Final thesis

Visualizing Endpoint Security Technologies using Attack Trees

by

Stefan Pettersson

LIU-IDA/LITH-EX-A--08/031--SE

2008-06-16
Final thesis

Visualizing Endpoint Security Technologies using Attack Trees

by

Stefan Pettersson

LIU-IDA/LITH-EX-A--08/031--SE

2008-06-16
Final thesis

Visualizing Endpoint Security Technologies using Attack Trees

by

Stefan Pettersson

LIU-IDA/LITH-EX-A--08/031--SE

Supervisor : David Byers
Dept. of Computer and Information Science
at Linköpings universitet

Examiner : Prof. Nahid Shahmehri
Dept. of Computer and Information Science
at Linköpings universitet
Abstract

Software vulnerabilities in programs and malware deployments have been increasing almost every year since we started measuring them. Information about how to program securely, how malware shall be avoided and technological countermeasures for this are more available than ever. Still, the trend seems to favor the attacker. This thesis tries to visualize the effects of a selection of technological countermeasures that have been proposed by researchers. These countermeasures: non-executable memory, address randomization, system call interception and file integrity monitoring are described along with the attacks they are designed to defend against. The coverage of each countermeasure is then visualized with the help of attack trees. Attack trees are normally used for describing how systems can be attacked but here they instead serve the purpose of showing where in an attack a countermeasure takes effect. Using attack trees for this highlights a couple of important aspects of a security mechanism, such as how early in an attack it is effective and which variants of an attack it potentially defends against. This is done by the use of what we call defensive codes that describe how a defense mechanism counters a sub-goal in an attack. Unfortunately the whole process is not well formalized and depends on many uncertain factors.

Keywords: endpoint security, attack tree, memory corruption, non-executable memory, address randomization, system call interception
Acknowledgements

I would like to thank my examiner Nahid Shahmehri for giving me great freedom during my thesis work and also my supervisor David Byers for providing very valuable input on the report and catching many things that I did not. I would also like to thank my colleagues at High Performance Systems for giving me inspiration and support during the course of the thesis, especially Marianne for providing with the time needed and C-J for the expertise.

Finally, I would like to thank my fiancé Josefine for putting up with me and Bram Moolenaar as well as Leslie Lamport for creating Vim and \LaTeX, respectively.
Contents

List of Figures ix

List of Tables x

1 Introduction 1
   1.1 Background and motivation 1
   1.2 Goal 2
   1.3 Scope 3
   1.4 Intended audience 3
   1.5 Contribution 3
   1.6 Related work 4
   1.7 Conventions 4
   1.8 Document structure 4

2 Foundations 5
   2.1 The triad 5
   2.2 Detection and prevention 5
      2.2.1 Anomalies and misuse 6
   2.3 Memory during program execution 7

3 Offensive 13
   3.1 Stack-based buffer overflow attacks 13
      3.1.1 Arbitrary code execution 14
      3.1.2 Injecting code 16
      3.1.3 Exploiting a vulnerability 16
   3.2 Buffer overflows in other memory regions 18
      3.2.1 Heap-based buffer overflows 18
      3.2.2 .bss-based buffer overflows 19
   3.3 Format string attacks 20
   3.4 Specific memory attacks 21
      3.4.1 Overwriting important data 22
      3.4.2 Overwriting function pointers 22
      3.4.3 Overwriting code pointers 22
      3.4.4 Reusing existing code 23
   3.5 Integer problems 24
   3.6 Race conditions 25
   3.7 Injection attacks 27
      3.7.1 Command injection 27
# CONTENTS

3.7.2 Directory traversal .......................... 28

4 Defensive ........................................ 31
   4.1 Non-executable memory ....................... 31
      4.1.1 $W \oplus X$ .............................. 31
      4.1.2 Software emulated ....................... 32
      4.1.3 Hardware-assisted ....................... 33
   4.2 Address randomization ........................ 33
      4.2.1 What to randomize ....................... 34
      4.2.2 When to randomize ....................... 35
   4.3 System call interception ........................ 35
      4.3.1 Operating system calls ................... 36
      4.3.2 System call monitoring ................... 37
      4.3.3 Monitoring syscall sequences .............. 37
      4.3.4 Mimicry attacks .......................... 37
      4.3.5 Other monitoring techniques ............... 38
      4.3.6 Concurrency vulnerabilities ............... 39
   4.4 File integrity monitoring ..................... 40
      4.4.1 Detecting changes ....................... 41
      4.4.2 Trust issues ............................. 41

5 Analysis ........................................ 43
   5.1 Defensive technologies ........................ 43
      5.1.1 Non-executable memory ................... 43
      5.1.2 Address randomization .................... 44
      5.1.3 System call interception ................... 45
      5.1.4 File integrity monitoring ................. 45
   5.2 Attack trees .................................. 46
   5.3 Visualizing defensive techniques ............... 47
      5.3.1 Attack tree ................................ 47
      5.3.2 Defensive codes ........................... 48
      5.3.3 Analysis ..................................... 50

6 Conclusions .................................... 53
   6.1 Experiences ................................... 53
      6.1.1 On uncertainty ............................ 53
      6.1.2 Pros and cons ............................. 54
   6.2 Further work ................................ 55

Bibliography .................................... 56
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Examples of programming languages</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>A series of pop and push instructions on a stack</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>A stack frame</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Code for demonstration of stack during function calls</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Call stacks for breakpoints</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>Before and after a local variable is pushed</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of stack with char buffer</td>
<td>10</td>
</tr>
<tr>
<td>2.8</td>
<td>The <code>.text</code>, <code>.data</code>, <code>.bss</code>, heap and stack segments</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>Example of where data are stored in memory</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>A function vulnerable to a stack-based buffer overflow</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Different results depending on input length</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Overwriting the return pointer</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>Example shellcode</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Shellcode padded with a NOP sled</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>Vulnerable code</td>
<td>16</td>
</tr>
<tr>
<td>3.7</td>
<td>Call stack after code injection</td>
<td>17</td>
</tr>
<tr>
<td>3.8</td>
<td>Simple example of a heap-based overflow</td>
<td>18</td>
</tr>
<tr>
<td>3.9</td>
<td>Simple example of a <code>.bss</code>-based overflow</td>
<td>19</td>
</tr>
<tr>
<td>3.10</td>
<td>Code with format string vulnerability</td>
<td>21</td>
</tr>
<tr>
<td>3.11</td>
<td>Variable ordering on the stack</td>
<td>22</td>
</tr>
<tr>
<td>3.12</td>
<td>Vulnerable authentication routine</td>
<td>23</td>
</tr>
<tr>
<td>3.13</td>
<td>Stack during return-to-libc</td>
<td>24</td>
</tr>
<tr>
<td>3.14</td>
<td>Wrapper program vulnerable to command injection</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Different segment limits</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Metasploit showing different targets</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Protection rings as provided by modern CPUs</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Assembly to make an <code>exit</code> syscall</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Sampled syscall sequences</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Mimicry calls</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Attack tree</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Attack tree with defensive codes</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Vulnerabilities cataloged by CERT. .......................... 1
1.2 Vulnerabilities cataloged by US-CERT. ...................... 1
5.1 Defensive codes. .............................................. 48
<table>
<thead>
<tr>
<th>Abbreviations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DEP</td>
<td>Data Execution Prevention</td>
</tr>
<tr>
<td>EBP</td>
<td>Extended Base Pointer</td>
</tr>
<tr>
<td>EIP</td>
<td>Extended Instruction Pointer</td>
</tr>
<tr>
<td>ESP</td>
<td>Extended Stack Pointer</td>
</tr>
<tr>
<td>GOT</td>
<td>Global Offset Table</td>
</tr>
<tr>
<td>HIDS</td>
<td>Host Intrusion Detection System</td>
</tr>
<tr>
<td>HIPS</td>
<td>Host Intrusion Prevention System</td>
</tr>
<tr>
<td>ICD</td>
<td>Idealized Character Distribution</td>
</tr>
<tr>
<td>IDS</td>
<td>Intrusion Detection System</td>
</tr>
<tr>
<td>IPS</td>
<td>Intrusion Prevention System</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last-In-First-Out</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NIDS</td>
<td>Network Intrusion Detection System</td>
</tr>
<tr>
<td>NIPS</td>
<td>Network Intrusion Prevention System</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>NVD</td>
<td>National Vulnerability Database</td>
</tr>
<tr>
<td>NX</td>
<td>No eXecute</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>TOCTOU</td>
<td>Time Of Check – Time Of Use</td>
</tr>
<tr>
<td>XD</td>
<td>eXecute Disable</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter will explain why this topic was chosen, set the scope and give an overview of the layout of the rest of the document.

1.1 Background and motivation

It is possible to specify a program design as a set of desired functions and then write code that performs those functions. If anything is missing or not working properly, it is considered a bug and is generally fixed. Security issues tend to appear when (1) the aforementioned function appears to work properly when it actually does not, and (2) when functionality is (unintentionally) included in the program but not in the original design specification, i.e. unknown, unintended functionality.

Security issues have been on the increase in recent years and many factors are involved. More computers and thus software is being used, programs increase in size and complexity and the security industry is getting larger which means more people are involved in finding issues.

The results of this can easily be seen in statistics on the number of reported vulnerabilities. See table 1.1 [1] and 1.2 [2] for vulnerabilities reported to CERT at Carnegie Mellon university and US-CERT National Vulnerability Database (NVD), respectively.

<table>
<thead>
<tr>
<th>Year</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td># of vulns</td>
<td>2,437</td>
<td>4,129</td>
<td>3,784</td>
<td>3,780</td>
<td>5,990</td>
<td>8,064</td>
<td>7,236</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>2001</th>
<th>2002</th>
<th>2003</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td># of vulns</td>
<td>1,677</td>
<td>2,164</td>
<td>1,504</td>
<td>2,456</td>
<td>4,927</td>
<td>6,602</td>
<td>6,689</td>
</tr>
</tbody>
</table>
These figures represent vulnerabilities in general. Another kind of security issue is that of malware, programs designed with malicious intent. Traditionally these have consisted of viruses, logic bombs and trojans, among others [3]. During the turn of the millennium, self-replicating worms like Melissa, ILOVEYOU, Code Red and Nimda became abundant on the Microsoft Windows platform. While a new major worm-outbreak like these might be unlikely, malware seems to be as plentiful as ever and their infection vectors\textsuperscript{1} change continuously. Commercial anti-virus firm F-Secure reported a near 100% increase in malware detection signatures for year 2007, in early December. The number of malware signatures developed in 2007 alone were thus estimated to be equal to the amount produced during the previous twenty years [4]. This has later been shown to be an accurate estimation [5].

The technologies presented in this dissertation are aimed at individual computer systems as opposed to computer networks. The security of individual computers is interesting due to the rise of botnets. These are networks of compromised computers with some kind of remote control software installed that is generally used for malicious purposes. The computers participating in the botnet are popularly known as zombies and are compromised and added to the net in a wide variety of ways. Server-side vulnerabilities as exploited by the Windows worms in the beginning of the century have decreased rapidly, new techniques are client-side bugs in web browsers, mail clients and instant messengers as well as plain social engineering\textsuperscript{2} with the help of trojans.

Botnets may serve many different purposes for its owner, with sending spam, hosting phishing or scam sites, executing denial of service attacks and distributing keystroke loggers or similar malware, being the most prevalent. These botnets have also been found to be very sizable, some stretching into hundreds of thousands of zombie computers.

While the amount of vulnerabilities is increasing, so is research on how to prevent them or make them difficult to exploit. Many security features have been implemented in operating systems and as third-party products. This report is about the fundamental technologies used in some of these products.

\section*{1.2 Goal}

The research conducted regarding intrusion detection/prevention is ample and widespread. This thesis tries to compile, explain and compare a selection of well-studied technologies according to a set of criteria. It will try to answer the following questions regarding each technology:

1. How does it work?

2. What does it defend against?

3. Can it be bypassed/attacked?

\footnote{The way a computer gets infected by malware, e.g. via email or through a website. The expression \textit{attack vector} is used in the same way.}

\footnote{Social engineering is the act of manipulating people into performing actions or revealing information.}
Significant effort is put into explaining, and in some cases, demonstrating the exploitation of vulnerabilities. This is considered very important in understanding how the vulnerabilities are to be prevented or mitigated.

The defense technologies presented will not protect against all attacks individually. Larger coverage might however be possible by combining different ones, thus the most important question might be how they can be successfully combined. *Can the coverage of these technologies be visualized with the help of attack trees?*

### 1.3 Scope

The defense technologies discussed in this thesis have been chosen so that access to a program’s source code is not a strict requirement. Consequently, they are aimed not at developers but at users. An exception to this criterion is source code for the operating system kernel. Also, for the sake of discussion; the section on address space randomization touches on the possibilities that arise when program source code is available. The technologies chosen for study are as follows:

- **Memory level** defenses mainly include address space randomization and non-executable memory segments.

- **Operating system level** is mainly concerned with operating system call interception.

- **File system level** covers file integrity monitoring.

### 1.4 Intended audience

The intended audience has basic experience in some high-level, C-like programming language like C, C++ or Java. This is mainly required for the section on exploitation of memory level vulnerabilities. Furthermore, interest in computers and computer security in general will certainly help. It should not be looked upon as a strict requirement though since the document will try to explain difficult topics.

### 1.5 Contribution

The thesis presents what seems to be a new use for attack trees. Instead of constructing a tree that describes the ways to attack a specific system, a general tree is made for an imaginary system. The attack tree is then used to visualize where in an attack certain security countermeasures have effect. Hopefully this will demonstrate the coverage of countermeasures so that intelligent decisions regarding which ones to implement can be done. To the author’s best knowledge this usage of attack trees is novel.

