AUTOMATIC DETECTION OF MEMORY CORRUPTION ATTACKS

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science

by

Pankaj Kohli
200607011
`pankaj_kohli@research.iiit.ac.in`

International Institute of Information Technology
Hyderabad, India
December 2008

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Automatic Detection of Memory Corruption Attacks" by Pankaj Kohli, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Dr. B. Bruhadeshwar

To my parents,
for their everlasting love and support.

Knowing is not enough; we must apply. Willing is not enough; we must do.
- Johann Wolfgang von Goethe

Acknowledgments

I would like to thank my adviser, Dr. Bruhadeshwar for his support and guidance during the development of the ideas in this thesis, and for the helpful comments on the text. I thank Dr. Srinathan who inspired me as an individual through his various teachings during my course work at IIIT Hyderabad. I am thankful to Dr. Venkaiah for his immense support in the lab. Many thanks to my friends Sai Sathyanarayan and Vinayak Kumar, who shared their valuable ideas with me during my research work. I am grateful to all the faculty of IIIT Hyderabad whose encouragement towards research helped me contribute some good work to the research community.

Abstract

With the growth of Internet, there has been a tremendous increase in the security attacks against computer systems. Due to the improper use of programming language by programmers, software often exposes security vulnerabilities, such as buffer overflows and format string bugs. Such software vulnerabilities relating to memory safety, are the most common vulnerabilities used by attackers to gain control over the execution of a program running on a computer system. By carefully crafting an exploit for these vulnerabilities, attackers can make a privileged program transfer execution-flow to a malicious piece of code. Such memory corruption attacks are among the most powerful and common attacks against software applications. In the recent years, memory corruption attacks have accounted for more than half of all the reported CERT advisories.

A large number of defensive techniques have been described in the literature that either attempt to eliminate specific vulnerabilities entirely or attempt to combat their exploitation. The work presented in this thesis makes two significant contributions. Firstly, it presents *FormatShield*, a novel approach to defend against format string attacks. Secondly, it presents *Coarse Grained Dynamic Taint Analysis*, a generic technique that uses information flow tracking to defeat a broad range of memory corruption attacks.

FormatShield automatically identifies call sites in a running process that are vulnerable to format string attacks. Using binary rewriting, the list of exploitable program contexts at these vulnerable call sites is dumped into the program binary. Attacks are detected when malicious input is found at such call sites. FormatShield can defend against all types of format string attacks, i.e. arbitrary memory read attempts and arbitrary memory write attempts, including non-control data attacks. Coarse grained dynamic taint analysis works by labeling the data received from untrusted sources, such as network, as unsafe or *tainted*. Data derived from such tainted data is itself marked as tainted. Attacks are detected when control branches to a location specified by the unsafe data. It does not requires source code of the program to be protected and is capable of defending against a broad range of memory corruption attacks, including non-control data attacks. Also, our experiments show that it incurs modest performance overhead, making it suitable for use in production environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Software security has become an increasing necessity for guaranteeing, as much as possible, the correctness of computer systems. Unfortunately, software vulnerabilities are omnipresent. In the last few decades, many new vulnerabilities have been discovered, and old ones have been continuously exploited. Memory corruption in C and C++ programs have been known for decades and are one of the oldest classes of software vulnerabilities. Attackers have been exploiting these vulnerabilities since the days of the Internet Worm of 1988 (also known as *Morris Worm*). Despite decades of research on memory corruption attack countermeasures, these software vulnerabilities are still a real and concrete threat, as a recent breakdown of the NIST National Vulnerability Database (NVD) of software vulnerabilities depicts in Figure 1.1.

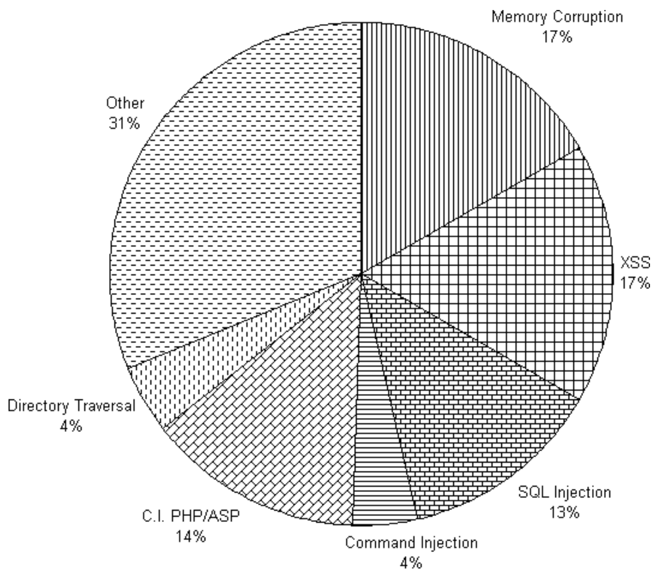


Figure 1.1: Breakdown of NIST National Vulnerability Database (NVD) of software security vulnerabilities (2006 and 2007-Q1/Q2)

Since their first public exploitation, a number of different types of memory corruption vulnerabilities have been discovered and, unfortunately, successfully exploited by attackers. To date, buffer overflows are probably the most common memory corruption vulnerability. As the name suggests, a buffer overflow vulnerability takes advantage of an erroneous or the lack of bounds checking on a buffer. As a direct consequence, buffer overflows can be exploited to write past the end of a buffer with the intent to corrupt adjacent memory locations. This carefully crafted corruption permits an attacker to generally execute arbitrary code, or, to perform actions that are as dangerous. Memory corruption can be eliminated by using type-safe languages such as Java. Unfortunately, these languages do not provide enough low-level control on memory management and data representation. These are features which are mostly required for systems software, therefore it is unlikely that these type and memory safe languages will substitute C or C++, at least in this context.

To date, a number of memory corruption attack countermeasures have been studied and proposed. For instance, to defeat stack-based buffer overflows, Cowan et al. proposed StackGuard, a compiler patch that inserts a canary value right before the return address to detect attempts to corrupt a return address using a buffer overflow. Unfortunately, not only has it been shown how to bypass StackGuard-based protections, but the offered protection is, most importantly, limited to a specific vulnerability. In fact, it has been designed to work only for stack-based buffer overflows. It has no effect, as is, for other kinds of buffer overflows, or for other memory corruption attacks, such as format strings, therefore providing a very limited degree of protection.

The countermeasure proposed by Cowan et al. is just one of several others that have kept researchers busy providing different and progressively more complete solutions to memory corruption attacks in the past several years. Proposed solutions cover a broad range of Computer Science disciplines, going from safe programming languages, anomaly detection, and information-flow, to techniques that modify the underlying compiler, system libraries, the operating system, or the hardware. Among these approaches, transformation techniques which aim to provide artificial diversity, or are based on information-flow approaches, seem to be the most promising and effective against a broad class of memory corruption attacks.

Transformation techniques that aim to provide artificial diversity to a program base their assumption on the fact that, generally speaking, memory corruption exploits leverage on the monoculture of systems. Therefore, once a memory corruption vulnerability is discovered on an application running on a particular operating system (OS) and architecture, it is fairly easy for an attacker to take over the rest of similar configurations (e.g., application, OS, architecture) which present the same vulnerability. In fact, this is possible because a process address space is organized always in the same way, for a particular OS and architecture, and memory corruption exploits rely on corrupting particular memory locations (e.g., where a function return address is stored on the stack, or where a function global offset table (`GOT`) is stored in the vulnerable process address space) with suitable values. For this reason, memory corruption exploits can be thwarted by applying approaches that use diversity on computer systems, such as address space randomization (ASR), instruction set randomization (ISR), and so on. Unfortunately, these approaches either cannot deal with all the memory corruption attacks or, even when successful, they provide only probabilistic protection.

On the other hand, information-flow (or *taint analysis*) based approaches focus their attention on how untrusted data is used and processed by a protected application (*i.e.*, how information flows into the application), to check whether these data corrupt security sensitive memory locations. More precisely, taint analysis is a technique which aims to detect whether untrusted data is incorrectly used in security sensitive actions. To this end, taint analysis usually marks untrusted data coming from taint sources (*e.g.*, data coming from the network, or other input related system calls) as being tainted, and, as data propagates through memory, it propagates the taint information associated with the data itself. Any attempt to use security sensitive data which has been marked as tainted at security sensitive points, called *sinks* (*e.g.*, particular system calls, or when a function is about to return), is generally a manifestation of an attack. For instance, a memory corruption attack which aims to corrupt a code pointer, can overwrite a function return address with the intent to hijack the legal process execution flow. Naturally, the overwritten return address will be marked as tainted as a consequence of this attack attempt. The low-level implicit policy adopted by taint-based approaches does not allow to dereference tainted code pointers, as function return addresses are security sensitive data which should never become tainted. In fact, they are manipulated by the application code which is considered to be trusted. However, current taint analysis based approaches are not very practical as they either require source code or non-trivial hardware extensions or incur prohibitive run time overheads. More importantly, they do not defend against corruption of data or data pointers, making them susceptible to privilege escalation.

Unfortunately, as protection mechanisms improve, so do the attacks. Recently, Chen et al. pointed out that it is no longer necessary to exploit a memory corruption vulnerability with the goal to hijack a legal process' control-flow to cause harm. In fact, damage can also be caused by memory corruption attacks which do not target code pointers but, instead, aim to overwrite data and data pointers. Even if this seems to be a strict restriction, Chen et al. showed that these attacks can be as powerful as the classic ones (*i.e.*, which corrupt code pointers). It may be argued that such attacks are not very common nowadays, as their exploitation requires a better understanding of the application logic. As long as memory corruption attacks which corrupt code pointers will still be effective, attackers have no motivations to make their exploit harder to code and, most of all, succeed. Naturally, should existing research countermeasures become widely used, the attackers will modify their attacks to stick to this new technique, as confirmed by recent and related research as well. Therefore, it is our belief that comprehensive memory corruption attack countermeasures should no longer ignore such a class of memory corruption attacks.

