz, Xinhui Han, Chengyu Song, and Wei Zou. Collecting Autonomous Spreading Malware Using High-Interaction Honeypots. In *Proceedings of 9th International Conference on Information and Communications Security (ICICS'07)*, December 2007.