---

3 Attack trees are covered in section 5.2 on page 46 and are used to describe how a system can be attacked.
1.6 Related work

Compilations of attacks against computer programs and corresponding countermeasures exists by the dozen. Some of these are *Generic Anti-Exploitation Technology for Windows* by eEye [6], *Security Intrusions and Intrusion Prevention* by John Wilander [7], *Penetration Testing: Offensive Security Testing* by Carl-Johan Bostorp [8] and *Buffer Overflow and Format String Overflow Vulnerabilities* by Kyung-Suk Lhee and Steve J. Chapin [9].

1.7 Conventions

All code examples are written in C. Call stacks are positioned with lower addresses facing upwards. Program code is written in monospace text, have line numbers and are delimited above and below by horizontal lines. Values in base 16 (hexadecimal) are written with the prefix “0x” and when referring to memory addresses they are also typeset as monospace. Assembly instructions are written in AT&T format and raw byte code is written like plain strings (04b8) or escaped like in C code (\x04\xb8).

1.8 Document structure

First in chapter 2 some foundations are laid about computer security in general along with a description of how a process in UNIX is handled in memory. This to prepare for chapter 3 where a few attack techniques are described and discussed thoroughly. Then chapter 4 surveys selected defensive measures that try to prevent or mitigate the effects of previously discussed attacks. These defensive measures are then analyzed and incorporated in an attack tree in chapter 5. Chapter 6 concludes.
Chapter 2

Foundations

2.1 The triad

Computer security is a tremendously large subject and therefore difficult to capture concisely. One popular method lists three major aspects in which the security of a system can be compromised. These are the definitions as given by Gollman [3]:

- **Confidentiality** is the prevention of unauthorized disclosure of information.
- **Integrity** is the prevention of unauthorized modification of information.
- **Availability** is the prevention of unauthorized withholding of information or access.

These points capture a large number of, but arguably not all, security issues. When sensitive files are stolen from a file server, it might be considered an attack against confidentiality. If the files are not stolen but changed in any way, their integrity is compromised. Finally if they are deleted, it is an attack on the files’ availability.

The power of the CIA triad (as it is popularly called) is its breadth. Confidentiality can be breached if a cryptanalyst manages to break an encryption scheme just as much as if he or she simply peeks over your shoulder when you type in your password. The integrity of information can be violated by someone slipping on the keyboard and by malicious programs. The files deleted in the previous paragraph is an attack on availability just like pulling the plug to the server or overwhelming it with requests so that its network connection crawls to a halt.

Some of these proposed breaches could be malicious attacks and some of them might be accidental. This is an important distinction to make when talking about risks and threats. We will not deal with those specific issues in this thesis.

2.2 Detection and prevention

Intrusion detection is a broad term and captures several different ways of detecting intruders. Programs that perform such detection are called intrusion
detection systems (IDS) and have been around for decades. It was suggested by James P. Anderson in 1980 [10] that audit records could be used to discover intruders in a system.

When referring to an IDS one usually means a system that detects attacks in a network, such a system is more accurately called a network IDS (NIDS). This is usually deployed as a computer connected to the network that is fed with all the traffic to monitor. The computer runs software that analyzes the traffic stream and raises some kind of alarm when an intrusion has been detected so that the administrator can respond to it.

If an IDS only concerns itself with the host it sits on it is called a host IDS (HIDS). One kind of HIDS technology that monitors the computer’s file system will be treated in section 4.4.

An intrusion prevention system (IPS) is the result of an IDS that is instructed on not only how to detect the intrusion but to stop it as well. These also exist as network IPS (NIPS) and host IPS (HIPS).

The design of various detection and prevention systems varies greatly. The SANS Institute provides a comprehensive list of frequently asked questions (FAQ) on intrusion detection [11]. The US-based National Institute of Standards and Technology (NIST) recently released a guide to intrusion detection and prevention systems [12]. Another valuable resource is the seminal paper by Newsham and Ptacek [13] which demonstrates fundamental problems with intrusion detection on the network level.

2.2.1 Anomalies and misuse

Intrusion detection can, regardless of the technology used or at which layer, be done in two major ways: by detecting misuse or by detecting anomalies [3]. *Misuse detection* works by looking for actions and data defined as malicious, also known as attack signatures. *Anomaly detection*, on the other hand, defines good behavior and looks for statistical deviations from this baseline. There are several difficulties associated with each of these approaches.

Developing attack signatures is generally easy since we know what we are looking for. However, malicious behavior is a moving target and if an attacker is aware of a specific signature it is often possible to modify the attack to not match that signature [14]. In addition, the overwhelming drawback of attack signatures is that the attack must be known before it can be detected. Thus, unknown attacks cannot, by definition, be detected.

The difficulty with anomaly detection is the accurate modeling of a system’s acceptable behavior. This issue gets even more challenging considering proprietary applications where access to documentation and source code is limited. In contrast to misuse detection, if an attack constitutes an anomaly, anomaly detection has the potential to detect novel attacks. At the same time it is important to note that an attack does not have to be an anomaly and an anomaly not an attack.

The majority of today’s network intrusion detection systems (NIDS) are built as misuse detectors [3].
2.3 Memory during program execution

The description of memory handling given below mainly applies to UNIX-like operating systems. However, other operating systems work basically the same way.

Computer programs are essentially a long series of instructions for the computer to perform. In a high-level programming language like C or C++ there are, for example, instructions to perform complicated arithmetic, concatenate strings and handle data structures like arrays or matrices. These instructions are then translated by a compiler into smaller ones known as machine instructions that the central processing unit (CPU) can understand. Machine instructions have very basic functionality such as add, subtract and move numbers, check for rudimentary conditions and jump to other code sections. Obviously, one high-level instruction like computing the square root of a number will require several machine instructions. Taking this further, simply moving a window on your desktop requires thousands of machine instructions.

printf("Hello world!");

0x08048074: mov $0x4,%eax
0x08048079: mov $0x1,%ebx
0x0804807e: mov $0x8049098,%ecx
0x08048083: mov $0xd,%edx
0x08048088: int $0x80

04b8 0000 bb00 0001 0000 98b9
0490 ba08 000d 0000 80cd

Figure 2.1: High-level language, assembly language and machine instructions shown in hexadecimal.

Since humans find it difficult to memorize numbers (machine instructions are numeric), mnemonic codes are used as reference and the resulting code is called assembly language. The three code snippets in figure 2.1 (high-level language, assembly and machine instructions, respectively) are roughly equivalent.

The assembly in figure 2.1 also shows that every instruction has an address (shown to the left in the figure, prefixed with 0x) which can be used to reference that particular instruction. Addresses are also used by the CPU to know what the next instruction is. While an instruction is being performed, the address of the next one is kept in a small memory unit in the processor called a register. There are several registers in a modern processor; the one that holds the address of the next instruction is called the extended instruction pointer (EIP) in Intel x86 architectures, or more generally, the program counter (PC).

A major feature of high-level languages is the notion of functions. Most programs have certain functionality that needs to be performed several times, such a series of instructions can be put into a function.

Functions can be viewed as black boxes; you provide some input and get a certain output. The function input, processing done and output can be completely arbitrary. The important part is that a function is viewed as a series of instructions that are stored at a particular place in memory and called when needed.
The use of functions is made possible by (1) being able to jump to a different place in code (referenced by address) to continue execution from there and (2) a data structure known as the stack.

(1) There are several instructions that can be used to jump (also known as branch) to different addresses in code. The most common ones have mnemonic codes like `JMP` and `CALL`.

The effect of jumps is that code is not necessarily executed in a linear fashion; it will branch out in functions calling other functions, each at different addresses. Due to this branching it is crucial to remember from where each function was called so that execution can return from where it went off. This can be accomplished through the use of a call stack.

![Figure 2.2: A series of pop and push instructions on a stack.](image)

(2) The stack is what is known as a last-in-first-out (LIFO) queue and can be visualized as a stack of plates. Plates can be put onto or removed from the top of the stack. To be able to remove a plate in the middle of the stack, the ones sitting on top of it have to be removed first. These two operations; to put a plate on top of the stack and to remove the plate from the top is called to push and pop, respectively. To clarify, we push objects onto a stack and pop objects off of a stack. See figure 2.2 for an example. To control the stack, two registers (just like the PC) are used; the stack pointer (SP) and base pointer (BP). On Intel x86 architectures they are referred to as extended stack pointer (ESP) and extended base pointer (EBP). The stack pointer contains the address of (points to) the top of the stack so that we know where to push or pop items. The base pointer points to a fixed location of the current stack frame, which will be explained shortly. In practice, the stack is simply a piece of memory that the CPU uses to store things it needs to remember for a while.

![Figure 2.3: A stack frame.](image)

When a function is called, a few things happen; first (1) the input to the function (known as parameters) is pushed onto the stack. Then (2) the current EIP (holding the address of the next instruction) is pushed onto the stack and (3) the EBP is pushed as well. Finally (4) space is allocated on the stack for any local variables used inside the function. The data in (1) through (4) is known as...
as the stack frame. See figure 2.3. A stack structure used in this way to control function calls is known as a call stack.

After creating the stack frame (and saving previous values), the processor’s EIP, ESP and EBP registers are updated. The EIP now points to the first instruction in the called function and the address in ESP has been changed to point to the top of the current stack frame. The EBP points at a fixed location in the current stack frame, serving as a base from where parameters and variables are referenced via positive and negative offsets. The reason for doing (2) and (3) is to be able to update these registers and later retrieve the old values when the function is finished, thus, “something the CPU needs to remember for a while”, as stated earlier.

When all instructions inside the function have been processed the CPU is ready to return from where it left off. The operations done at the beginning of the function call are now made in reverse. First any local variables are removed from the stack, then the old base pointer is popped so that the location of the previous stack frame is recovered. The return address is then popped into the program counter so that execution can continue from where it left off before the function call. Finally, parameters that were used as input to the function are removed from the stack.

```
1 int doubler(int param) {
2   int result = param * 2;
3   /* breakpoint (b) */
4   return 0;
5 }
6
7 int main(void) {
8   int input = 42;
9   /* breakpoint (a) */
10  doubler(input);  // function call
11  /* breakpoint (c) */
12  return 0;
13 }
```

Figure 2.4: Code for demonstration of stack during function calls.

Consider the code snippet in figure 2.4. When the main() function is about to call doubler() at line 9 the stack frame of main() looks like figure 2.5(a). When doubler() is called and the breakpoint at line 3 is reached, the parameter param, the return address (the line below the function call) and old base pointer has been pushed onto the stack. Also, space for the local variable result has been allocated and the resulting value has been stored there. See figure 2.5(b). doubler() then returns back to main() and pops the old base pointer and return address to continue execution at the breakpoint on line 11. The call stack (figure 2.5(c)) now looks like it did before the call to doubler(). The small gap between the two stack frames only serves to illustrate that stack frames are not always adjacent.

Remember that doubler() could have called another function, resulting in a third stack frame on top of doubler()'s. The handling of the call stack is

---

1Being local variables, they are not used outside the function and are therefore removed.
result = 84
old base pointer
return address
param = 42

input = 42
old base pointer
return address
(a)

input = 42
old base pointer
return address
(b)

input = 42
old base pointer
return address
(c)

Figure 2.5: Call stacks for breakpoints in figure 2.4.

<table>
<thead>
<tr>
<th>address</th>
<th>value 1</th>
<th>value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2c</td>
<td>0x2c</td>
<td>local variable</td>
</tr>
<tr>
<td>0x30</td>
<td>old base pointer</td>
<td>0x30</td>
</tr>
<tr>
<td>0x34</td>
<td>return address</td>
<td>0x34</td>
</tr>
<tr>
<td>0x38</td>
<td>parameters</td>
<td>0x38</td>
</tr>
</tbody>
</table>

Figure 2.6: Before and after a local variable is pushed.

sometimes referred to as the flow of control; the return address decides where execution is to continue when the current function is finished.

An analogy of this procedure is Alice (the CPU) who is sitting in front of her desk reading a manual (running a program). Suddenly she finds herself in the need to get more coffee (a function call). The thing is, Alice tends to forget what she is doing when she returns so she jots down “read the manual” (the return address) on a sticky note (the stack) and heads for the coffee machine. On returning to her desk she checks the note (pops the return address into EIP) and continues on reading.

There are two more crucial details regarding the stack. (1) On Intel x86 architectures (which is by far the most common) the stack is said to “grow down toward lower memory addresses”. Essentially, it is positioned upside down with the bottom at a higher address than the top. When an item is pushed onto the stack its address is lower than the other items. See figure 2.6.