1.2 Goal of the thesis

The work proposed in this thesis aims to provide comprehensive solutions to memory corruption attacks. To this end, we propose two novel techniques which provide protection from memory corruption attacks. While, first approach aims to defend against format string attacks, the second approach provides a generic defense against a broad range of memory corruption attacks. Both approaches provide a deterministic defense against attacks which corrupt code or data pointers or arbitrary data. The underlying mechanisms of the proposed approaches are different and this can suggest contexts where the proposed strategies find, respectively, a better deployment with respect to each other. In particular:

- In our first approach, we present *FormatShield*, a novel approach that uses binary rewriting to defend against format string attacks. FormatShield identifies call sites in a running process that are vulnerable to format string attacks. Using binary rewriting, the list of exploitable contexts associated with each vulnerable call site is dumped in the program binary, and is made available at run time. Format string attacks are detected when malicious input is found at a vulnerable call site with an exploitable context. FormatShield does not require source code or recompilation of the program to be protected and is capable of defending against all types of format string attacks.

- Our second approach takes advantage of taint-tracking and anomaly detection techniques. The proposed strategy first transforms a given program binary P into P', which is P with the variables' size information, using the debugging information present in the program binary. Then, by coupling taint analysis and security policies on these variables, our approach propagates taint on these variables and dynamically analyzes sinks, that is, relevant events (*e.g.*, system calls or security sensitive functions) of the transformed binary P'. Attacks are detected when control is dictated by a tainted variable. To detect data or data pointer corruption, we check if any data has been marked as tainted in an illegitimate manner, such as by an overflow of an variable. Propagating taint on variable level instead of individual bytes of memory reduces the performance overhead considerably, making our approach practical to be used on production environments.

The thesis ends by providing experimental results and comparison to existing and similar techniques. Moreover, performances and effectiveness of these approaches are also discussed, as well as weaknesses, limitations and possible improvements.

1.3 Terminology and Conventions

- **Attack, Intrusion.** This thesis will talk about attacks and intrusions, and in this context the meaning of these words is an action, by an attacker, that aims to change the flow of control of a program; or escalate privileges without any authentication or providing sufficient credentials.
- **Attacker.** A person launching an attack.
- **Vulnerability.** A flaw or weakness in a system's or program's design, implementation or operation and management, that could be exploited.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 discusses the most important concepts about memory corruption attacks, what they are, what different kind of memory corruption vulnerabilities exist out there, and how they can intuitively be exploited. Chapter 3 describes the related work done in the the field of detection of memory corruption attacks. Chapter 4 introduces our first approach, FormatShield, followed by Coarse Grained Dynamic Taint Analysis in chapter 5. The thesis ends by giving concluding remarks and future directions that can be taken to improve the proposed approaches in Chapter 6.

Chapter 2

Preliminaries

2.1 Process Address Space

A process is a program in execution. This means that the operating system has loaded the program executable in memory, has arranged for it to have access to command-line arguments and environment variables, and has started it running. A process can have several conceptually different areas of memory allocated to it:

Code

Often referred to as the *text*, this is the area in which the executable instructions of the program reside. In operating Systems such as Linux or Unix, multiple instances of the same program share their code if possible; i.e. only one copy of the instructions for a program resides in the memory at any time. This is transparent to the running programs.

Data

Statically allocated and global data that are initialized with non-zero values reside in the *data* region of the program. Each running instance of a program has its own data region. This region starts immediately after the program code in the address space.

BSS

Block Storage Space (BSS) contains global or static data that are initialized to zero. Each running instance of a program has its own BSS region. BSS begins immediately after data region in the process address space.

Heap

The memory that is allocated at runtime is allocated from the region known as *heap*. This region is of variable size and grows as more memory is allocated. Each running copy of a program has its own heap. This region begins immediately after the BSS region in the address space.

Stack

Local variables and function arguments are stored in the program *stack*. Stack is also used for the storing some bookkeeping information such as the return address representing the return of a function to its caller. It is of variable size and grows as functions are called and space for their local variables is allocated. Stack is the only region in the process address space that grows downwards, i.e. from higher addresses towards lower addresses.

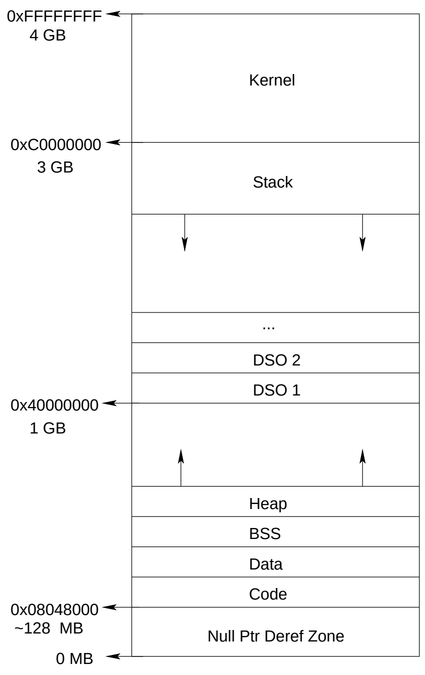
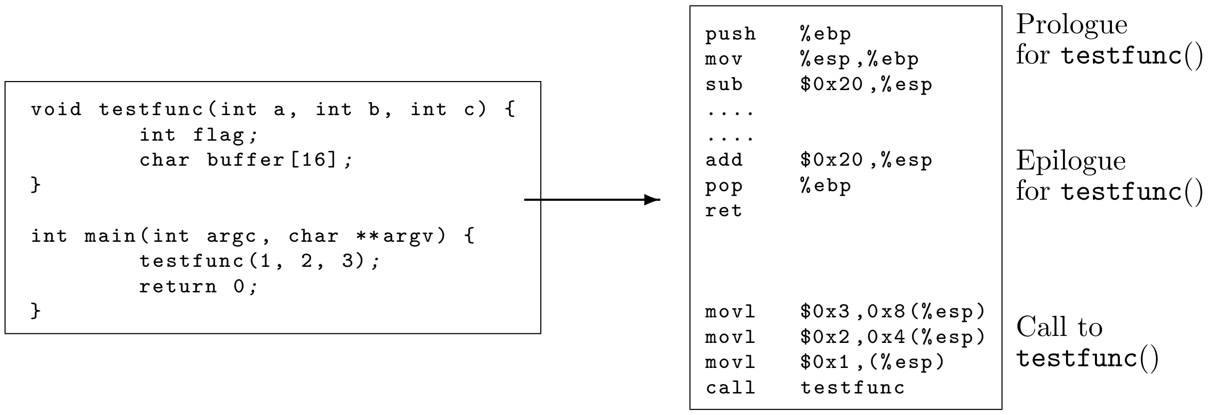


Figure 2.1: Process Address Space

Figure 2.1 shows the address space of a typical 32-bit process. The top 1 GB of the address space is used by the kernel and is not visible to the process. Stack begins immediately below the kernel and grows towards lower addresses. Dynamically Shared Objects (DSO) or shared libraries are loaded between the heap and stack regions.

2.2 Intel x86 Function Call Mechanism

As stated earlier, the stack is used for storing local variables and function arguments. The part of the stack that is used by the function being executed to hold its local variables is known as the *stack frame*. Prior to a function call, the stack pointer (or `ESP` register) points to the top of the stack and the base pointer or the frame pointer (or `EBP` register) points to the base of the stack frame. Before calling a function, the caller pushes its arguments on the stack in reverse order. When the assembly `call` instruction is executed, the return address, i.e. the address of the instruction just after the `call` instruction, is pushed on the stack and execution jumps to the callee. The return address is pushed on the stack so that the caller function can resume its execution when the callee finishes. The callee now must setup its own stack frame before manipulating any data on the stack. This mechanism is known as the *function prologue*. The prologue begins by pushing the value of frame pointer on the stack. The stack pointer is then copied to the frame pointer so that it points to the stored value of the frame pointer, which is the base of the new stack frame being created. Some space is then reserved on the stack for local variables by subtracting some offset from the stack pointer. Similarly, the callee function finishes by winding up the stack. This process is known as the *function epilogue*. The epilogue releases the space used by local variables by adding some offset to the stack pointer and restores the value of the frame pointer. The callee then uses `ret` instruction to return to its caller.



```c
void testfunc(int a, int b, int c) {
        int flag;
        char buffer[16];
}

int main(int argc, char **argv) {
        testfunc(1, 2, 3);
        return 0;
}
```

```
push   %ebp
mov    %esp,%ebp
sub    $0x20,%esp
....
....
add    $0x20,%esp
pop    %ebp
ret


movl   $0x3,0x8(%esp)
movl   $0x2,0x4(%esp)
movl   $0x1,(%esp)
call   testfunc
```

Figure 2.2: C code and the generated code showing the function prologue, epilogue and call

Consider the code fragment given in figure 2.2. It defines a function `testfunc()` that has three integer arguments, and is called from `main()`. Before the `call` is executed, the arguments are pushed on the stack in reverse order, i.e. first 3, then 2 and finally 1. Execution of `call` then causes the return address to be pushed on the stack and control branches to `testfunc()`. In `testfunc()`, the prologues first pushes the frame pointer on the stack, copies stack pointer to the frame pointer and allocates space on the stack by subtracting some offset from the stack pointer. Epilogue, on the other hand, winds up the stack, restores frame pointer and returns to the `main()`.