(2) Text strings like “Hello world!” in figure 2.1 above are stored as byte arrays (also known as buffers) in memory, one byte per character. The important thing is that, in contrast to the stack, the byte array grows toward higher

<table>
<thead>
<tr>
<th>address</th>
<th>value 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>HELL</td>
</tr>
<tr>
<td>0x44</td>
<td>0 W0</td>
</tr>
<tr>
<td>0x48</td>
<td>RLD\0</td>
</tr>
<tr>
<td>0x4c</td>
<td>old base pointer</td>
</tr>
<tr>
<td>0x50</td>
<td>return address</td>
</tr>
<tr>
<td>0x54</td>
<td>parameters</td>
</tr>
</tbody>
</table>

Figure 2.7: Example of stack with char buffer.
addresses. The last character in a string has a higher address than the preceding ones. See figure 2.7 where a buffer containing “HELLO WORLD” is stored as a local variable in a stack frame. The \0 at the end is a null byte (all zero byte) to signal the end of the string. Strings that are ended with a null are called null-terminated. (1) and (2) will turn out to be very important when discussing buffer overflow attacks later.

lower addresses

| .text |
| .data |
| .bss |

heap grows toward higher addresses

higher addresses

| stack |

grows toward lower addresses

Figure 2.8: The .text, .data, .bss, heap and stack segments in process memory.

```c
1 int foo = 56; // global init: .data
2 int bar; // global uninit: .bss
3
4 int main(void) {
5    int egg = 42; // local init: stack
6    double spam; // local init: stack
7    char parrot[] = "stone dead"; // local init: stack
8    int *witch = malloc(2 * sizeof(int)); // dynamic: heap
9    static char brian[4]; // global uninit: .bss
10   return 0;
11 }
```

Figure 2.9: Example of where data are stored in memory.

With the, for our purposes, most important part of program memory introduced, we can put it into perspective. If the memory of one process (program) in a UNIX-like environment is viewed as a contiguous block it can be split up as in figure 2.8. The .text segment holds the executable code, just like the machine code in figure 2.1. The .data and .bss segments hold data that are initialized and uninitialized, respectively. The important difference is that since .data is already initialized at compile time its size is fixed whereas the size of .bss is alterable during runtime. Both of these segments are writable in contrast to .text which is read-only. Lastly, the heap is used for dynamic (runtime) general purpose memory allocation. Note that the heap grows towards higher addresses while the stack grows towards lower addresses. See figure 2.9 for example code of where data is stored.

This is the foundation needed to understand the basics of the attacks against a process’ memory that will be covered in the next chapter.
Chapter 3

Offensive

In order to understand the defensive countermeasures presented in the next chapter the attacks the defend against must be well understood. This chapter discusses a selection of common attacks against computer programs.

It is important to read this part carefully in order to understand how to defend against these kinds of attacks. The previous chapter and some experience of high-level programming language is assumed.

3.1 Stack-based buffer overflow attacks

Remember Alice from the previous section? What if, when she returns from the coffee machine, the sticky note says something completely different, like “Give Mallory all your life savings then throw yourself out the window”? Alice would have to be very liberal in her trust of her own note’s integrity to fall for that one. Computers are.

The story of Alice above is an example of overwriting the return address on the call stack in order to hijack the flow of control in execution. Doing this by overflowing buffers stored on the stack was made famous by Aleph One in 1996 [15]. Overflowing buffers is possible due to lack of length checking before writing a string to memory. The fact that the call stack grows towards lower memory addresses while buffers grow in the opposite direction as shown in figure 2.7 will become very important.

Consider the code in figure 3.1; three local variables are declared, two integers and an eight byte long character buffer. The function strcpy() writes the contents of str to the buffer buf until it finds a null character which terminates the string. When /* breakpoint */ on line 5 is reached, just before str is written to buf, the stack will look like figure 3.2(a). Now, if str is eight characters or less, it will fit in the memory allocated for buf. So, if the string is “PENGUIN” (which with the null byte is eight bytes long) the stack will look like 3.2(b) upon return on line 7. Instead, suppose “AAAAAAAAAAAAAAAAAAAAAAAA” (23 A’s) is given as input. strcpy() will continue writing the string until it hits the null byte, effectively writing 24 bytes to a buffer that only has room for eight. As can be seen in figure 3.2(c) the results are devastating: the whole stack frame is overwritten with A’s. This will of course become a problem when the CPU tries to pop the return address it pushed earlier when entering vulnerable()...
and continue execution from there.

1 int vulnerable(char *str) {
2   int num1 = 13;
3   char buf[8];
4   int num2 = 14;
5   /* breakpoint */
6   strcpy(buf, str);
7   return 0;
8 }

Figure 3.1: A function vulnerable to a stack-based buffer overflow.

<table>
<thead>
<tr>
<th>num2 = 14</th>
<th>num2 = 14</th>
<th>num2 = 14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PENG</td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>UIN\0</td>
<td>AAAA</td>
</tr>
<tr>
<td>num1 = 13</td>
<td>num1 = 13</td>
<td>AAAA</td>
</tr>
<tr>
<td>old base pointer</td>
<td>old base pointer</td>
<td>AAAA</td>
</tr>
<tr>
<td>return address</td>
<td>return address</td>
<td>AAAA</td>
</tr>
<tr>
<td>address of str</td>
<td>address of str</td>
<td>AAA\0</td>
</tr>
<tr>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
</tr>
</tbody>
</table>

Figure 3.2: Different results depending on input length.

The hexadecimal value corresponding to the character “A” is 0x41, thus the processor will try to execute code at address 0x41414141. This, as far as the processor knows, is the return address it saved on the stack earlier. Whatever exists at that address, if anything, it is highly unlikely to be valid code. This will result in the program crashing due to a segmentation violation (also known as segmentation fault). This happens when a program tries to read or write to memory it does not have access to.

Instead suppose there is executable code at the address 0xbffff37c. By studying how the stack frame is overwritten in figure 3.2 we decide to instead input the string AAAAAAAAAAAAAAAAAAA\x7c\xf3\xff\xbfAAA This will result in a the stack frame looking like figure 3.3, forcing the processor to continue execution at 0xbffff37c. The flow of control has been hijacked.

3.1.1 Arbitrary code execution

So, it is now possible to direct the CPU to an arbitrary address for further execution of code. The most important question now is not what code to execute but where it is located. We have to know the exact address since even a small variation will likely result in a segmentation violation and crash. Fortunately for the attacker, these addresses are determined at compile-time\(^1\) of the program

---

\(^1\)This is not strictly true, there are certain aspects of program execution that modify the process memory slightly. Ignore this for now.
in question and can be found with the help of a few programming tools. Since copies of the same executable file are often distributed with the same version of operating system, the addresses will be the same on several sites.

```
x31\xc0\xb0\x46\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88
\x43\x07\x89\x5b\x08\x89\x43\x0c\xbb\x0b\x8d\x4b\x08\x8d\x53\x0c
\xcd\x80\xe8\xe5\xff\xff\xff\xff\x2f\x62\x69\x2f\x68
```

Figure 3.4: Example shellcode from [16].

Before continuing with the important question of where, we will cover the code execution itself. Attacks like this one, overwriting the return address, are among the ones often referred to as to enable “arbitrary code execution” in security advisories. The reason for this is that the attacker often is able to inject her own code into the memory space of the vulnerable process and direct execution to it. Injected code consists of a set of machine instructions like in figure 3.4. The code is interpreted in the same way as the byte instructions in figure 2.1 earlier but are here given in the format of a string, therefore bytes are escaped with \x.

Code such as that in figure 3.4 usually executes a command shell from where the attacker can execute further commands. It is usually called shellcode regardless of whether it executes a command shell or not. The process of writing shellcode is quite involved but since examples are readily available for a multitude of platforms and purposes, this generally does not matter for less experienced attackers. It is entirely possible to keep a library of shellcode and just pick an appropriate one when needed.

An attacker can put code into a computer’s memory in several different ways. Regardless of where, the exact address has to be known and despite checking a copy of the same executable file there are many aspects that can affect the memory layout. Among these are environment variables used and even the length of the executable’s file name. Hitting the correct address on the first try, which is often required, in a 32 bit address space, with more than four billion addresses to choose from, is tremendously difficult. To alleviate this problem, a long string of one-byte NOP instructions can be used. Most processor architectures define a “no operation” machine instruction. This instruction has the value 0x90 on Intel x86 platforms and executing it has no effect besides incrementing the EIP, thus continuing with the next instruction. Now, if the attacker is able to inject
a large number of NOP instructions immediately followed by the shellcode, the
effective target area can be increased. After guessing an address holding one
of the NOP instructions, execution will continue from there and “slide” all
the way into the shellcode. Because of this behavior, a string of NOP instructions
used in this way is popularly known as a **NOP sled**. See figure 3.5 where the
shellcode has been prepended with a NOP sled.

### 3.1.2 Injecting code

Before going further with the problem of finding an address containing one of
the instructions in the NOP sled, we must inject it into the process’ memory.
If the buffer used for overwriting the return address is large enough to hold the
shellcode along with a sufficiently large NOP sled, that would be the natural
choice. That way, code can be injected, return address overwritten and execution
diverted, in one go. As for “large enough”, the shellcode in figure 3.4 is only 46
bytes long and there are even smaller ones around. So, in theory, any buffer
larger than this would suffice.

### 3.1.3 Exploiting a vulnerability

To show how these bits fit together we demonstrate it against the vulnerable
example program in figure 3.6. The target buffer is 512 bytes long. Since that
is the only local variable, the return pointer should be around 512 bytes from
the buffer’s starting address. This can easily be checked for by giving an 530
byte input and watching the program crash due to a segmentation violation.

```c
1 int main(int argc, char *argv[]) {
2    char buffer[512];
3    strcpy(buffer, argv[1]);
4    return;
5 }
```

**Figure 3.6:** Vulnerable code.

Now, the return address needs to be overwritten with an address pointing
to a NOP sled leading to shellcode injected into the process’ memory space.
Fortunately, we already discussed how this can be done in one step by filling the
buffer with the shellcode. Suppose we find out that the buffer starts at address 0xbfffef0c in the stack. Then the following string has potential.

\[402 \text{ NOP instructions} + \text{shellcode} + 50 \text{ return addresses}\]

With NOP instructions being one byte long, the shellcode 46 bytes and an address four bytes we get a string of length

\[(402 \times 1) + 46 + (50 \times 4) = 648 \text{ bytes}\].

One of the fifty addresses at the end are very likely to overwrite the original return address. They will also overwrite some data beyond the return address but this does not affect the exploitation. The resulting stack frame will look like figure 3.7. This return address was chosen to be slightly higher than than the buffer’s start address to ensure that it points to the NOP sled. The reason for using exactly 402 NOP instructions is to make the length of the NOP sled and shellcode divisible by four. This is necessary for the return addresses to be properly aligned in memory which is a requirement on Intel x86 platforms.

Now the program in figure 3.6 is run with the above NOP padded shellcode and appropriate return address as input. When copying the input into buffer, the return address saved on the stack will be overwritten with an address pointing to the NOP sled. When returning from the function, execution will slide down (on) the sled and eventually hit the shellcode that gets executed.

The command line session of the exploitation is given below. vuln is the name of the vulnerable program and the parameters following it (the ones prepended with dollar signs) expands into the above mentioned input string that overflows the buffer. The shell prompt sh-3.1# signifies that root status has been gained, i.e. the exploit was successful and the vulnerable program ran with high privileges.

```
sh-3.1# whoami
root
sh-3.1#
```
To summarize; the program in figure 3.6 is vulnerable to a buffer overflow attack and since the buffer in question is a local variable to a function; it is stack-based. The vulnerability can be confirmed by giving input that is larger than the 512-byte buffer and checking if the instruction pointer (the EIP register) gets changed due to the buffer overflowing. Now an overflow string is created which overflows the buffer and hijacks the instruction pointer by overwriting the return address. Instead of continuing execution at the place where main() was called, it continues somewhere on the NOP sled which eventually leads to the shellcode so that it starts executing. In other words; the program is forced to run code inserted into it.

3.2 Buffer overflows in other memory regions

Buffer overflows in the .bss segment and on the heap are closely related to stack-based overflows. Like their counterpart, they are made possible due to lack of bounds checking in the C functions used for writing data.

An important difference with these memory regions compared to the stack is the absence of return pointers. The stack is special in that it is involved in the execution of the program, the heap and .bss which will be discussed shortly, only house data. Thus, attacks can only target (i.e. overwrite) other data that is housed near a vulnerable buffer. Injecting and jumping to code in these segments is, however, entirely possible if they are marked as executable. This varies between implementations and will be covered in section 4.1.

3.2.1 Heap-based buffer overflows

Recall from section 2.3 that the heap allows for dynamic memory allocation. In C, this is done with the malloc() library call. This enables the programmer to allocate an appropriate amount of memory even when “an appropriate amount” is unknown during compile-time. In short, it enables constructs where memory can be allocated depending on how a user interacts with the program. This means that it is possible to write code like malloc(number_input_by_user) to allocate memory.

```c
1 int main(void) {
2 char *one = (char *)malloc(16);
3 char *two = (char *)malloc(16);
4 strcpy(one, "AAAAAAAAAAAAAAABBBBCCCCCCCCDDDD");
5 printf("one: %s\n", one);
6 printf("two: %s\n", two);
7 }
```

Figure 3.8: Simple example of a heap-based overflow.

Take the code in figure 3.8 as a basic example: the buffers one and two are allocated on the heap, each 16 bytes wide. Then, when one is filled with 32 bytes, some of it will spill over in the buffer two. When run, the following is outputted.
Two things to note in the above output: (1) one outputs all the 32 bytes previously set by `strcpy()` although its only 16 bytes large. (2) two only contains 8 bytes where one might have expected 16. Eight bytes appear to be missing. In reality, 32 bytes have been allocated in two 16-byte blocks with 8 bytes in between as seen below (each box holds four bytes):

```
<table>
<thead>
<tr>
<th>one</th>
<th>one</th>
<th>one</th>
<th>one</th>
<th>?</th>
<th>?</th>
<th>two</th>
<th>two</th>
<th>two</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAA</td>
<td>AAAA</td>
<td>AAAA</td>
<td>AAAA</td>
<td>BBBB</td>
<td>CCCC</td>
<td>CCCC</td>
<td>DDDD</td>
<td>\0</td>
<td></td>
</tr>
</tbody>
</table>
```

(1) is explained by `printf()`’s string handling; when the format specifier `%s` is used, it prints until a terminating null byte is encountered. Thus, the end of this string is actually stored in two’s address space. As for (2), memory regions on the heap are not adjacent but also contain meta data in between each region (marked with question marks above). The meta data is needed in order to manage the memory on the heap dynamically during runtime.

The example in figure 3.8 is not much of an attack although the buffer evidently is overflowed. Keep in mind though that the heap and .bss segment can be used to store important data and pointers just like the stack. For examples of this, see section 3.4.1 or 3.4.2.

### 3.2.2 .bss-based buffer overflows

Variables that are declared with the `static` keyword are initialized at compile time and any changes to them in functions will remain between function calls. This is made possible by storing these variables not on the stack but in the .bss segment which is located immediately below the heap (see figure 2.8). Overflows in this segment can be achieved in exactly the same way as in the previous example.

```c
1 int main(void) {
2     static char one[16];
3     static char two[16];
4
5     strcpy(one, "AAAAAAAAAAAAAAAAABBBBCCCCCCCCDDDD");
6     printf("one: %s\n", one);
7     printf("two: %s\n", two);
8 }
```

Figure 3.9: Simple example of a .bss-based overflow.

The code in figure 3.9 performs the same operation as that of figure 3.8 but with one difference; the buffers one and two are declared as `static` and is therefore stored in the .bss segment. The output is as follows.

one: AAAAAAAAAAAAAAAAAABBBBCCCCCCCCDDDD
two:BBBBBCCCCCCCCDDDD
Here, in `.bss`, the sizes of the buffers are expected to be unchanged throughout execution. This is why no meta data is needed and one and two are placed right next to each other in memory. This is evident from the overflow where 16 bytes are written to two’s address space. The situation can also be visualized as below.

<table>
<thead>
<tr>
<th>one</th>
<th>one</th>
<th>one</th>
<th>one</th>
<th>two</th>
<th>two</th>
<th>two</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAA</td>
<td>AAAA</td>
<td>AAAA</td>
<td>AAAA</td>
<td>BBBB</td>
<td>CCC</td>
<td>CCC</td>
<td>DDDD</td>
</tr>
</tbody>
</table>

### 3.3 Format string attacks

Format strings are strings that contain placeholders that define how the final string is interpreted. In C, such placeholders are identified by a percent sign (`%`). The most basic use is to insert variables holding numbers into strings. Several functions in the C language use format strings. `printf()` for example; where

```c
int num = 42;
printf("The answer is %d.\n", num);
printf("But %d is less than %d.\n", num, 96);
```

would result in

```
The answer is 42.
But 42 is less than 96.
```

as output. In the first case `%d` is the placeholder that `printf` replaced with for the number 42 contained in the variable `num`. On the second line two placeholders are used, one references the variable `num` and the other the value 96. Additional formatting is possible, like only showing two decimal places of a floating point number with `%2f` or displaying string constants with `%s`.

Note that `printf()` works like any other function discussed in section 2.3. Its parameters are pushed onto the stack before called and later during function execution, when a format specifier is found in the string its corresponding value is read from the stack and inserted appropriately. This is where the vulnerability comes in.

If there are more format specifiers in the format string than parameters passed to, for example, `printf()`; values will still be read off of the stack and inserted into the string. There is no check to see if the number of specifiers match up to the number of parameters. `printf()` assumes that they are present on the stack.

Format string vulnerabilities manifest themselves when a programmer writes

```c
printf(stringinputbyuser); // vulnerable
```

instead of

```c
printf("%s", stringinputbyuser); // not vulnerable
```

The difference is subtle but in the first case, format specifiers input by the user will be interpreted by `printf()` and in the second they will not. While the input string `%s` will cause the first `printf()` to display a string from the stack the second one will just output the string “%s”. If no format specifiers exist in the input string, the two statements will be functionally equivalent. Here is a command line session showing this against the program `fs`, shown in figure 3.10.
Figure 3.10: Code with format string vulnerability.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    char *password = "reallysecret";
    printf("The argument was: ");
    printf(argv[1]);
    printf("\n");
}
```

In the first round the legitimate string “hejsan” is given. Then the format specifier `%x` is used, for each specifier 4 bytes is read from the stack. Eventually the string specifier (`%s`) is used to display the string in password which is stored as a local variable to `main()` on the stack. Format string vulnerabilities can thus be used to read a process’ memory.

The vulnerability does not end there, however. Due to a format specifier called `%n`, it is possible to write arbitrary data to nearly arbitrary locations in the process’ memory. The `%n` specifier does not produce any output; instead it writes the number of bytes that have been written so far to the address given in its corresponding parameter.

```c
int num;
printf("AAAABBBBCCCCDDDD%n", &num);
```

After this, `num` will hold the value 12 because of the twelve characters outputted before `%n`. Writing arbitrary values to arbitrary locations using `%n` is a quite involved process and will not be covered here. (Erickson provides a thorough description of format string attacks in the book *The Art of Exploitation* [16].) The explanation given should be enough to understand the fundamental problem behind format string vulnerabilities; they make it possible to both read and write to memory.

### 3.4 Specific memory attacks

The attack covered in section 3.1 is the most popular and basic variant of buffer overflow attack: to overflow a buffer on the stack and overwrite the return address. However, data can be stored at other locations in memory as well and under certain conditions, they can also be overflowed like demonstrated with the heap and `.bss` earlier. This section will give a few more detailed examples of this.
3.4.1 Overwriting important data

```c
int main(int argc, char *argv[]) {
    int number;
    char buffer[16];
    char command[] = "/bin/ping";
    /* more code */
    system(command);
}
```

Figure 3.11: Local variables are pushed onto the stack in the order they are declared. `buffer` can be overflowed to overwrite `command`.

Recall that the local variables of a function are pushed to the stack in the order they are declared in the function. Thus, if `buffer` in the code in figure 3.11 is flowable during the function’s execution it is possible to overwrite the command in `command` later passed to `system()`.

The code in figure 3.11 has an example of a “high-level” vulnerability which is largely dependent on the program itself. A buffer overflow on the stack like this would only affect the program’s own logic. There are countless of potential scenarios, with varying degrees of complexity, where a program’s logic can be affected like this. Another prime example of this is overwriting authentication or authorization values.

3.4.2 Overwriting function pointers

In certain situations it can be advantageous to target variables stored below the buffer on the stack instead of the return pointer. A great example of this is when `function pointers` are present. Function pointers can be used to change a functions behavior during runtime. Consider the code in figure 3.12; `proceed()` is a function pointer that initially points to the `unauthorized()` procedure. If the given password matches the one defined on line 4 it is changed to point to the `authorized()` procedure. Thus calling `proceed()` on line 32 leads to different functions being called depending on where it points (which in turn depends on whether the password was correct).

Two aspects of this code make it exploitable: (1) there is a buffer overflow vulnerability on line 27 and (2) the function pointer is stored after the buffer on the stack as can be seen on lines 19 and 20. This makes it possible to overwrite the function pointer to point to `authorized()` so that it is executed at line 32 without providing the correct password.

3.4.3 Overwriting code pointers

With a `code pointer` we mean pointers that point to executable code. Examples of this are the return address on the stack and programmer-defined function pointers as in section 3.4.2. There are even more code pointers in a program and many of them can be overwritten to affect the flow of control. Examples of this
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PASSWD "reallysecret"

void authorized() {
    printf("Welcome!\n");
    // give access
    return;
}

void unauthorized() {
    printf("Goodbye!\n");
    // kick user out
    return;
}

int main(int argc, char *argv[]) {
    void (*proceed)() = unauthorized;
    char givenpasswd[512];

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <password>
", argv[0]);
        exit(1);
    }

    strcpy(givenpasswd, argv[1]);

    if (strcmp(givenpasswd, PASSWD) == 0)
        proceed = authorized;

    proceed();
}
```

Figure 3.12: Authentication code that is vulnerable to a buffer overflow. This is exploited by overwriting a function pointer.

are pointers in the global offset table (GOT) that is used to reference functions included from code libraries and pointers in .dtors that are automatically run at program termination. Exploitation of such pointers are explained by Erickson [16] in detail.

### 3.4.4 Reusing existing code

If it for some reason is impossible to inject code into a process and execute it, there is the possibility of using code that is already present. This is popularly known as return-to-libc attacks due to the standard C library being linked to every application coded in C. Just as in most of the previous attacks the goal is to affect the flow of control. But, instead of pointing to injected code, some appropriate function in the C library is targeted.

A good example of such a function is `system()` which is declared in the C library. `system()` takes one pointer to a command string as argument which it executes by calling the standard shell `/bin/sh`.

Exploitation can be done through a regular stack-based buffer overflow where
the return pointer is overwritten to point to the address of system(). (This address can easily be extracted with a debugger and will be static until the library is recompiled.) After jumping, when system() starts executing it expects to have its argument stored on the stack just as with any function call. Therefore, a pointer to the command string of choice must be stored on the stack above the return address.

Consider a stack that looks like in figure 3.13(a) where it is possible to overflow the buffer. In 3.13(b), the return address is overwritten with the address of system(). The next four bytes are overwritten with garbage and the following four with an address pointing to the string /bin/sh. When the function returns, the address of system() will be jumped to and “CCCC” will be interpreted as a saved return address. See 3.13(c). When the function executes it assumes that its only parameter was pushed before the return address. So, it uses the string /bin/sh which starts an interactive shell. The shell temporarily stops further execution and when it is terminated, the program will try to jump to the saved return address “CCCC” and segfault.

There are numerous variants of return-to-libc but the one demonstrated above is the most basic. Attacks that reuse existing code are still under active research. Shacham developed a technique that avoids using function calls entirely, instead building functions by using existing code [17].

### 3.5 Integer problems

Arithmetic on computers differ from that on paper in that the operands have an upper limit of how large values they can assume. This is due to variables used to store values having a fixed size. If not careful, this can lead to unexpected results. Additions, for example, are performed modulo their maximum capacity plus one. Thus, if a variable with a 32 bit capacity (the most common size for integers) holds a value of 0xffffffff and some positive value larger than 0xf is added to it, it will appear to “wrap around”, starting from zero again. In the following example, val will become equal to 0x3.

```c
int val = 0x13;
val = val + 0xffffffff;
```

To be able to represent negative numbers in binary, the concept of signed integers are used. In a signed integer, the most significant bit (MSB) is a one
if the number is negative and a zero if it is positive. To make signedness work properly in arithmetic, other modifications to the representation have to be done as well but this is beyond the scope of this discussion. We can settle with the view that \(0x00000000\)–\(0x7fffffff\) are interpreted as positive while \(0x80000000\)–\(0xffffffff\) are negative. \(0x7fffffff\) is the largest positive value available for 32 bit integers while \(0x80000000\) is the largest negative value.

\[
\begin{align*}
0x00000001 &= 1 \\
0x7fffffff &= 2147483647 \\
0x80000000 &= -2147483648 \\
0xffffffff &= -1
\end{align*}
\]

The result of the unexpected wrap-around is commonly referred to as integer overflows. Problems that arise due to treating unsigned integers as signed or vice versa are called signedness bugs. Bugs of this kind are not exploitable in the same way as buffer overflows in that it is not possible to overwrite data. However, when user controlled integers are used to calculate the amount of data to write into a buffer or to check boundaries, attacks can be possible. A classic example of a bug like this, a signedness bug, is given by blexim [18].

```c
1 int copy_something(char *buf, int len) {
2    char kbuf[800];
3    if (len > sizeof(kbuf))
4        return -1;
5    return memcpy(kbuf, buf, len);
6 }
```

Here a character buffer along with its length is taken as input and the length is checked to ensure that the buffer will fit in its destination. If this is the case the buffer is copied. The problem is that when the comparison is made on line 3, the integer \(len\) is considered signed but \(memcpy()\) treats it as unsigned. So, when a length (\(len\)) larger than \(0x7fffffff\) is given, the if statement will evaluate to false since \(len\) is a negative value, but \(memcpy()\) will treat it as unsigned, i.e. an enormous positive value. This will of course result in a huge amount of data (whatever is stored above \(buf\) in the stack) being copied into \(kbuf\), causing a buffer overflow.

### 3.6 Race conditions

According to the Free On-Line Dictionary Of Computing (FOLDOC), a race condition can be defined as “Anomalous behavior due to unexpected critical dependence on the relative timing of events”. Or, in gentler but less general words; “when two or more entities through reading or writing to the same resource at the same time make the outcome unpredictable”. Race conditions are sometimes known as concurrency vulnerabilities. In computers this is a problem that occurs when processes or threads that run in parallel use a shared resource, be it a temporary file or a global variable.

Race conditions can lead to serious reliability issues for example when two threads in a program both want to increment a variable. The execution might move forward as expected like this.
Thread 1 sees that variable = 0.  
Thread 1 loads variable into register to increment it.  
Thread 1 stores 0 + 1 = 1 into variable.  
Thread 2 sees that variable = 1.  
Thread 2 loads variable into register to increment it.  
Thread 2 stores 1 + 1 = 2 into variable.

This results in variable getting the value 2. If the threads suffer from a race condition, however, the outcome could be unpredictable.

Thread 1 sees that variable = 0.  
Thread 1 loads variable into register to increment it.  
Thread 2 sees that variable = 0.  
Thread 2 loads variable into register to increment it.  
Thread 1 stores 0 + 1 = 1 into variable.  
Thread 2 stores 0 + 1 = 1 into variable.