2.3 ELF File Format

In this section, we briefly describe the ELF[1] file format for executable, object and library files. Understanding the ELF file format is necessary to understand how our approach works. The ELF file format defines a binary interface that is descriptive enough to allow linking of several object files as well as to form a process image during execution. ELF file format recognizes the following three kinds of files:

- Relocatable files that can be linked with other relocatable files to form an executable or a shared library.
- Executable files that can be directly loaded into memory and executed.
- Shared libraries that can be linked with other relocatable files and shared libraries. This linking happens in two stages. First, the link editor, *e.g.*, `ld`, processes the shared library with other relocatable files to create an executable. Secondly, the dynamic loader, *e.g.*, `ld.so`, combines it with the executable to form the process image.

The ELF specification provides two parallel views of the object files: *linking view* and *loading view*. The linking view is needed to build the program while the loading view is required to form the process image. To support linking, the linking view divides the file into sections that contain the code, data, and the other useful information for linking such as symbol tables and relocation tables.

[1]ELF or Executable and Linking Format is the typical executable file format for Unix/Linux based operating systems.

The execution view, on the other hand, describes how various portions of the file should be mapped onto the memory while forming the process image. It divides the file contents into segments that have permissions like read/write attached to them. Both of these views are present in the same file and the ELF header provides a roadmap to describe them. The ELF header forms the first 52 bytes of the file and contains information such as the file type, position of the section header table and the program header table, the beginning of the executable code, and a 16 byte magic number for quick identification of ELF files.
We next describe the linking and loading view.

2.3.1 Linking View

The linking view divides all the contents of the file, except the ELF header, the section header table and the program header table, into various sections. These sections should satisfy the following conditions:

- Every section must have exactly one section header in the section header table. The section header table may however contain an entry that does not correspond to any section.
- Every section occupies one contiguous block of memory, possibly of zero byte length.
- No two sections overlap.
- The sections together with the header tables and the ELF header may not cover all the space in the object file.

The section header table contains an entry for every section in the object file specifying several attributes. We here describe some of these attributes.

- **Name** The name of the section. This is actually an offset in a string table that contains all the names.
- **Flags** The flags of a section describe if the contents of the section are writable, allocable (whether it will appear in the process image) or executable.
- **Address** If the section is allocable, this field determines the virtual address at which the section will be loaded.
- **Offset** This field gives the byte offset from the beginning of the file to the first byte in the section.

2.3.2 Loading View

All the ELF files ultimately represent some code to be executed and some data to be used by that code. To execute the ELF file, the operating system first "loads" the program into memory. The loading view of the ELF file contains control information for construction of this process image. At runtime, the process image is made up of segments of memory that hold the code, data, stack, etc. Each segment of the file corresponds to a segment in the virtual address space. A segment holds one or more sections. Sections with similar access permissions are grouped together to form a segment. For example, a typical executable file contains `.text` section to hold code, `.data` section to hold initialized global data and `.bss` section to hold uninitialized global data. The `.text` section and `.data` section are contained in two different segments. This is because different access permissions are required for text and data: while data section must be writable, the text section may not be writable. Because of their similar access permissions, `.data` and `.bss` sections may be a part of the same program segment.

Information about how to form the segments in memory is contained in the program header table of the ELF file. The program header table is an array of program headers, each of which describes attributes of one segment. We here describe some these attributes.

- **Type** This describes the kind of segment. For example, a value of `PT_NULL` specifies an unused entry in the program header table, a value of `PT_LOAD` specifies that the segment must be loaded into memory, `PT_DYNAMIC` specifies that the segment contains dynamic linking information, etc.
- **Offset** This gives the offset in the ELF file, of the first byte of the segment.
- **Address** This specifies the virtual address at which the segment will be loaded in memory.
- **Flags** This specifies the access permissions of the segment.

Figure 2.3 shows the linking and loading view of a typical ELF file.

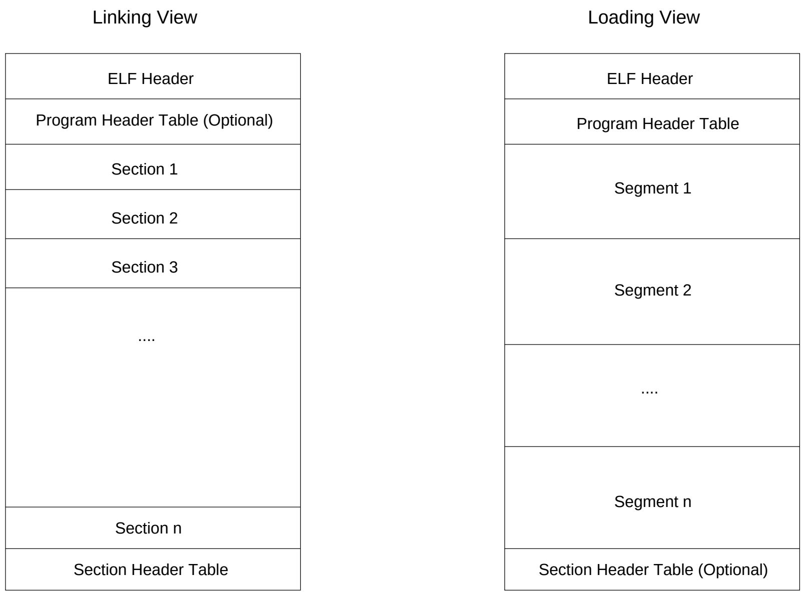


Figure 2.3: Linking and Loading view of an ELF executable

Next, we describe some special sections and how they contribute in linking an ELF file with another. A section either contains executable code, or program data to be used by code, or control information required to provide the linking view. Code resides in the following three sections:

- **.text** It contains the main executable instructions.
- **.init** It contains the executable instructions that contribute to the initialization of the process image. When a program starts to run, the code in this section of the executable is executed before the control passes to the entry point of the program, *i.e.* `main()` for C programs. In addition, the code in `.init` sections of all the dynamically loadable libraries that are linked to the executable is also executed before passing control to `main()`.
- **.fini** It contains the code to be executed when the program exits normally.

Data that is used by the code in above sections normally resides in the following sections:

- **.data** This section contains initialized global data.
- **.bss** This section contains uninitialized global data. This sections does not occupy any space in the file. The contents of the section are set to zero while forming the process image.

A number of sections are used for linking of the program. Linking two object files basically involves resolving the symbols that are defined in one object file and used in another. The symbols may either be functions or just global data. To support linking, ELF files have a special symbol table that contains a list of all the symbols used or globally defined in the file. There are typically two symbol tables *viz.* `.dynsym` and `.symtab`. `.dynsym` is used exclusively by the dynamic linker while `.symtab` contains entries for static linking.

2.3.3 Dynamic Linking

For dynamic linking, the ELF linker primarily uses two processor-specific tables, the *Global Offset Table* (contained in `.got` section) and the *Procedure Linkage Table* (contained in `.plt` section).

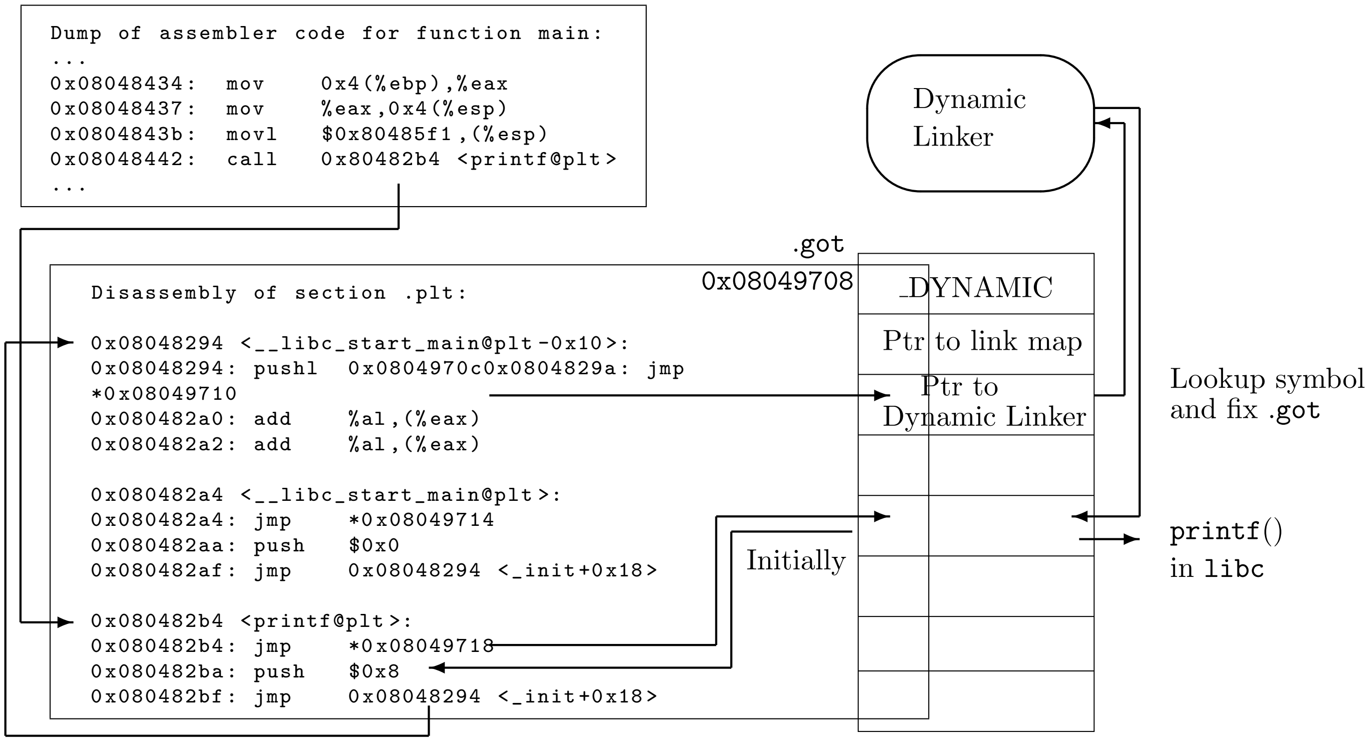
- **Global Offset Table (`.got`)** ELF linkers support position independent code (PIC) through the `.got` in each shared library. The `.got` contains absolute addresses to all of the static data referenced in the program. The address of the `.got` is normally stored in a register (`ebx`) which is a relative address from the code that references it.
- **Procedure Linkage Table (`.plt`)** Both the executables that use the shared libraries and the shared library itself has a `.plt`. Similar to how the `.got` redirects any position-independent address calculations to absolute locations, the `.plt` redirects position-independent function calls to absolute locations.