In this case, variable gets the value 1 since it is loaded by the second thread before the first one had time to store it. What the effect of this anomaly is entirely dependent on the affected program. What is important here is that a race condition bug has occurred.

Fithen [19] classifies the possible problems regarding race conditions:

Deadlock: one or more threads may become permanently blocked.

Loss of information: saved information is overwritten by another thread. (This was the case above.)

Loss of integrity of information: information written by multiple threads may be arbitrarily interlaced.

Loss of liveness: imbalance in access to shared resources by competing threads can cause performance problems.

Such conditions can become security problems when the check involves e.g. authentication or authorization routines. A privileged process is about to write to a file: (1) it checks for some condition and (2) if it evaluates properly it (3) performs the write. The problem is, an attacker might be able to switch the file for another during the time gap between (2) and (3), after checking but before writing. This problem is sometimes referred to as a time of check – time of use (TOCTOU) problem [20].

Security-related race conditions mostly concern file system objects since this is the most common shared resource on a computer.

26
3.7 Injection attacks

This section will cover two different attacks that are of less technical nature than the previous ones. Both these attacks are made possible by the lack of validation of user input and the mixing of data and code.

3.7.1 Command injection

For lack of a better name for this attack, it will be referred to here as command injection. The attack shares properties with code injection as treated in section 3.1.2 in the sense that code is given when data is expected. Despite this similarity command injection is treated separately due to it working at a completely different layer.

Historically, a common source of command injection vulnerabilities was when a program that a user interacted with asked for parameters that were used to run a command. A program doing this is commonly called a wrapper; it wraps around the command in question and executes it on the user’s behalf.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char cmd[64] = "cat ";
    strcat(cmd, argv[0]);
    printf("--begin-----------------
");
    system(cmd);
    printf("--end-----------------
");
}
```

Figure 3.14: Wrapper for the program cat that is vulnerable to command injection.

Figure 3.14 shows example code of a wrapper program, wrapper, that prints the content of a file by executing the program cat with the given argument and printing a line above and below the output. The difference can be seen in the command line session below.

```
stef@host$ cat bruce.txt
If you think technology can solve your security problems, then you don’t understand the problems and you don’t understand the technology. -- Bruce Schneier
stef@host$ wrapper bruce.txt
--begin-----------------
If you think technology can solve your security problems, then you don’t understand the problems and you don’t understand the technology. -- Bruce Schneier
--end-----------------
```

\[2\text{\texttt{cat}}\text{ is short for “concatenate” and is a traditional UNIX program; it is mainly used for printing the content of files.}\]
The difference is subtle but imagine a scenario where a user, for some reason, is only allowed to execute the wrapper program. (Assume that wrapper runs with different privileges than the user.)

```
stef@host$ ls /
Permission denied
stef@host$ wrapper "bruce.txt;ls /"
--begin----------------
If you think technology can solve your security problems, then you don’t understand the problems and you don’t understand the technology. -- Bruce Schneier
bin/  doc/  include/  lost+found/  mnt/  srv/  usr/
boot/  etc/  man/  opt/  root/  sys/  var/
devel/  home/  lib/  media/  proc/  sbin/  tmp/
--end----------------
```

Issuing the command `ls /` to list the content of the root directory is not allowed. However, since `wrapper` is vulnerable to command injection it can be forced to execute the command for us. The content of the root directory is printed after Schneier’s quote but before the end line, meaning that is is executed by `wrapper`. This is possible because (1) the variable `cmd` given as parameter to `strcat()` on line 9 in figure 3.14 becomes `cat bruce.txt;ls /` and (2) UNIX-like systems interpret 3 semicolons (;) as command delimiters. `wrapper` is manipulated into running the command.

### 3.7.2 Directory traversal

`Directory traversal` has a lot in common with command injection as discussed previously in section 3.7.1, but it is not about executing additional commands. Rather, directory traversal can be a possible vulnerability when a program handles files and the name of the file is given by the user.

Files in common operating systems are referenced by their `path`. This path can be `absolute` or `relative` and looks a bit different depending on which operating system is used. Absolute paths describe where a file is in relation to the top of the directory structure. In Windows this could look like this:

```
C:\Windows\system32\calc.exe
```

when referencing the calculator program. In UNIX-like systems the `C:` abstraction is not used and forward slashes (`/`) are used in place of backslashes (`\`). The equivalent path would be like this:

```
/usr/bin/xcalc
```

Both of the examples above are absolute paths. Relative paths refer to a file in relation to the current directory. If we currently are situated in the directory `C:\Windows` the path

```
system32\calc.exe
```

---

3Actually, shells used in such systems does this interpretation.
can be used. If we however where situated in the directory `C:\Windows\Temp`
the relative path would be

```
..\system32\calc.exe
```

where the two dots (..) means “the directory above”, in this case `C:\Windows`,
and then down in `system32\calc.exe`. UNIX-like systems use double dots in
the exact same way. All this might seem trivial but is a major point in directory
traversal attacks.

No code example will be given to demonstrate directory traversals because
the concept is the same as with the code in figure 3.14. Consider instead a
hypothetical web application where the user is allowed to upload a file, an image
for example. If a file called `mypic.jpg` is uploaded and is saved as

```
C:\Inetpub\wwwroot\images\mypic.jpg
```

a directory traversal attack could possible. One would be to upload a malicious
executable with the name

```
..\..\..\Windows\system32\calc.exe
```

so that it will be saved as

```
C:\Inetpub\wwwroot\images\..\..\..\Windows\system32\calc.exe
```

which is equivalent to

```
C:\Windows\system32\calc.exe
```

effectively overwriting the calculator program with the attacker’s executable file.
Chapter 4

Defensive

This chapter will survey the chosen defensive techniques that try to prevent or mitigate the attacks covered in the previous chapter. The text will refer to the attack chapter in some discussions so make sure the attacks are well understood. The same perquisites still hold.

4.1 Non-executable memory

Many of the attacks previously studied rely on being able to inject code into a process’ memory and force the program to execute it. One strategy to mitigate such attacks is to prevent execution of code in certain parts of memory.

The von Neumann architecture, upon which the majority of today’s personal computers are built, does not make a distinction between data and code. Both are stored in the same memory and code is thus just data treated as executable instructions by the CPU [21].

So, the five-byte data string “4a41434b49” could be interpreted as the text string “JACKI” or as the assembly instructions

\[
\begin{align*}
&\text{dec } %edx \\
&\text{inc } %ecx \\
&\text{inc } %ebx \\
&\text{dec } %ebx \\
&\text{dec } %ecx
\end{align*}
\]

depending on the context. This is one of the reasons why it is possible to execute shellcode injected as data into a process’ memory [21].

The Intel x86 processor has the ability to mark regions of memory as readable and/or writable but there has not been a flag like that for execution privileges until recently. Because of this, there are two approaches to disallowing execution in certain memory regions; the hardware-supported flag and software emulation of it when the flag is unavailable on the platform.

4.1.1 $W \oplus X$

$W \oplus X$ is not a technology per se but rather a concept. The idea is that every memory page should be writable (W) or executable (X), but not both. This is
Figure 4.1: The gray areas signify where the segment limit start and end (the executable area). Segment limits in (a) allows execution in the entire memory space while (b) provides a non-executable stack and (c) only allows execution in the .text segment.

the behavior of the exclusive or (xor) operator, either one can be true but not both at the same time. This idea was described by Theo de Raadt of OpenBSD [22]. The following two sections cover how memory can be made non-executable.

4.1.2 Software emulated

It is possible to emulate execution permissions in memory with the help of the “code segment limit”. This limit makes it possible to designate the first n megabytes of virtual memory as executable while the rest is not [23]. This method was first introduced for Linux as a kernel patch by Solar Designer [24] and has later been used by PaX Linux security patch [25], the OpenBSD operating system [26] and Red Hat’s ExecShield [23].

By default, the code segment limit stretches the entire 4 GiB\(^1\) of virtual address space which implies that everything is executable. This is illustrated in figure 4.1(a). PaX, ExecShield and similar patches adjust the segment limit to cover as little of the memory space as possible, effectively denying execution in certain memory regions. Figure 4.1(b) shows a non-executable stack and 4.1(c) a even less permissive setup where only the .text section is executable.

The major drawback of using code segment limits for removing execution permissions is its lack of granularity [23]. The resolution is in the order of megabytes and therefore it can become difficult to find the perfect boundary. Additionally, the fact that only one boundary can be set does not provide any flexibility. In other words it is impossible to allow execution in a subset of non-executable memory.

It should be noted that PaX does not rely entirely on this segment limit but can provide granular control via other means [25].

---

\(^1\)GiB is short for *giga binary byte* and is more accurate than the probably more familiar *mega byte* when dealing with numbers of base 2 (binary). For example 1 GiB = 2\(^30\) bytes = 1,073,741,824 bytes compared to 1 GB = 10\(^9\) bytes = 1,000,000,000 bytes.
4.1.3 Hardware-assisted

Execution permission flags were introduced to the Intel x86 architecture by AMD with the release of their Athlon and Opteron 64-bit processors [27]. AMD calls the technology NX as in “No eXecute”. Shortly afterwards Intel released their version, instead called XD for “eXecute Disable” [28]. These security features are sometimes, mostly in marketing materials, also referred to as “Enhanced Virus Protection”. Both of these technologies work exactly the same despite having different names [29].

NX/XD provides an execution permission bit for each entry in the page-translation-table [30] [28]. Thus, it is now possible to prevent execution on a page-level (4 KiB\(^2\)) granularity. The NX/XD flag works much in the same way as the read/write or user/supervisor flags (these are treated in section 4.3.1). Execution permissions does not, however, depend on whether it is done as user or supervisor.

It is possible to detect the presence of NX/XD capability in the processor by issuing the CPUID instruction with the value 0x810000001 loaded into EAX. If bit 20 in the return value placed in EDX is set, NX/XD capabilities are available [31] [27].

If the CPU has hardware support (NX/XD flags), two requirements must be fulfilled in order to use it. (1) Physical address extensions (PAE) needs to be enabled. PAE is a technology that allows 32-bit processors to handle more than 4 GiB of memory [32]. (2) The operating system must supply a policy for when pages are to be executable or not.

When an instruction held in a memory region where the page is marked as non-executable by the operating system is fetched for execution an exception is raised by the processor. This stops execution and effectively stops code from being run.

The data execution prevention (DEP) feature which arrived with Windows XP Service Pack 2 was the first Microsoft product to take advantage of the NX/XD bit [33]. There is also a patch for the Linux kernel with similar functionality [34].

4.2 Address randomization

Address randomization relies on the assumption that an attacker needs to have accurate knowledge of a process’ memory layout in order to reliably exploit it.

Recall from the sections on memory corruption attacks that return-to-libc require exact knowledge of the library call’s address to work. The stack-/heap-/\_bss-based overflows can sometimes settle with a less accurate guess if a sufficient number of NOP instructions can be used.

Traditionally, this requirement has not been a major issue for attackers, partly because of the quite ingenious use of NOP instructions but mostly due to the deterministic behavior of the virtual address space of executables. A specific compiled executable (including copies of it) will always have basically the same layout during execution. Therefore, the address of \texttt{system()} on the attacker’s operating system will be the same as that of the same system at another site.

\(^2\)See previous footnote on GiB.
msf > use windows/smb/ms05_039_pnp
msf exploit(ms05_039_pnp) > show targets

Exploit targets:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Windows 2000 SP0-SP4</td>
</tr>
<tr>
<td>1</td>
<td>Windows 2000 SP4 French</td>
</tr>
<tr>
<td>2</td>
<td>Windows 2000 SP4 Spanish</td>
</tr>
<tr>
<td>3</td>
<td>Windows 2000 SP4 English/French/German/Dutch</td>
</tr>
<tr>
<td>4</td>
<td>Windows 2000 SP0-SP4 German</td>
</tr>
<tr>
<td>5</td>
<td>Windows 2000 SP0-SP4 Italian</td>
</tr>
<tr>
<td>6</td>
<td>Windows XP SP1 English</td>
</tr>
</tbody>
</table>

msf exploit(ms05_039_pnp) >

Figure 4.2: Metasploit showing different targets.

This has led to pre-packaged exploits for vulnerabilities coming with different addresses for different targets.

In figure 4.2, the exploitation tool Metasploit Framework [35] shows the different targets provided for the 2005 Plug’n’Play vulnerability [36]. This suggests that the vulnerable code in each of these targets are somewhat different and that the address offsets differ.

The goal of address randomization is to introduce randomness into the memory layout of processes so that hardcoded addresses as in figure 4.2 cannot be used.

The first implementation of address randomization was done by the PaX team [25] in their Address Space Layout Randomization (ASLR) patch for the Linux kernel. Address randomization has later been inserted into the mainstream Linux kernel [37] and Microsoft Windows Vista [38]. It is also available by default in OpenBSD since version 3.5 [26]. Prior to Vista getting support for address randomization a small company called Wehnus released a host intrusion prevention system; WehnTrust [39], that among other functions provided address randomization.

### 4.2.1 What to randomize

Remember the memory layout of a process as discussed in section 2.3 on page 7. Since more or less all of those areas can be used to leverage a buffer overflow vulnerability, they must all be randomized. If, for example, the addresses in .text section are deterministic, we could potentially jump past a security critical piece of code. If the stack is randomized, but the heap is not, it can leave an opening. This goes for the other segments as well.

Additionally, the return-to-libc exploit addressed in section 3.4.4 on page 23 should be taken into consideration. The C library can be (and usually is) loaded as a shared library and addresses of the functions inside it need to be randomized as well.
Bhatkar et al. [40] notes that two types of randomizations are needed; that of (1) absolute locations and that of (2) relative locations. The issue of absolute locations can be solved by randomizing each memory segment’s base address. This can be done by subtracting a large random number from the stack pointer or allocating a large random block from the heap at process startup [40] [41].

Relative locations refer to the distance between two places in the same memory segment. Changes to the base addresses does not affect the relative distances. Instead this is solved by (1) inserting random gaps between logical memory units and/or (2) permuting the order in which units are stored [40].

Another detail that complicates things for the defender is the fact that not all bits in an 32-bit address can be random. This was first noted by the PaX team in their ASLR implementation [25]. Process memory is split into three regions under PaX (executable, areas handled by mmap() and the stack), these get 16, 16 and 24 bits of randomization, respectively. In Windows Vista, the executable image has 8 bits, the heaps 5 bits each and the stack 14 bits [42]. Note that Vista performs some additional randomization, this only considers the base addresses. Consequently, there are not $2^{32}$ possible locations.

4.2.2 When to randomize

Yet another issue with randomization is at which point in time the addresses are to be changed. This is important since it is favorable to randomize as often as possible to make intelligent brute force attacks difficult while, at the same time, as seldom as possible to reduce overhead. To make guessing attacks as difficult as possible, process memory should ideally be re-arranged on every attack probe. This way, an attacker never learns anything from each attempt and the probabilities will be static [43]. Although ideal, this interval is too abstract. In reality, implementations perform randomization at (1) compile-time, (2) system boot-time, (3) load-time or (4) during run-time.

Clearly, the ideal timing can only be reached with randomizations done as in (3) or (4). In the former case, the program is assumed to crash on each attack probe and reload with a new address space. This leads to an interesting problem with servers that use the fork() library function to create a new process to handle an incoming request that will be discussed later in chapter 5. As for the latter case (4) doing re-randomizations during run-time is impractical [40]. Generally, it is more efficient to keep the transformation close to compile-time [40].

PaX transforms the address space during program load [44], like in (3). The majority of randomizations in Windows Vista are done on system boot as in (1), thus requiring a complete restart of the operating system to change the address space [42].

4.3 System call interception

System calls are generally used by every program running on a computer but programmers rarely use them directly. Instead they are wrapped in library functions in for example the C library. This section explains how system calls work and how they can be used for security purposes.
4.3.1 Operating system calls

The operating system performs many tasks, one of which is to provide an abstraction layer between hardware and application programs. The applications are prevented from performing this interaction by themselves for convenience, fault tolerance and security reasons. This restriction is made possible by the use of hardware-enforced access control. Most modern CPU architectures include hardware flags to signify which access level certain code has. These access levels are usually referred to as protection rings [3]. Four levels (two bits) are defined in x86 but UNIX only uses two; the operating system lives in ring0 and applications in ring3. (These are the user/supervisor flags referred to in section 4.1.3.) Access to inner rings are only allowed via special interfaces known as system calls (often abbreviated as syscalls) [45].

System calls are more or less analogous to function calls as discussed in 2.3 on page 7. An application will have to make syscalls for doing seemingly basic things like accessing the filesystem or sending data across networks. The application asks for a service by issuing a syscall, the kernel processes the request and returns the result to the application.

In UNIX, syscalls are traditionally made by loading values into registers and sending an interrupt, thereby letting the operating system kernel take over control. When the kernel is finished, the results, if any, will be left in the registers. In assembly, use of the exit syscall which is used to terminate a program could look like figure 4.4.

```
mov %eax,0x5 ; load the value of exit into eax
mov %ebx,0x0 ; load 0 as the exit status code into ebx
int 0x80 ; interrupt and transfer control to kernel
```

Figure 4.4: Assembly to make an exit syscall.

The assembly instruction int 0x80 is equal to \xcd\x80 in machine instructions. As can be seen in figure 3.4 on page 15 depicting the shellcode injected via a buffer overflow in section 3.1.1, two system calls are made; on the first and third line.
The idea of monitoring the system calls a program makes is based on the assumption that these syscalls are a good measure of a program’s behavior [46]. For a program to perform any meaningful task, help from the kernel via syscalls is generally needed, thus supporting this assumption.

4.3.2 System call monitoring

Monitoring system calls for security reasons was first suggested by Forrest et al. in [46]. Forrest’s paper centers on the view that one can distinguish between actions that are native of a program and actions that are inserted into it (i.e. injected via a buffer overflow), thus providing anomaly detection.

4.3.3 Monitoring syscall sequences

The technique Forrest used was to sample and store sequences of system calls that a program makes during run-time and then in later runs, check if it exhibits known sequences. We refer to that initial phase as training. If all calls made by a program from start to finish are seen as one long series, the sampling is done by sliding a fixed size window along it and recording the call sequences contained in it. See figure 4.5.

This is obviously more complicated in a real scenario where a program would elicit different call sequences depending on the execution path, of which there usually are many.

In a later paper [47] Hofmeyr, Forrest and Somayaji showed that system call sequences indeed are a good discriminator between normal and abnormal program behavior. The technique does have some problems associated with it, though. Since then, many researchers have attacked these problems [48] [49] [50] [51] [52]. On of the more notorious of these will be covered next.

4.3.4 Mimicry attacks

There are two major problems with this and similar schemes. First, it is very difficult to extract the required call traces from the program properly. In other
words: developing a good way to train the anomaly detector is a difficult problem [53]. We will not go into this problem further in this thesis. Secondly, just taking the syscall and its ordering into consideration makes the implementation vulnerable to a mimicry attack, first envisioned by Wagner et al. in [53].

Like other function calls, system calls have parameters. This was shown earlier in figure 4.4 where `exit` had one parameter of “0” to signify its status code. Parameters like this are ignored in the above syscall sequence implementation. This could become a problem when a benign system call is populated with a malicious parameter. This weakness has been exploited by several researchers [54] [55] with the help of mimicry attacks.

The mimicry attack tries to mimic a program’s normal behavior to evade detection. The way it does is best illustrated with an example. Suppose we need to make a monitored program open a file, make two successive writes to it and then close the file, this can for example be done via a buffer overflow vulnerability. Let us use the example sequence in figure 4.5. Our shellcode would then have to contain the syscalls `open`, `write`, `write` and `close`, see 4.6(a). These are not covered in the sampled sequences and calls to them in that order would be flagged as anomalous. To avoid this, our syscall sequence has to be “hidden” among legitimate sequences, mimicking an allowed sequence of calls.

The key to mimicry attacks is being able to prevent a syscall from having side-effects, effectively making it a no-operation so that it can be executed without affecting the program’s logic. Examples of this would be opening a non-existent file or reading zero bytes. Nearly all syscalls can be made into no-ops [54]. Since our current goal is to call `open`, `write`, `write` and `close`, these calls obviously need to be present in the program itself and each must be recorded in at least one allowed sequence. The shellcode using a mimicry attack could in this case look like figure 4.6(b). Considering that the calls written in italics are no-operations, the call sequence is identical to the sequence 4.6(a). Still, it matches each sampled sequence in figure 4.5.

Constructing attack sequences like in figure 4.6(b) manually is tedious and error prone. However, there has been research showing that discovery of mimicry attacks can be automated [56] [57].

4.3.5 Other monitoring techniques

With the advent of the mimicry attack, it became obvious that more sources of information were needed than the mere sequence of calls made by a program. Techniques for syscall monitoring have been classified by Gao et al. in [58] into black-, gray- and white-box approaches.

Black-box approaches only concern themselves with the actual system calls made by a program: the syscall number and its arguments.
Gray-box techniques utilize additional information that can be extracted from the process’ memory during runtime. Examples of this is the monitoring the instruction pointer and analyzing the call stack while the program is executing.

White-box detectors extend this by also examining the program via static analysis on the source code or binary itself, possibly making changes to it.

One elaborate black-box approach to counter mimicry attacks suggested by Kruegel et al. [49] centers on the detection of anomalous syscall arguments. Four models of how to recognize anomalous arguments are presented: string-length, character distribution, structural inference and token finder. Looking for anomalies in string lengths given as arguments to syscalls is straightforward. Many times malicious input by an attacker is pushed through via syscalls and if a buffer overflow or format string is involved the length of the given string can be much longer than usual. Character distribution takes advantage of the fact that a large part of the strings handed to syscalls only cover a small part of the available (256) character space. This is measured using standard frequency analysis. Just as in any language, the characters used are not uniformly distributed, which will result in patterns emerging. This method calculates the distribution by percentage without taking the individual characters into consideration. Thus, the three strings “parrot”, “raptor” and “passwd” would have the same distribution, with “r”, “r” and “s”, respectively, occurring two times and the rest one time each. This distribution, called idealized character distribution (ICD) would be 0.33, 0.17, 0.17, 0.17, 0.17 followed by 0 occurring 251 times. The ICD is later used in a statistical test to detect anomalous arguments. Structural inference tries to infer grammar in allowed system call argument strings. For example, if a process only opens files in its working directory, the path in the argument will lack a leading slash character, which can then be inferred. Token finder seeks to determine if arguments to a syscall are part of a limited set or drawn at random. A threshold is set during training and if the number of unique arguments exceed the threshold, the argument is said to be random. Otherwise the arguments are tokens in a set and deviations are considered anomalies.

Gray-box anomaly detection has become a widely studied subject with the arrival of mimicry attacks [52] [58] [59] [60] [57]. The use of call stack monitoring for anomaly detection was pioneered by Wagner and Dean via static analysis [53] but later adapted by Feng et al. for use during runtime [52].

Instead of monitoring the syscall sequences themselves, the execution paths taken by these calls are used as the metric [52]. A model of allowed paths is constructed during training and later compared against to detect illegal control transfers. This method not only allows for detection of attacks that aim to execute code, but attacks that try to skip security-related code sections such as authentication routines.

4.3.6 Concurrency vulnerabilities

Robert Watson released a paper [61] in 2007 covering his research into concurrency vulnerabilities in syscall monitor systems. According to Watson, syscall interception schemes\(^3\) inherently vulnerable to race conditions due to lack of

\(^3\)This covers monitoring techniques in general where a program’s behavior is determined by the syscall it makes. These include the scheme presented in section 4.3.3.
atomicity in system calls and execution being concurrent. Regardless of whether a system supports true concurrency (multiple processors) or not (single processor), race conditions can be created and exploited.

The syscall interceptor is viewed as a reference monitor\footnote{This is not strictly a reference monitor since it does not adhere to all the requirements defined by Anderson in \cite{62}. The term is used somewhat sloppily here because its tasks are similar.} between the application and kernel. With the help of a different thread/process the arguments of a syscall can be changed so that the reference monitor and kernel will make different interpretations of its meaning. This can lead to privilege escalation and audit bypass.

Watson identifies two ways to mitigate the vulnerability. (1) Rewriting the kernel operation model or (2) integrating the security monitoring into the kernel itself.

In the first approach the goal is to send the complete syscall with associated arguments to the kernel at once instead of on-demand. This would ensure that the kernel and interceptor would receive the same data so that different interpretations are impossible.

The second approach strives to eliminate race conditions by performing security checks inside the kernel. This way resources can be locked, checked and unlocked to prevent changes due to races. Watson notes that changes to the kernel is somewhat contradictory to the original idea of syscall interception.

### 4.4 File integrity monitoring

At least one aspect sets the method presented in this section apart from the previous two. The protection provided is not as “automated”; supervision and intervention by a human user is strictly required since it is more about detection than prevention. Arguably though, this is a requirement regardless of what techniques are used.

The filesystem is often the largest shared resource in a computer. Problems with race conditions which were treated in section 3.6 on page 25 frequently involve the filesystem. Furthermore, vulnerabilities that allow writing to arbitrary files often result in privilege escalation attacks. The file system is a security critical resource and needs to be protected just like process memory. The obvious goal is to prevent unauthorized reads and changes from being made, or in computer security terminology, protecting the confidentiality and integrity of the files. Discretionary access control to files is available in virtually every modern operating system. But, if a privileged program is subverted into doing changes this makes little difference. Unauthorized changes made by authorized users are of special interest.

A filesystem can be fairly disorganized with unexpected content, constantly changing and, lately, extremely large. Fortunately, far from everything in the filesystem is of security critical nature.

The technique we will discuss to address this problem are known as file integrity monitors, host integrity monitors, host intrusion detection systems (HIDS) and others. Of these, file integrity monitoring appears to be the most accurate; it will be the term used here. The fundamental concept of the technique is simple and can be generalized to four steps. (1) Extract and store
information about the files and directories that are to be monitored. Periodically, (2) extract the same information again, (3) make comparisons to detect changes and, if changes are detected (4) send an alert to the administrator.

File integrity monitors have been available for a long time, the conceptual idea even longer. Tripwire, the program which made the technique famous, was released in November 1992 [63]. Since then multiple other implementations have surfaced, primarily for the Linux and BSD platforms. Among these OSSEC, AFICK, Samhain, Osiris and Aide are the most well known.

4.4.1 Detecting changes

Many different methods are available for detecting whether two sets of data differ, even more so when they are stored as files in a file system. The choice is dependent on a trade-off between accuracy and speed and resources. Full accuracy is achieved by storing a full copy of the file and later making a byte-by-byte comparison. This enables exact changes to be detected. Of course, this doubles the storage requirement for each monitored file and is comparatively slow. Another extreme would be to only store the file size, the time of last modification (if available in the file system), other metadata or combinations thereof. This is not very accurate since changes can be padded to not affect the original file size and modification times can usually be manipulated.

Additionally, there are techniques that can reliably detect that a change has occurred but not exactly what has been changed. This is usually done with the help of cryptographic checksum algorithms such as MD5 [64] or the SHA family [65]. Among the pros of using these is that they are faster than making byte-by-byte comparisons, very reliable and require very little storage space.

4.4.2 Trust issues

File integrity checkers as discussed here run in user space and therefore trust the operating system when doing file processing. This means that the checker asks the operating system to open and read a file so that it is able to calculate its checksum. See section 4.3.1 on how operating system calls work.

This trust can be exploited by the use of rootkit techniques. An example of this is to incorporate code into the operating system that covertly intercepts system calls (much like in section 4.3) and hides/manipulates the information passed back to user space or the other way around. Techniques like this is popularly used by attackers to hide their presence on a compromised system.

One way rootkits can aid in subverting file integrity checkers is suggested in [66]. Here the operating system calls are modified so that when open is issued against a protected file the correct file is opened and thus has the correct checksum. However, when exec is called to execute it, the file is replaced by a malicious one. The integrity checker, which only uses open will never know. This particular technique, of course, only applies to executables.
Chapter 5

Analysis

The technologies surveyed in chapter 4 are seldom individually advertised as panaceas. This observation is widely accepted and supports the concept of “defense in depth” where multiple defensive actions protect the same resource at different layers. In this section, the defensive technologies will be analyzed further and an attack tree will be constructed to aid in visualizing where they are effective.

5.1 Defensive technologies

This section will give a short analysis of each defensive technique covered in chapter 4.

5.1.1 Non-executable memory

The protection provided by non-executable memory is very specific and relies a lot on the policy given by the operating system. It prevents code from being run in specified memory regions. Thus, provided that the stack area is marked as non-executable, stack-based overflows that jump to code injected on the stack (as shown in section 3.1.3 on page 16) will fail after jumping. This also applies to code injected into the heap or any other memory space marked as non-executable. It is very important to note that this does not remedy the buffer overflow vulnerability itself, it just prevents injected code from being executed.