Apart from these two tables, the linker also refers to `.dynsym`, which contains all of the file's imported and exported symbols, `.dynstr`, which contains name strings for the symbols, `.hash` which contains the hash table which the runtime linker can use to lookup symbols quickly, and `.dynamic`, which is a list of tagged values and pointers. In the `.dynamic` section, the important tag types are:

- **DT_NEEDED** This element holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the `DT_STRTAB` entry.
- **DT_HASH** This element holds the address of the symbol hash table which refers to the symbol table referenced by the `DT_SYMTAB` element.
- **DT_STRTAB** This element holds the address of the string table.
- **DT_SYMTAB** This element holds the address of the symbol table.

Linux uses lazy linking to resolve function addresses, *i.e.* a function is resolved only when it is called. This is because resolving all the functions at load time imposes a performance penalty in terms of having the dynamic linker resolve functions that may not be called from the executable. Such a performance penalty becomes more evident in case of large libraries like `glibc` that contains a huge number of functions. The idea is to invoke the dynamic linker to resolve a function reference only when the function call is made. A function call, thus transfers control to the code in `.plt` which in turn does the following:

- Invoke the dynamic linker if it is the first call to this function. The dynamic linker resolves the function address and returns control to the function directly. In addition, it fixes the address of the function to be used in later calls.
- In case this is not the first call to this function, the address of the function is already available. Call the function directly.



```
Dump of assembler code for function main:
...
0x08048434:  mov    0x4(%ebp),%eax
0x08048437:  mov    %eax,0x4(%esp)
0x0804843b:  movl   $0x80485f1,(%esp)
0x08048442:  call   0x80482b4 <printf@plt>
...
```

```
Disassembly of section .plt:

0x08048294 <__libc_start_main@plt-0x10>:
0x08048294: pushl  0x0804970c0x0804829a: jmp
*0x08049710
0x080482a0: add    %al,(%eax)
0x080482a2: add    %al,(%eax)

0x080482a4 <__libc_start_main@plt>:
0x080482a4: jmp    *0x08049714
0x080482aa: push   $0x0
0x080482af: jmp    0x08048294 <_init+0x18>

0x080482b4 <printf@plt>:
0x080482b4: jmp    *0x08049718
0x080482ba: push   $0x8
0x080482bf: jmp    0x08048294 <_init+0x18>
```

Figure 2.4: Calling an externally defined function using `.plt` and .got

Now, we describe how the `plt` code invokes the dynamic linker or the function. Figure 2.4 shows a typical `plt` entry for the function `printf()` and describes how `.plt` and `.got` sections contribute to making a function call. The first three entries of the `.got` hold special values. The first entry points to the dynamic structure, i.e. the `.dynamic` section. This is because the dynamic linker needs to locate its own dynamic structure without having yet processed its relocation entries. The second `.got` entry points to the link map of the object at runtime. The link map of an ELF object is like a handle for that object available to the dynamic linker. The link map contains information like the base address at which the object is loaded, references to the dynamic structure and the symbol tables and hash tables for symbol lookup, and pointers to link maps of other objects that this object depends on. There is loads of other interesting information in the link map but we shall not delve any deeper in this. This entry is initially 0 and is set by the dynamic loader before transferring control to the program. The third entry contains the address of the function `_dl_runtime_fixup` of the dynamic linker. This function is used to relocate data at runtime. This is also set by the dynamic linker before transferring control to the program initially. The `.got` entry corresponding to `printf()` initially points to the instruction `push $0x8` in the `.plt` entry for `printf()`. When the `.plt` code gets executed for the first time, it pushes the offset of relocation entry in `.rel.plt` corresponding to `printf()` on the stack. Then it pushes a pointer to the link map and calls the dynamic linker. The dynamic linker unwinds the stack and processes the relocation request. The relocation offset points to the `.got` entry corresponding to `printf()`. The dynamic linker searches for the `printf()` in other objects defined in the link map and puts the correct address in the `.got` entry. Then, it directly passes control to the required function. As the `.got` entry is fixed, further calls to `printf()` need not go through the dynamic linker.

2.4 Memory Corruption Attacks

Software programmers often make certain assumptions of the input that is expected from the user. Memory corruption vulnerabilities arise when an external input does not meet such assumptions and is used without any validation. By providing a kind of input that the programmer did not expect, an attacker can cause the program to execute malicious code. The most commonly exploited memory corruption vulnerabilities include buffer overflows, integer overflows and format string bugs. Before discussing the tools and approaches available for detection of memory corruption attacks, it is necessary to understand these attacks. In this section, we briefly describe some of the most commonly exploited memory corruption attacks.

2.4.1 Buffer Overflows

A *buffer overflow*, or *buffer overrun*, is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer. The result is that the extra data overwrites adjacent memory locations. The overwritten data may include other buffers, variables and program flow data, and may result in erratic program behavior, a memory access exception, program termination (a crash), incorrect results or especially if deliberately caused by a malicious user - a possible breach of system security. Buffer overflows can be triggered by inputs specifically designed to execute malicious code or to make the program operate in an unintended way. As such, buffer overflows cause many software vulnerabilities and form the basis of many exploits.

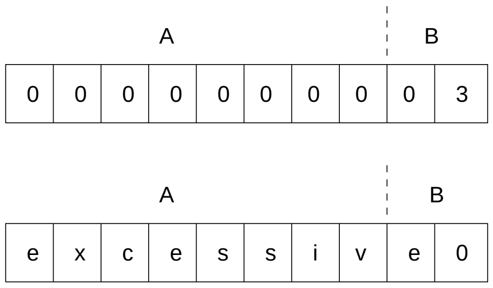


Figure 2.5: A Buffer Overflow

A buffer overflow occurs when data written to a buffer, due to insufficient bounds checking, corrupts data values in memory addresses adjacent to the allocated buffer. Most commonly this occurs when copying strings of characters from one buffer to another. In figure 2.5, a program has defined two data items which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte integer (a *short int*), B. Initially, A contains nothing but zero bytes, and B contains the number 48 (shown in the figure as `0x00` followed by `0x03`). Characters are one byte wide. Now, the program attempts to store the character string “excessive” in the A buffer, followed by a zero byte to mark the end of the string. By not checking the length of the string, it overwrites the value of B. Although the programmer did not intend to change B at all, B's value has now been replaced by a number formed from part of the character string. In this example, on a little-endian[2] system that uses ASCII, "e" (ASCII value 101 or 0x65) followed by a zero byte would become the number 101. If B was the only other variable data item defined by the program, writing an even longer string that went past the end of B could cause an error such as a segmentation fault, terminating the process.

A buffer overflow is classified as a *stack overflow* or *heap overflow* depending on the region of memory in which overflow occurs. Also, the techniques to exploit a buffer overflow vulnerability vary per architecture, operating system and memory region. For example, exploitation on the heap is very different from on the call stack. In this section, we explain different types of buffer overflows and their modes of exploitation.

Stack Overflows

A stack overflow refers to a buffer overflow in the stack region. Since stack is typically used for storage of local variables, a stack overflow occurs when a process tries to write past the end of a local variable, usually an array. The excess bytes spill over and overwrite other local variables and stored frame pointer and return address. By carefully crafting a malicious input, an attacker can overwrite the stored return address so that the new return address points to the injected code itself. This would result in execution of injected code when the function returns. If the vulnerable program is running with root privileges, the injected malicious code also executes with root privileges.

```
void func(char *str) {
        char buffer[64];
        strcpy(buffer, str);
}

int main(int argc, char **argv) {
        func(argv[1]);
        return 0;
}
```

Figure 2.6: A sample program vulnerable to stack overflow.

Consider the code fragment given in figure 2.6. Here the function `func()` copies unchecked user input to a 64 byte buffer and is therefore vulnerable to stack overflow. The stack frame of `func()` when it is called by `main()` is shown in figure 2.7. For an attacker to completely overwrite the stored return address, he must inject a 72 byte input so that it overwrites the 64 byte buffer, the stored frame pointer (4 bytes) and the stored return address (4 bytes). The crafted input usually contains the malicious code that an attacker wishes to execute, followed by the new return address. This input is copied by `strcpy()` to the local variable `buffer`, resulting in corruption of the stored return address on the stack. When the function returns, it jumps to the new stored return address which points to the injected code, resulting in execution of malicious code.

Shellcode. To accomplish something, the attack needs to execute some code to do what the attacker wants. This code is usually the code provided by the attacker and injected through function arguments, input data or system variables. The most common piece of code used is

[2]Endianness defines the byte (and bit) ordering of a multibyte data item when it is stored in memory. In *little endian* architectures such as Intel, the higher byte is stored at higher memory address and lower byte at lower memory address. In *big endian* architectures such as Motorola, the lower byte is stored at higher memory address and higher byte at lower memory address.