Much work has been put in by the research community to analyze the effectiveness of NX/XD [67] [68] [69] [70] [71] [72] [73] [74]. Several methods of bypassing the protection have been suggested. The most interesting among them might be techniques that reuse code already present in the program, either in itself or in linked shared libraries. An example of this are return-to-libc style exploits discussed in section 3.4.4 on page 23. This way, execution never leaves the .text segment in memory and NX/XD is effectively rendered useless [74] [71] [67].

Another important point to make is that higher level exploits that jump certain routines or overwrite key values like in 3.4.1 on page 22 and section 3.4.2 on page 22 will obviously not be affected by NX/XD. Such vulnerabilities are,
however, very code-dependent and have to be tailored for a specific program to work.

As previously stated, making memory non-executable does not affect the real vulnerability: that it is possible to overwrite a program’s memory.

5.1.2 Address randomization

Since the PaX ASLR implementation for Linux was released there has been a large amount of research aimed at bypassing [75] [43] and improving [76] [40] [41] [77] this style of protection. One reason for the popularity of address randomization among some groups in academia might be that the risk of an exploit succeeding is quantifiable.

Address randomization is interesting in how it can be implemented in several different ways; PaX considers ASLR a part of kernel functionality [44] while others modify the program loader in user space [76]. Bhatkar et al. have tried performing the randomizations both on binary level [40] and source code level [41]. These approaches all have different trade-offs. Ideally, transformations should be made to binaries so that users can enable protection without access to kernel or program source code. However, Bhatkar et al. noted that the information available in compiled code is insufficient and requires changes to compilers and linkers [41]. Transforming source code is almost as good if there is access to it.

There are two general ways to break address randomization schemes: via information leakage or through brute force attacks. Information leakage can manifest itself through vulnerabilities like format strings (see section 3.3 on page 20) which can be used to dump parts of a process’ memory in certain cases so that the randomized addresses are discovered. Also, Linux systems provides a detailed description of each process’ memory via the /proc virtual file system. This is of great importance considering local attacks.

Brute force attacks connects back to the statement regarding daemons that use fork() in section 4.2.2 on page 35. If a server daemon runs as a parent process that use fork() to create copies of itself to handle incoming requests, the child processes will have the same memory layout as the parent. Thus, if an attack probe is sent with an invalid address resulting in the child process crashing due to segmentation fault, the next probe will hit a new child process with an identical memory layout, enabling the attacker to perform a linear search for the correct offset. This is equivalent to random sampling without replacement. On the other hand, if a server process is re-randomized on every probe, a linear search will not be possible; every probe is equally likely to succeed: random sampling with replacement. While the second scenario obviously is preferred, Shacham et al. showed that it only provides one additional bit of entropy compared to the first scenario [43]. As an example a PaX protected Apache server was given, 16 bits of entropy are expected to require $2^{16}/2 = 2^{15} = 32,768$ probes when the memory layout is unchanged throughout the attack. With ideal re-randomization intervals (between every probe) $2^{16} = 65,536$ probes would be expected [43]. The amount of probes is only doubled, making it just as vulnerable to brute forcing.

A general complaint against address randomization is that computers and their programs are supposed to be deterministic. Address randomization changes this property, making the memory space unpredictable so that debugging will
become very difficult. A more implementation specific problem is that the address space provided by 32-bit processors is too small for effective randomization. The available entropy is too low and it makes brute force attacks feasible [43]. However, with the advent of processors that use 64-bit addresses this problem will go away since every added bit of entropy doubles the required search space.

5.1.3 System call interception

The amount of academic research into operating system call interception is staggering. Several different solutions to several different problems are proposed, and this is important, syscall interception is used in very diverse ways. For example preventing a web browser from being exploited by restricting which syscalls it can perform [78] is a different problem from trying to detect anomalies in the syscalls issued by a mail server [59].

It is not difficult to see why system calls are popular in security monitoring. They are the gateway that user applications must use to interface with the underlying operating system. For any meaningful work to be done, the operating system must be involved. Because so many syscalls are defined and many seemingly innocuous calls (reading a file) can be used for malicious purposes (reading the password file), depending on their arguments and context (what program, by whom etc), they are very difficult to monitor. With potential comes difficulties. This is evident from the huge amount of papers on creating and breaking new techniques for syscall interception.

Here is a quick rundown of some developments in syscall interception. In 1996 Forrest et al. [46] proposed the idea of monitoring syscalls, Goldberg et al. [78] created Janus which was used to restrict a program to using only a specified subset of calls. In 1998 Forrest and Hofmeyer et al. [47] introduced system call sequences for anomaly detection. This scheme has since been broken by numerous researchers with mimicry attacks originally introduced by Wagner [53] [54] [55] [57] [56] [79]. During this process, it became evident that more information was needed in addition to sequences. Using additional input to the detection engine was coined gray-box techniques by Gao et al. in [58]. Different approaches to this were to focus on the arguments to syscalls [49] or trying to keep state with the help of the program’s call stack [52]. Some of these gray-box approaches have been successfully attacked so that mimicry attacks can be performed [57]. Lately, research into interceptor’s vulnerability to race conditions have also been done [61]. There will probably be more research in this area. In parallel to all this, there is the fundamental problem of modeling a program or system’s behavior.

5.1.4 File integrity monitoring

There exists a fundamental paradox with integrity checkers as discussed in section 4.4. It was stated that “Unauthorized changes made by authorized users are of special interest”, insinuating that this can be prevented. This is not necessarily the case. If a user has sufficient privileges to change a file that is monitored, e.g. the password file which is only writable by the administrator, the user might also be capable of modifying the monitor or operating system, making evasion possible. It is very hard to protect something from its owner.
More problems arise if the database containing the checksums the monitored files is stored along with the files themselves. Some of these can be mitigated by signing the database entries cryptographically, but then, the key cannot be stored with the database. Without going into the details of this: it is a difficult problem.

5.2 Attack trees

The inventor of attack trees; Bruce Schneier, defines them as “a formal, methodical way of describing the security of systems, based on varying attacks” [80]. The tree begins with the root node where the goal of attack is, e.g. “open safe”. Children of the root will then describe ways of reaching this goal. Each path to the root node is a unique attack. Attack trees have been further studied by Moore et al. [81].

Attack trees can be represented graphically as a tree or in textually outlined form. The latter is preferred when dealing with larger trees and is the way that will be used here. The example tree that follows is the one given by Schneier when presenting the technique, see Schneier’s article [80] for the graphical representation of it.

Goal: open safe
OR 1. pick lock
   2. learn combo
      OR 1. find written combo
          2. get combo from target
          OR 1. threaten
              2. blackmail
              3. eavesdrop
              AND 1. listen to conversation
                  2. get target to state combo
          4. bribe
      3. cut open safe
      4. install improperly

Nodes are marked either OR or AND. This means that to “learn combo” you can “find written combo” OR “get combo from target”. But, when you “eavesdrop” you must “listen to conversation” AND “get target to state combo” in order to achieve this sub goal. Each leaf can also be tagged with one or more nearly arbitrary values. Here (P) is for possible and (I) is for impossible.

Goal: open safe ([!p]P)
OR 1. pick lock (I)
   2. learn combo (P)
      OR 1. find written combo (I)
          2. get combo from target (P)
          OR 1. threaten (I)
              2. blackmail (I)
              3. eavesdrop (I)
              AND 1. listen to conversation (P)
                  2. get target to state combo (I)
4. bribe (P)
3. cut open safe (P)
4. install improperly (I)

According to the tree it is possible to open the safe (goal) because it is possible to learn the combo (2.2) from the target (2.2) by bribing (2.2.4) him or her. It is also possible to cut the safe open (4). Note that it is possible to listen to a conversation (2.2.3.1) but since it impossible to get the target to state the combination (2.2.3.2) and because this is a AND requirement; eavesdropping (2.2.3) is impossible.

These values, here the Boolean possible/impossible, could also be the cost for carrying out the attack, the probability of success or whether it requires any special skill. Several metrics can also be combined. Another advantage with the tree structure is that sub-trees can be re-used. For example, if the main goal can be reached by compromising a Apache webserver a ready-made attack tree for Apache servers can be attached as a subtree.

The creation of an attack tree begins with the goal of attack being defined. Attacks that aim to achieve this goal are then conceived and added in an iterative fashion, preferably by several different persons. As Schneier notes [80] this “requires a certain mindset and takes practice”.

5.3 Visualizing defensive techniques

How do the defense technologies presented in chapter 4 make a difference? What attacks do they protect against? The most important question might be: at what stage in the course of an attack does it make a difference? We will attempt to visualize this with the help of attack trees.

This shifts the focus of attack trees somewhat; instead of showing how a system can be attacked the focus is on how attacks on an arbitrary system can be stopped or mitigated.

5.3.1 Attack tree

Figure 5.1 shows the attack tree that will be used for the purpose of visualizing the defensive techniques. It has been constructed according to the instructions in section 5.2 during the course of this thesis. The tree is neither comprehensive nor complete. The attacks, especially those that target memory, can be modified in countless ways. Some other possible attacks are left out due to being unrelated or unlikely. All this strives to reduce the complexity of the tree which would otherwise get out of hand quickly due to the resolution required to visualize some of the defense techniques.

Another issue might be the lack of target specification; what kind of system is under attack here? It has been decided to leave this aspect open because the attacks in this tree are very generic. It matters little if the target is a webserver serving dynamic content from an attached database or if it is a desktop computer mainly engaged in web browsing, email reading and word processing.
5.3.2 Defensive codes

Figure 5.2 shows the same attack tree as in figure 5.1 but with the addition of something this dissertation will term defensive codes. Defensive codes are tuples of three elements; noun, modifier and verb. The available options are referenced in table 5.1 and can (at least in theory) be combined freely. For example if non-executable memory regions will prevent an attack, we will write (N+!). If file integrity monitoring has the potential to detect the attack; it is (F-?). The verb “prevent” implies “detect”, i.e. prevention implies detection. These codes have been inserted on a selection of nodes in the attack tree.

Table 5.1: Defensive codes.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Non-executable memory regions</td>
</tr>
<tr>
<td>A</td>
<td>Address space randomization</td>
</tr>
<tr>
<td>S</td>
<td>System call interception</td>
</tr>
<tr>
<td>F</td>
<td>File integrity monitoring</td>
</tr>
<tr>
<td>+</td>
<td>will</td>
</tr>
<tr>
<td>-</td>
<td>will under simple condition</td>
</tr>
<tr>
<td>!</td>
<td>prevent and detect</td>
</tr>
<tr>
<td>?</td>
<td>detect</td>
</tr>
</tbody>
</table>

The process of placing defensive codes is not formal and is done in much the same way as creating the tree. Coming up with codes and placing them is an iterative process that requires time and understanding of both the attack and defense technologies. There are however a few guiding principles on how codes should be placed:

- Conditions should be of very simple nature. It can for example state that some way of bypassing the defense exists (mimicry attacks) or that the defense is not definitive but very likely to be effective (address randomization).

- If possible, bypasses to a defense should be listed as another attack. For example ⟨1.1.3.2⟩ use existing code in memory could be viewed as a bypass to inserting code into a stack buffer.

- The defensive code should be placed as low in the tree structure as possible in order to make it unambiguous and avoid conditions. For example, stating ⟨1.1.3.1.1⟩ fill stack buffer (N+!) is preferred over stating ⟨1⟩ execute code (N-!) under the condition “shellcode is placed on the stack”.

- Defensive codes are propagated upwards if it affects a lone OR subgoal or one of several AND subgoals.

- Defensive codes that are propagated from a child are denoted with with square brackets like in ⟨1.1⟩ jump to code in memory [A+!] which is due to ⟨1.1.2⟩ find address of code (A+!).
Goal: compromise security of program

OR 1. execute code
   OR 1. jump to code in memory
      AND 1. overwrite code pointer
         OR 1. overflow vulnerable stack buffer
         2. find vulnerable format string
         2. find address of code
         3. get code into memory
         OR 1. inject code
            OR 1. fill stack buffer
            2. fill heap buffer
            3. fill .bss buffer
            4. use environment variable
            2. use existing code in memory
         4. make program jump to code pointer
   2. execute command
      AND 1. inject command as data
      2. make program evaluate data as command
   2. modify internal logic
      AND 1. find address of data
      2. overwrite data
         OR 1. overflow vulnerable stack buffer
         2. overflow vulnerable heap buffer
         3. overflow vulnerable .bss buffer
         4. find vulnerable format string
   3. make changes to critical file