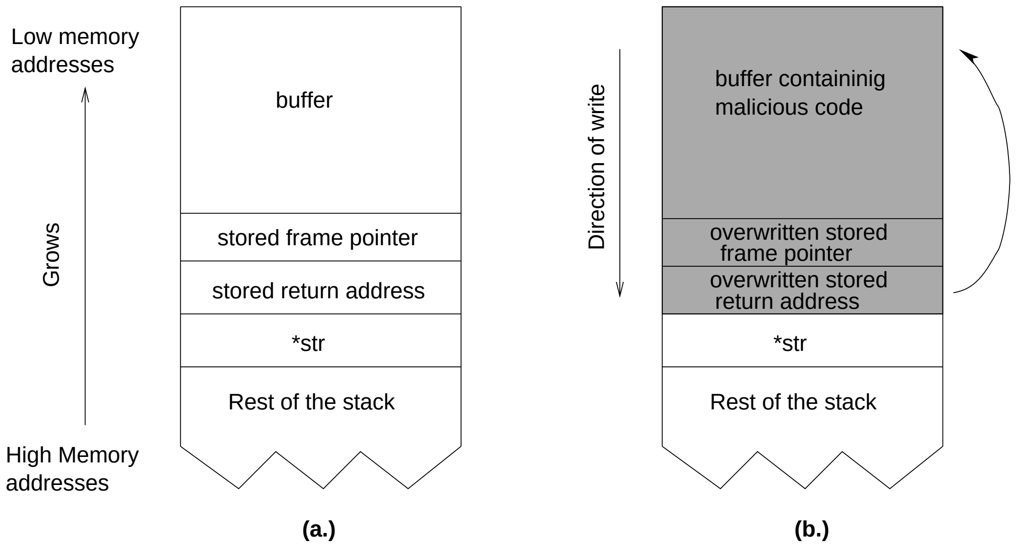


Figure 2.7: Stack frame for the function `func()` **(a.)** before and **(b.)** after `strcpy()`. An attacker injects a large input that overwrites the stored return address on the stack making it point to the injected code.

shellcode. Shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called *shellcode* because it typically starts a command shell from which the attacker can control the compromised machine. It is commonly written in machine code, but any piece of code that performs a similar task can be called shellcode.

Heap Overflows

Heap is used for dynamic allocation of memory at program runtime, and its allocation and deallocation is handled in the C library by functions such as `malloc()`, `calloc()`, `realloc()` and `free()`. The strength and popularity of heap overflow exploits comes from the way specific memory allocation functions are implemented within the individual programming languages and underlying operating platforms. Many common implementations store control data in-line together with the actual allocated memory. This allows an attacker to potentially overflow specific sections of memory in a way that these data, when later used by function such as `free()`, will allow an attacker to overwrite virtually any location in memory with the data he wants. In order to completely understand how this can be achieved, we first describe one of the most common implementations of heap-managing algorithms used by GNU C library in Linux, known as Doug Lea's malloc or *dlmalloc*.

GNU C Library Implementation. The GNU C library keeps the information about the memory slices the application requests in so called *chunks*. Each chunk consists of *boundary tags* and the memory allocated to user. Boundary tags are fields containing information about the size of this chunk and the previous chunk. Figure 2.8 (a.) shows the boundary tags of an allocated chunk. Here `P` is the pointer returned by `malloc()`. The `prev_size` element has a special function: If the chunk before the current one is unused (it was free'd), it contains the length of the chunk before. In the other case, if the chunk before the current one is used - `prev_size` is part of the *data* of it, saving four bytes. The `size` field contains the length of the current block of memory, the data section. When `malloc()` is called, four is added to the size passed to it and afterwards the size is padded up to the next double-word boundary. So a `malloc(7)` will become a `malloc(16)`, and a `malloc(20)` will become `malloc(32)`. For `malloc(0)` it will be padded to `malloc(8)`. Since this padding implies that the lower three bits are always zero and are not used for real length, they are used to indicate special attributes of the chunk. The lowest bit, called `PREV_INUSE`, indicates whether the previous chunk is used or not. It is set if the previous chunk is in use. To test whether the current chunk is in use or not, the next chunk's `PREV_INUSE` bit within its size value is checked. The second least significant bit is set if the memory area is `mmap`'ed. The third least significant bit is unused.

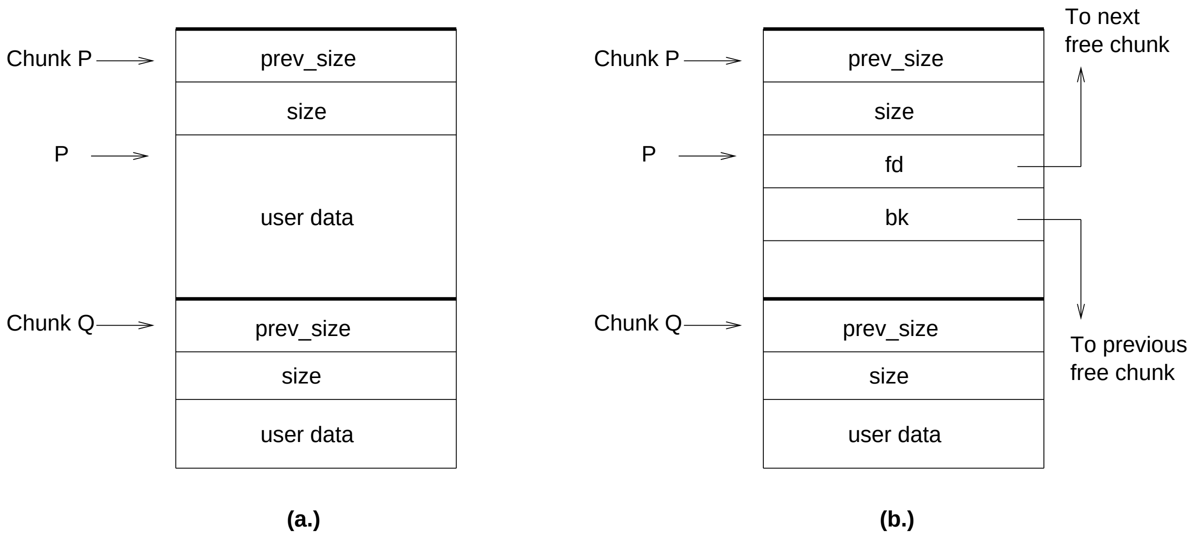


Figure 2.8: **(a.)** Allocated Chunk, and **(b.)** `free`'ed chunk

Once we `free()` the chunk, using `free(P)`, some checks take place and the memory is released. If its neighboring blocks are free too (checked using the `PREV_INUSE` flag of the next chunk), they will be merged to keep the number of reusable blocks low, but their sizes as large as possible. The resulting chunk is then placed in a *bin*, which is a doubly linked list of free chunks of a certain size. If a merge is not possible, the next chunk is tagged with a cleared `PREV_INUSE` bit, and the chunk being free'ed changes a bit. There are two new values, where the user data was previously stored. Those two values, called `fd` and `bk` - *forward* and *backward*, are pointers to the neighboring chunks in the bin. Figure 2.8 (b.) shows the modified tags of a chunk when it is free'ed. Every time a new `free()` is issued, the bin are checked, and possibly unconsolidated blocks are merged. The whole memory gets defragmented from time to time to release some memory.

When `free()` needs to take a free chunk `P` off its list in a bin, it replaces the `bk` pointer of the chunk next to `P` in the list with the pointer to the chunk preceding `P` in this list. The `fd` pointer of the preceding chunk is replaced with the pointer to the chunk following `P` in the list. The `free()` function calls the `unlink()` macro for this purpose. The `unlink()` macro is important from an attackers point of view. If we rephrase its functionality, it does the following to the chunk `P`. The address (or any data) contained in the back pointer of a chunk is written to the location stored in the forward pointer plus 12. Figure 2.9 illustrates this process. If an attacker is able to overwrite these two pointers and force the call to `unlink()`, he can overwrite any memory location with anything he wants.

```
#define unlink( P, BK, FD ) { \
        BK = P->bk; \
        FD = P->fd; \
        FD->bk = BK; \
        BK->fd = FD; \
}
```

```
*(P->fd+12) = P->bk;
*(P->bk+8) = P->fd;
```

Figure 2.9: The `unlink()` macro and its equivalent code

Exploitation. Now suppose a program under attack has allocated two adjacent chunks of memory, `P` and `Q`. Chunk `P` has a buffer overflow condition that allows an attacker to overflow the first of them (chunk `P`). Overflowing the first chunk leads to overwriting the following chunk (chunk `Q`). An attacker tries to construct the overflowing data in such a way that when `free(P)` is called, `free()` will decide that the chunk after `P` (not necessarily chunk `Q`) is free and will try to consolidate `P` and the next chunk. This is done by crafting two fake chunks, `R` and `S`, in `Q`'s memory space. Fake chunk `R` is constructed in such a way that using its size value tricks `free()` into believing that the next chunk is `S`. The fake chunk `S` is constructed in such a way that its `PREV_INUSE` bit is 0. This would trick `free()` into believing that chunk `R` is free, and it will try to consolidate chunks `P` and `R`. Now when `free(A)` is called, it will check if the next chunk is free. It will do so by looking into the boundary tag of first fake chunk `R`. The size field from this tag will be used to find the next chunk, which is also constructed by the attacker. The next fake chunk `S`'s `PREV_INUSE` bit is 0, so the function will decide that `R` is free and will call `unlink(R)`. This will result in the desired location being overwritten with the data specified by an attacker. Figure 2.10 shows this process.

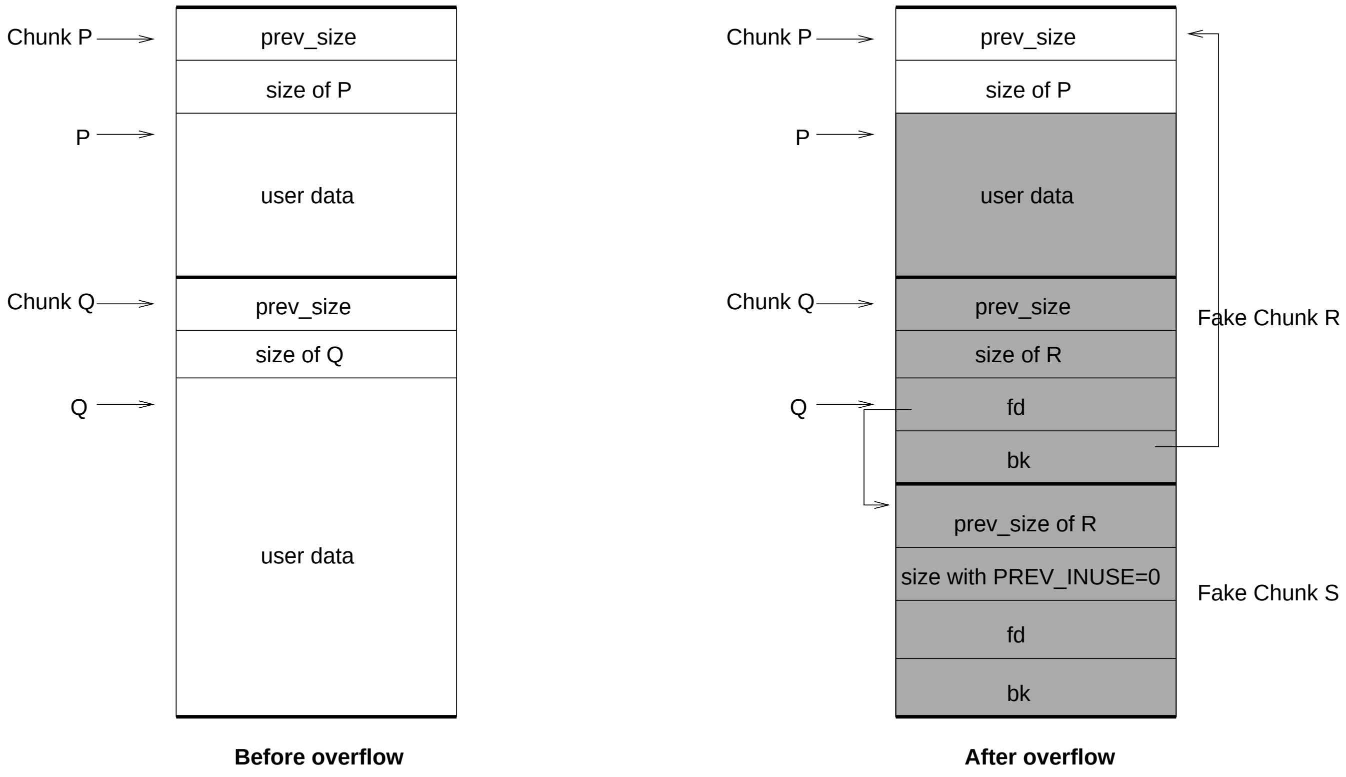


Figure 2.10: An attacker constructs fake chunks by overflowing a chunk

2.4.2 Format String Attacks

Format-string exploits are a relatively new class of exploit. Like buffer-overflow exploits, the ultimate goal of a format-string exploit is to overwrite data in order to control the execution flow of a privileged program. Format-string exploits also depend on programming mistakes that may not appear to have an obvious impact on security. Luckily for programmers, once the technique is known, it's fairly easy to spot format-string vulnerabilities and eliminate them. But first some background on format strings is needed.

Format Strings and `printf()`. Format strings are used by format functions, like `printf()`. These are functions that take in a format string as the first argument, followed by a variable number of arguments that are dependent on the format string. Since the number of arguments can vary, at program run time, there is no way for the format function definition to know the number and type of arguments passed to it. To overcome this, the format string specifies the number and type of arguments that follow. The format string consists of a number of format specifiers such as `%s`, `%d`, `%c`, etc., that specify how the corresponding argument is to be interpreted. For example, in `printf("%d %s", a, b)`, the format string `"%d %s"` specifies that two arguments follow the format string argument, since there are two format specifiers `%d` and `%s`. It also specifies that the first argument `a` is to be interpreted as an unsigned integer and the second `b` is to interpreted as an address of a string. The format function simply evaluates the format string passed to it and performs a special action each time a format parameter is encountered. Each format parameter expects an additional variable to be passed, so if there are three format parameters in a format string, there should be three additional arguments to the function, in addition to the format-string argument.

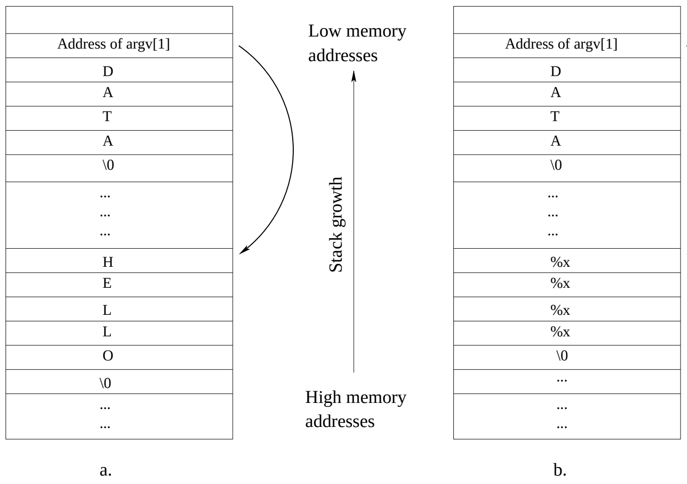


Figure 2.11: Stack layout when `printf()` is called. **(a.)** On giving a legitimate input, the program prints HELLO. **(b.)** On giving a malicious input ("%x%x%x%x"), the program prints 44415441 (hex equivalent of DATA).

Exploitation. Format string vulnerabilities occur when programmers pass user supplied input to a format function as the format string argument, *i.e.* using code constructs such as `printf(argv[1])` instead of `printf("%s", argv[1])`. The format string argument is scanned by the format function for the presence of format specifiers, and for each format specifier an argument is picked from the stack and processed. The way an argument is processed depends on the format specifier. To exploit a format string vulnerability, a malicious user can insert format specifiers in the input. For example, passing a string of `%x` or `%s` format specifiers would cause the above vulnerable call to dump contents of the stack. Figure 2.11 illustrates this process. Using direct parameter access, *i.e.* `%N$d` allows an attacker to access the N^{th} argument on the stack without accessing the previous $N-1$ arguments. The value of N is so specified by an attacker such that the corresponding address is picked from a location in a stack the contents of which is controlled by the attacker, often the format string itself. This allows an attacker to read any memory location within the address space of the process. The most common form of the attack makes use of `%n` format specifier, the purpose of which is to write the number of bytes printed so far to the address of an integer specified. For example, `printf("ABCD%n", &x)` will write 4 to `x`. The number of bytes printed so far can easily be controlled by printing an integer with a large amount of padding such as `%656d`. Using a `%n` format specifier, an attacker can overwrite the stored return address on the stack with the address of his own code, taking control of the program when the function returns.

2.4.3 Integer Overflows

An integer, in the context of computing, is a variable capable of representing a real number with no fractional part. Integers are typically the same size as a pointer on the system they are compiled on (*i.e.* on a 32-bit system, such as i386, an integer is 32-bits long, on a 64-bit system, such as SPARC, an integer is 64-bits long). Since it is often necessary to store negative numbers, there needs for a mechanism to represent negative numbers. The way this is accomplished is by using the most significant bit (MSB) of a variable to determine the sign: if the MSB is set to 1, the variable is interpreted as negative; if it is set to 0, the variable is positive. Since an integer is a fixed size, there is a fixed maximum value it can store. When an attempt is made to store a value greater than this maximum value, it is known as an *integer overflow*. Integer overflows cannot be detected after they have happened, so there is no way for an application to tell if a result it has calculated previously is correct. This can be dangerous if the calculation has something to do with the size of a buffer or how far into an array to index. Most integer overflows are not exploitable because memory is not being directly overwritten, but sometimes they can lead to other classes of bugs, frequently buffer overflows.

Since an integer overflow is the result of an attempt to store a value in a variable which is too small to hold it, the simplest example of this can be demonstrated by assigning the contents of large variable to a smaller one. Such an overflow is known as *width overflow*. If an attempt is made to store a value in an integer which is greater than the maximum value the integer can hold, the value will be truncated. If the stored value is the result of an arithmetic operation, any part of the program which later uses the result will run incorrectly as the result of the arithmetic being incorrect. Such overflows are known as *arithmetic overflows*. Another class of bugs called *signedness bugs* occur when an unsigned variable is interpreted as signed, or when a signed variable is interpreted as unsigned. This type of behavior can happen because internally to the processor, there is no distinction between the way signed and unsigned variables are stored. Signedness bugs can also be caused due to integer overflows. This can happen when an integer overflows so that it wraps around to a negative number. Figure 2.12 illustrates these bugs.

```
int a = 0x12345678;
char b;
b = a;
```

```
unsigned int a = 0xffffffff;
a++;
```

```
unsigned int a = 0x7fffffff;
unsigned int b = 0x7fffffff;
unsigned int c = a+b;
```

Figure 2.12: Different types of integer overflows - a width overflow, an arithmetic overflow and a signedness bug caused due to an arithmetic overflow

2.4.4 Double Free Attacks

Another possibility of exploiting memory managers in `dlmalloc` arises when a programmer makes the mistake of freeing the pointer that was already freed. In the case of a double-free error, the ideal exploit conditions are as follows. A memory block `A` of size `S` is allocated. It is later freed as `free(A)`, and forward or backward consolidation is applied, creating a larger block. Then a larger block `B` is allocated in this larger space. `dlmalloc` tries to use the recently freed space for new allocations, thus the next call to `malloc()` with the proper size will use the newly freed space. An attacker-supplied buffer is copied into `B` so that it creates an "unallocated" fake chunk in memory after or before the original chunk `A`. The same technique described earlier is used for constructing this chunk. The program calls `free(A)` again, thus triggering the backward or forward consolidation of memory with the fake chunk, resulting in overwriting the location of an attacker's choice.