      OR 1. directory traversal
         2. execute shellcode that change files
         AND 1. overwrite code pointer
         OR 1. overflow vulnerable stack buffer
         2. find vulnerable format string
         2. find address of code
         3. get code into memory
         OR 1. inject code
            OR 1. fill stack buffer
            2. fill heap buffer
            3. fill .bss buffer
            4. use environment variable
            2. use existing code in memory
         4. make program jump to code pointer
      3. execute command that change files
         OR 1. cause segfault
         OR 1. write to read-only memory
            AND 1. overwrite pointer
            OR 1. find format string
               OR 1. find address of pointer
               2. overflow buffer next to pointer
               OR 1. overflow vulnerable stack buffer
               2. overflow vulnerable heap buffer
               3. overflow vulnerable .bss buffer
            2. make program use pointer
            2. execute from invalid memory
               3. abuse integer overflow/signedness bug
                  AND 1. overwrite code pointer
                  OR 1. overflow vulnerable stack buffer
                  2. find vulnerable format string
                  2. find code pointer in memory
                  3. make program jump to code pointer
            2. trigger defense mechanism
               OR 1. execute from non-exec memory
                  AND 1. overwrite code pointer to point into non-exec
                  OR 1. overflow vulnerable stack buffer
                  2. find vulnerable format string
                  2. make program jump to code pointer
                  2. overwrite canary
                     OR 1. find vulnerable stack buffer
                     2. find vulnerable format string
            4. overwhelm with legitimate input

Figure 5.1: Attack tree.
5.3.3 Analysis

Some nodes, the ones marked with asterisks (*), in the attack tree (figure 5.2) will now be discussed specifically. Some countermeasures or techniques for bypassing protection discussed earlier are not taken into consideration in this analysis to limit complexity and avoiding unnecessary repetition.

⟨1⟩ execute code (S−!)

System call interception will prevent execution of code. This holds under the condition that a working mimicry attack is not found and exploited by the attacker. The defense will be effective regardless of how the code got into the process, via ⟨1.1⟩ or ⟨1.2⟩.

⟨1.1.2⟩ find address of code (A+!)

Address randomization will prevent an attacker from finding the address of code in memory. In this case, address randomization is done in all executable memory segments and the entropy is large enough for the attack to be considered impossible. None of the subgoals listed below this node are viable and since this node is also part of a series of AND subgoals, this property is propagated to ⟨1.1⟩.

⟨1.1.3.1⟩ inject code

None of the available nouns in the defensive codes will protect against or detect when code is injected into memory.

⟨2⟩ modify internal logic (S−!)

System call interception will detect modifications of internal logic. The logic of a program decides how a program is driven forward and a syscall interceptor which looks for anomalies is very likely to detect when a routine is skipped or when calls are done out of order. Detection of data being overwritten (e.g. changing auth=false to true) might be less likely, depending on the side effects. Assuming that the modification will lead to an unknown code path and that a mimicry attack is not found, the defense will hold.

⟨2.1⟩ find address of data (A+!)

Address randomization will prevent finding the address of specific data. Just as for ⟨1.1.2⟩ above, the addresses of the data to overwrite must be found in order to modify logic.

⟨3⟩ make changes to critical file (F+?)

File integrity monitoring will detect changes made to a file. Clearly this assumes that the file being changed is being monitored but since it is critical, it probably is.
Goal: compromise security of program

* OR 1. execute code (S+!)
OR 1. jump to code in memory [A+!]
   AND 1. overwrite code pointer
   OR 1. overflow vulnerable stack buffer
   2. find vulnerable format string
   * 2. find address of code (A+!)
   3. get code into memory
   * OR 1. inject code
   OR 1. fill stack buffer (N+!)
   2. fill heap buffer
   3. fill .bss buffer
   4. use environment variable (N+!)
   2. use existing code in memory
   4. make program jump to code pointer
   2. execute command
   AND 1. inject command as data
   2. make program evaluate data as command
   * 2. modify internal logic (S-!), [A+!]
   * AND 1. find address of data (A+!)
   2. overwrite data
   OR 1. overflow vulnerable stack buffer
   2. overflow vulnerable heap buffer
   3. overflow vulnerable .bss buffer
   4. find vulnerable format string
   * 3. make changes to critical file (F+?)
   * OR 1. directory traversal (S-!)
   2. execute shellcode that change files (S-!) [A+!]
   AND 1. overwrite code pointer
   OR 1. overflow vulnerable stack buffer
   2. find vulnerable format string
   2. find address of code (A+!)
   3. get code into memory
   OR 1. inject code
   OR 1. fill stack buffer (N+!)
   2. fill heap buffer
   3. fill .bss buffer
   4. use environment variable (N+!)
   2. use existing code in memory
   4. make program jump to code pointer
   3. execute command that change files
   * 4. make program crash
   OR 1. cause segfault
   OR 1. write to read-only memory
   AND 1. overwrite pointer
   OR 1. find format string [A+!]
   OR 1. find address of pointer (A+!)
   2. overflow buffer next to pointer
   OR 1. overflow vulnerable stack buffer
   2. overflow vulnerable heap buffer
   3. overflow vulnerable .bss buffer
   2. make program use pointer
   2. execute from invalid memory
   * 3. abuse integer overflow/signedness bug
   AND 1. overflow code pointer (A+!)
   OR 1. overflow vulnerable stack buffer
   2. find vulnerable format string
   2. find code pointer in memory (A+!)
   3. make program jump to code pointer
   2. trigger defense mechanism
   OR 1. execute from non-exec memory
   AND 1. overwrite code pointer to point into non-exec
   OR 1. overflow vulnerable stack buffer
   2. find vulnerable format string
   2. make program jump to code pointer
   * 2. overwrite canary
   OR 1. find vulnerable stack buffer
   2. find vulnerable format string
   4. overwhelm with legitimate input

Figure 5.2: Attack tree with defensive codes. The asterisks to the left mark the nodes that are analyzed further in section 5.3.3.
(3.1) directory traversal (S-1)

System call interception will detect a directory traversal. If the interception covers arguments of system calls or if the action leads to an unknown code path, the defense will hold.

(4) make program crash

While none of the technologies will protect or detect this and it might seem to late since (4) is advertised as a goal; it is a very common precursor to intrusion and should be monitored for.

(4.1.3) abuse integer overflow/signedness bug

Such problems, for example when a negative signed integer is treated as unsigned when deciding how much to write to a buffer, tend to write enormous amounts of data. This is almost guaranteed to result in a segfault. None of the countermeasures will stop or detect this.

(4.2.2) overwrite canary

Canaries (or cookies) are a security device used to protect against buffer overflows. This is not discussed at all in this thesis but when it detects an overflow the program in question is halted. This is equivalent to making the program crash (4) which is a parent goal. A good source of information on canaries and their effectiveness has been written by John Wilander [7].
Chapter 6

Conclusions

The analysis section (5.3.3) contains many keywords like; could, assuming, but, depends, might and if. This is a direct result of the enormous amount of parameters that are involved in the security of a system as immensely complex as a computer and its programs. Outcomes of actions are rarely absolute across the multitude of different platforms and settings that are possible.

This chapter discuss this issue of uncertainty inherent in describing the security of systems with attack trees.

6.1 Experiences

It is immediately evident that countermeasures that act at a high level in the tree cover more than one that act at a low level. For example ⟨1⟩ that can prevent anomalous code from executing, cover a larger part of the tree than ⟨1.1.3⟩ that would react on code being inserted into memory. This is of course a trivial conclusion to make now but without the tree it would not be as obvious. On the other hand, a countermeasure that acts in the lower regions of the tree have effect earlier in the course of the attack although it might not be as exhaustive.

Also, groups of sub-goals that are defined as AND requirements are of special interest. If one of the AND requirements can be prevented the whole parent node is prevented as well. It is therefore preferable to block nodes that are in an AND group.

6.1.1 On uncertainty

What good is a code stating that an attack is prevented but only under some arbitrary condition? The answer to this question vary depending on the attack, defense and system being considered. An alternative would be to only record defenses that are 100 % effective but this would generally not be beneficial. In fact, not many of the defensive technologies in use are definitive, a door having a lock without known vulnerabilities to picking might instead be vulnerable to drilling. Is this door secure under the condition that a drill is not used or is this a completely different attack requiring a different countermeasure? Is the resistance to picking irrelevant because of this? This is obviously a topic of discussion but generally it is not. In this specific case, picking the lock is
preferable in many ways; less noise, almost no traces, the tools are smaller and thus easier concealed, etc. Consequently, just because there is a known bypass to a countermeasure, the countermeasure does not lose its value completely.

How should these conditions be treated then? Should the attack and bypass be splitted into two separate attacks that have to be defended against or is it just an attack with a potential bypass? The guiding principles in section 5.3.2 recommend that a bypass is listed as a separate attack. This serves to make the attack tree clearer in that more of the defense mechanisms are definite. Sometimes, though, some aspect of the bypass make it less desirable to list as a separate attack. Maybe because the bypass is unlikely to succeed or even relies on the defensive measure being there. Further, there are situations where a bypass cannot possible be considered a separate attack, e.g. using a large enough stone to break the unbreakable window.

In addition to all this, there are of course situations where a condition does not depend on a potential bypass but instead something completely different. For an example that logs are viewed occasionally. Thus, whatever opinion you have considering the bypasses, conditions have their place and will describe when a - will actually be a + in the code.

Another problem is that of the generality of defense mechanisms. Trying to place “anti-virus software” or some other technology which relies on misuse detection, would be tremendously difficult. The same problems arise with anomaly detectors. What signatures are available and what is monitored? In these cases, the general protection provided by a certain type of technology would be far to difficult to define. Instead, a specific implementation of such a technology could be considered. Although still difficult, placing the protection available from Stefan’s Anti-virus Enterprise 6.2 can be done much more accurately than “anti-virus software”.

These issues all boil down to one thing, consistency within the attack tree is of utter importance. How conditions and possible bypasses are handled as well as the defense technologies available, must be clearly defined before the work on the tree is begun.

6.1.2 Pros and cons

Below, the experiences of this exercise is summarized as a list of pros and cons on using attack trees to visualize the effects of security technologies.

Pros

- Shows where in the course of an attack it gets blocked or detected.
- Has the potential to show which technique prevents or detects the most attacks at a single point.
- Subtrees can be reused even when defensive codes are specified.

Cons

- Granular attack trees are difficult to follow in outlined format and impractically large when represented graphically.
- Quite thorough understanding of the steps involved in an attack is needed.
• How the defensive codes are set relies on many factors from both the system and the defense technology.

• Might not suitable for comparisons across different systems. (This is mainly due to the previous con.)

• The placement of a defensive code can be ambiguous.

6.2 Further work

To fully take advantage of attack trees for visualizing the effects of security technologies two areas could benefit from further work. (1) The defensive codes used here were only constructed to cover the needs of this particular dissertation. To be able to generalize their use further they need to be formalized even more. (2) Formal processes of designing attack trees have been defined by the original author already. Currently, the process of assigning defensive codes to the nodes in the tree is done in the same way. This could probably be done in a more clever way and is the part of the thesis in most need of attention.

Additionally, some formal way of annotating probabilities concerning the codes could be interesting. For example, how likely is it that the condition will hold? Should probabilities be granular, i.e. 90 %, or coarse, i.e. “very likely”?
Bibliography


Software vulnerabilities in programs and malware deployments have been increasing almost every year since we started measuring them. Information about how to program securely, how malware shall be avoided and technological countermeasures for this are more available than ever. Still, the trend seems to favor the attacker. This thesis tries to visualize the effects of a selection of technological countermeasures that have been proposed by researchers. These countermeasures: non-executable memory, address randomization, system call interception and file integrity monitoring are described along with the attacks they are designed to defend against. The coverage of each countermeasure is then visualized with the help of attack trees. Attack trees are normally used for describing how systems can be attacked but here they instead serve the purpose of showing where in an attack a countermeasure takes effect. Using attack trees for this highlights a couple of important aspects of a security mechanism, such as how early in an attack it is effective and which variants of an attack it potentially defends against. This is done by the use of what we call defensive codes that describe how a defense mechanism counters a sub-goal in an attack. Unfortunately the whole process is not well formalized and depends on many uncertain factors.

Keywords: endpoint security, attack tree, memory corruption, non-executable memory, address randomization, system call interception
Copyright

Svenska
Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enskildakopior för enskilt bruk och att använda det oförändrat för icerkommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera åktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsmann i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är krävande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English
The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Stefan Pettersson
Linköping, November 27, 2008