2.4.5 Globbing Vulnerabilities

`glob()` implements filename pattern matching, following rules similar to those used by Unix shells. It is a pathname generator, which accepts an input pattern representing a set of filenames and returns a list of accessible pathnames matching that pattern. The input pattern is specified by using special metacharacters, taken from the following: $*?[]\{\}\sim'$. For example, a pattern of '/e*' would match all directories and files in the root of the file system that begin with the character 'e'. The ability of a remote or local user to deliver input patterns to `glob()` implementations allows for two general types of security exposures.

1. **`glob()` expansion vulnerabilities** A number of vulnerabilities result from a program assuming that the length of the user input is limited to the number of characters that are read in from an external source. This assumption is problematic because most such programs, *e.g.* FTP daemons, contain a parser rule for processing pathnames beginning with a tilde ($\sim$). The intended effect of this rule is to replace the tilde directory component with the referenced home directory. However, since this is performed by running the string through the `glob()` function, the vulnerable program will also expand any other wildcard characters present. This allows for user input that can exceed the number of characters read in from the external source, which can make otherwise benign unbounded string operations exploitable.

2. **`glob()` implementation vulnerabilities.** Certain `glob()` implementations contain buffer overflows in their internal utility functions. These overflows are typically triggered by requesting a pattern that expands to a very large pathname, or by submitting a pattern that the user intends to have the vulnerable program run through `glob()` several times.

2.5 Attack Targets

In all memory corruption attacks, an attacker gains control of the execution of a process by overwriting a critical structure in memory. In this section, we describe the targets or the critical structures that can be overwritten to exploit a memory corruption vulnerability.

Stored Return Address. Any function that is called must return to the caller, under normal circumstances. To return to the caller, an `call` instruction pushes the return address on the stack and jumps to the callee. The return address is the address of the instruction following the `call` instruction. Overwriting the stored return address on the stack will cause execution to return to a different address when the function returns. An attacker can overwrite the stored return address on the stack with the address of his own code, which will execute his code when the function returns.

Stored Frame Pointer. Frame pointer or the `EBP` register points to the base of the current stack frame. When a function is called, it saves the old frame pointer on the stack before making it point to the base of current stack frame. By overwriting a stored frame pointer an attacker can set up a fake stack frame, which would be used for referring to local variables and function arguments when the function currently executing returns. Thus he can cause the program execute with arbitrary data.

Function Pointers. A program may use function pointers that would be dereferenced at run time. By overwriting a function pointer, an attacker can make execution to jump to his own code next time the function is invoked using the function pointer.

Longjmp Buffers. Programs may use functions that save stack context in a *jump buffer* for non-local jumps, such as `setjmp()` and `sigsetjmp()`. Such functions are used for dealing with errors and interrupts. Later, if the program encounters an error, it can use `longjmp()` or `siglongjmp()` to return to the saved context. An attacker can overwrite the jump buffer with his own data, which can allow him to control the execution when the program tries to restore the context.

Global Offset Table (`GOT`) entries. Global Offset Table (`GOT`) is used to hold addresses of library functions. For each library function that the program uses, there is an entry in the `GOT` that holds its address. Since, these addresses are resolved at program run time, and dynamic linker writes addresses of the resolved functions in the `GOT` so that it need not be resolved again, `GOT` is made writable. An attacker can overwrite an entry in the `GOT` with the address of his own code, so that when the corresponding library function is called, the execution will jump to attacker's code. For example, overwriting the `GOT` entry for `printf()` will cause the execution to jump to the specified address when `printf()` is called.

`.dtors` entries. *Constructors* are functions that are called before `main()` starts executing. *Destructors* are functions that are called when `main()` returns. Every program, irrespective of whether it has defined any contructors or destructors, has a `.ctors` and a `.dtors` section, which are tables that holds addresses of defined contructors and destructors respectively. Also, these sections are a part of data segment at run time and are therefore marked as writable. By overwriting an entry in `.dtors` table with the address of his own code, an attacker can cause the program to execute his own code when `main()` returns.

2.6 Attack Variations

In real-world exploits there are a variety of issues which need to be overcome for exploits to operate reliably. Null bytes in addresses, variability in the location of shellcode, differences between different environments and various counter-measures in operation make it harder for an attacker to exploit a vulnerability. To overcome these, attackers make use of several techniques to make an exploit reliably. In this section, we discuss some of these techniques.

2.6.1 Return into `libc` Attacks

Most applications never need to execute anything on the stack, so an obvious defense against buffer overflow exploits is to make the stack non-executable. When this is done, shellcode existing anywhere on the stack is basically useless. This type of defense will stop the majority of exploits out there, and it is becoming more popular. The latest version of OpenBSD has a non-executable stack by default. Of course, there is a corresponding technique that can be used to exploit programs in an environment with a non-executable stack. This technique is known as *returning into libc*. `Libc` is a standard C library that contains various basic functions, like `printf()` and `exit()`. These functions are shared, so any program that uses the `printf()` function directs execution into the appropriate location in `libc`. An exploit can do the exact same thing and direct a program's execution into a certain function in `libc`. The functionality of the exploit is limited by the functions in `libc`, which is a significant restriction when compared to arbitrary shellcode. However, nothing is ever executed on the stack.

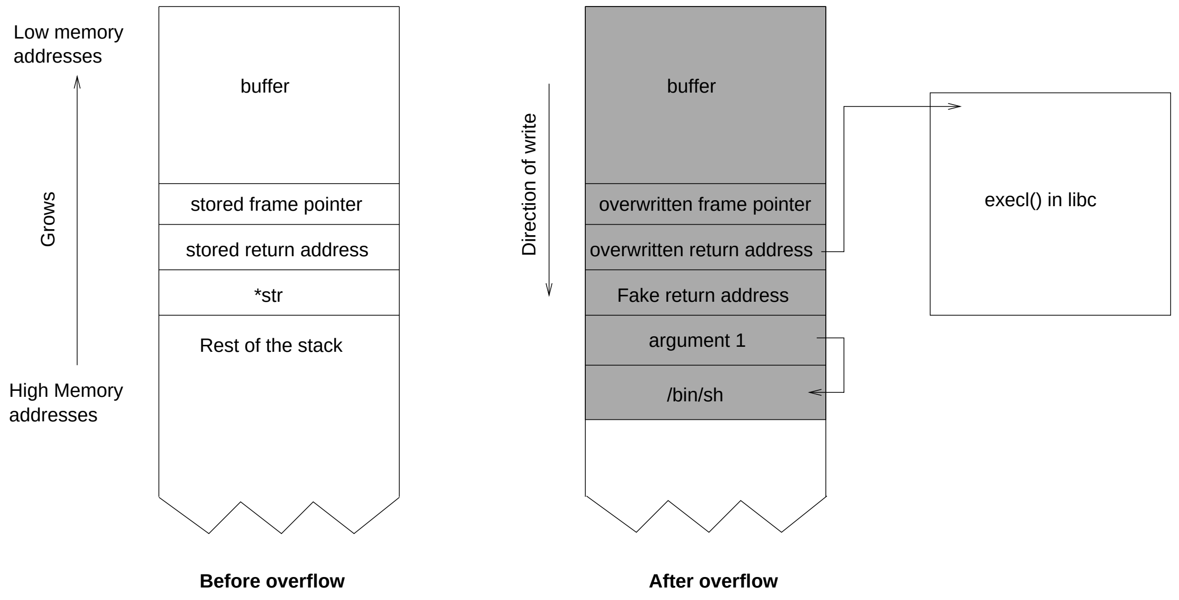


Figure 2.13: A return into `libc` attack

Figure 2.13 illustrates a return into `libc` attack. An attacker overflows a stack buffer, overwriting the stored return address with the address of `libc` function `execl()`. The input is constructed in the following way: the buffer, followed by the address of the library function which would overwrite the stored return address, followed by a fake return address, followed by the arguments to the `libc` function. The fake return address is required since a function definition assumes there is a return address stored on the top of the stack, which would be used to return to the caller. The arguments are still present on the stack but no code is executed on the stack. The example shown here calls only a single function in `libc`. It is even possible to *chain* return to several `libc` functions. This is accomplished by using the address of another `libc` function instead of the fake return address so that the first `libc` call returns to the second one. However, we shall not delve any deeper into this.

2.6.2 NOP Sled

`NOP` is a single byte instruction that does absolutely nothing. These are sometimes used to waste computational cycles for timing purposes and are actually necessary in the Sparc processor architecture due to instruction pipelining. A `NOP`-sled is the oldest and most widely known technique for successfully exploiting a stack buffer overflow. It solves the problem of finding the exact address to the buffer by effectively increasing the size of the target area. To do this much larger sections of the stack are corrupted with the `NOP` machine instruction. By creating a large array (or sled) of these `NOP` instructions and placing it before the shellcode, if the execution returns to any address found in the `NOP` sled, the `EIP` will increment while executing each `NOP` instruction, one at a time, until it finally reaches the shellcode. This means that as long as the return address is overwritten with any address found in the `NOP` sled, the `EIP` will slide down the sled to the shellcode, which will execute properly. This technique requires the attacker to guess where on the stack the `NOP`-sled is instead of the comparatively small shellcode. Figure 2.14 illustrates the `NOP`-sled technique.

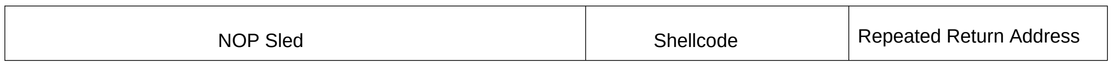


Figure 2.14: `NOP` Sled technique

While this method greatly improves the chances that an attack will be successful, it is not without problems. Exploits using this technique still must rely on some amount of luck that they will guess offsets on the stack that are within the `NOP`-sled region. An incorrect guess will usually result in the target program crashing and could alert the system administrator to the attacker's activities. Another problem is that the `NOP`-sled requires a much larger amount of memory in which to hold a `NOP`-sled large enough to be of any use. This can be a problem when the allocated size of the affected buffer is too small and the current depth of the stack is shallow (*i.e.* there is not much space from the end of the current stack frame to the start of the stack). Despite its problems, the `NOP`-sled is often the only method that will work for a given platform, environment, or situation; as such it is still an important technique.

2.6.3 Jump to Register

The *jump to register* technique allows for reliable exploitation of stack buffer overflows without the need for extra room for a `NOP`-sled and without having to guess stack offsets. The strategy is to overwrite the return pointer with something that will cause the program to jump to a known pointer stored within a register which points to the controlled buffer and thus the shellcode. For example, if register `EBP` contains a pointer to the start of a buffer then any jump or call taking that register as an operand can be used to gain control of the flow of execution. In practice, a program may not intentionally contain instructions to jump to a particular register. The traditional solution is to find an unintentional instance of a suitable opcode at a fixed location somewhere within the program memory. Figure 2.15 shows an example of such an unintentional instance of the i386 `jmp esp` instruction on a Win32 environment. The opcode for this instruction is `FF E4`. This two byte sequence can be found at a one byte offset from the start of the instruction `call DbgPrint` at address `0x7C941EED`. If an attacker overwrites the program return address with this address the program will first jump to `0x7C941EED`, interpret the opcode `FF E4` as the `jmp esp` instruction, and will then jump to the top of the stack and execute the attacker's code. When this technique is possible, the severity of the vulnerability increases considerably. This is because exploitation will work reliably enough to automate an attack with a virtual guarantee of success when it is run. For this reason, this is the technique most commonly used in Internet worms that exploit stack buffer overflow vulnerabilities.

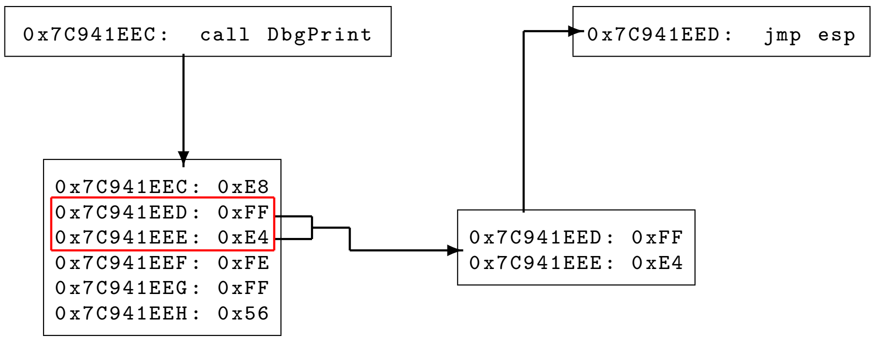


Figure 2.15: Jump to register technique

2.6.4 Code Spraying Attacks

Many operating system distributions offer randomization techniques that introduce uncertainty in the addresses used by a process from one execution to the next. Such randomization make it harder for an attacker to guess the correct address, which is the fundamental requirement for any kind of exploit, offering a generic defense mechanism to defeat any kind of memory corruption attack. To defeat such randomization, attackers are increasingly using *code spraying attacks*. In many applications, such as web browsers, the amount of memory allocated can be controlled by the remote user. In such a case, a malicious user can request a huge amount of memory and replicate the NOP-sled followed by shellcode in this memory several times. An attacker then triggers the vulnerability, causing the execution to jump into one of this NOP-sleds, thereby increasing the chances of exploitation considerably. For example, an *in-the-wild* exploit of JView Profiler vulnerability utilizes a code spraying attack as follows: A javascript-based code snip in the malicious web page first prepares a basic memory block of 256K bytes containing a large NOP-sled and a particular shellcode. This block is then replicated to 750 other memory blocks. As a result, the shellcode (including the NOP-sled) is sprayed all over the allocated heap space of 187.5M bytes. It then triggers the JView Profiler vulnerability (MS05-037), which results in the execution of the shellcode located somewhere in the allocated heap space. Note that randomization schemes can make the actual location of the injected shellcode (contained in the allocated heap space) hard to predict. However, the code-spraying attack is able to overcome this challenge by populating the shellcode in a large memory space. As long as the overwritten code pointer (e.g., return address) points to somewhere inside this large memory space, the shellcode will eventually get executed. The probability that the overwritten code pointer points to one of the NOP-sled is given by $750 * 256K/2^{32} = 4.6\%$, which has a very low entropy (4-bits if taking into account that the Windows kernel occupies the upper half of the address space).

2.6.5 Non-Control Data Attacks

Almost all the exploits that have been witnessed so far overwrite a control structure in memory, such as stored return address on the stack, GOT entries, function pointers, etc. However, it is possible to overwrite a program specific security critical non-control data using the same vulnerability.

Non-control data attacks can corrupt a variety of application data including user identity data, configuration data, user input data, and decision-making data. Such non-control data attacks have been studies in the past on several real world applications, such as FTP, Telnet, SSH and HTTP servers. The success of these attacks and the variety of applications and target data suggest that potential attack patterns are diverse. Attackers are currently focused on control-data attacks, but it is clear that when control flow protection techniques shut them down, they have incentives to study and employ non-control-data attacks.

Chapter 3

Related Work

Various methods for finding and preventing memory corruption attacks such as buffer overflows and format string attacks have been developed, including policy techniques, such as enforcing secure coding practices and mandating the use of safe languages, halting exploits via operating system extensions, statically analyzing source code to find potential vulnerabilities and detecting attacks at runtime. Each approach has its advantages; however, each also suffers from limitations. This chapter discusses some of the available approaches and highlights their strengths and weaknesses.

3.1 Security Policies and Code Reviews

Due to a large number of security vulnerabilities in software many companies have devoted extensive resources to ensuring security. Considerable time and effort is expended on implementing and enforcing secure coding procedures, following proper software development cycle and conducting code reviews. Microsoft has announced its renewed focus on security and claims to have performed a large-scale review of its Windows codebase as part of its Trustworthy Computing initiative. This code review involved up to 8,500 developers and lasted several months. However, bugs remain in Windows code, as evidenced by security patches for Windows Vista, released after the security review. This anecdote demonstrates that it is very difficult to verify by inspection that a complex piece of software contains no faults, and thus code reviews, no matter how thorough, will miss bugs. While policies and procedures encouraging secure coding and proper development practices aid in developing secure software, they cannot replace testing or guarantee the security of resulting code.

Some programs in the family of Unix operating systems require more privileges than the user that is executing them. These are called privileged programs and are generally executed with the full privileges of the owner of that program. However the lack of granularity that is available brings about a major security problem: programs that require administrator privilege to execute a specific system call can execute any other calls with the same privileges. This can cause problems if an attacker is able to inject foreign code into the application and forces the program to execute this using one of the memory corruption attacks. **Systrace** [50] introduces a way of eliminating the need to give a program full administrator privileges, instead allowing finer grained privileges to be given to an application: which system calls can be executed and with which arguments. As defining a policy of which system calls a program is able to execute is complex, Systrace offers a training ability, allowing a user to run the program while Systrace learns which system calls are executed. This allows Systrace to generate a base policy that can later be refined. It is also possible for Systrace to interactively generate system policies, where the user has to make a policy decision whenever an attempt to execute a system call that is not described by the current policy is performed.

3.2 Language Approach

An alternative that guarantees absence of buffer overflows is writing code in a *safe* language. Languages such as Java provide built-in bounds checking for arrays and buffers, and thus programs written in Java are immune to buffer overflows, forgoing errors in JVM implementation. However, large legacy codebases have been written in C; furthermore, many developers are opposed to switching to a different language, commonly citing worse performance as the reason. Efforts have been made to create languages based on C that provide similar capabilities. CCured and Cyclone are good examples of such languages.

CCured [45] works by executing a source to source transformation of C code and then compiling the resulting code with `gcc`. CCured performs a static analysis to determine pointer types (SAFE, SEQ or WILD). Instrumentation based on pointer type is inserted into the resulting source code whenever CCured cannot statically verify that the pointer will always be within the bounds of the buffer. This instrumentation changes pointer representation, so code compiled with CCured cannot interoperate easily with code compiled with gcc - wrapper functions are needed to add or remove metadata used by the instrumentation whenever an object passes between instrumented and uninstrumented code. Since CCured compiler cannot handle large programs, the developer may be required to re-write or annotate 2-3% of the code in order to enable CCured to perform the necessary transformations. At runtime, the resulting executable performs bounds-checking and terminates program with an error message if a buffer access violation has been detected.

Cyclone [36] is a dialect of C that has been developed to provide bounds-checking for C-like programs. Like CCured, it uses a combination of static and dynamic analysis to determine pointer types and only instruments pointers that require it. In order to make proper bounds information available at runtime, Cyclone changes Cs representation of pointers to include additional information, such as bounds within which a pointer is legal. A major disadvantage of Cyclone is that existing C code needs to be ported to the Cyclone language before it can be compiled with the Cyclone compiler. In order to convert the code, a developer is required to rewrite about 10% of the code, inserting special Cyclone commands. Furthermore, Cyclone differs enough from C that some C programmers may be reluctant to use it.

The main disadvantage of such safe languages is that they tend to be slower than C, and provide less control to the programmer. Another concern is the difficulty of translating the large amount of legacy code still in use today that has been written in C into a safe language. A good example is sendmail, in which multiple buffer overflows have been found over the years. The effort to translate such programs into a safe language would be immense and the cost unacceptable. Some programs, such as CCured, aim for an automatic source to source transformation of C into a safe language; however, they generally require programmer intervention and fail to run successfully on large programs.

3.3 Safe C Libraries

A large class of overflows resulting in exploits is enabled by unsafe C library functions. Many functions in the C library are written to provide fast performance and relegate the responsibility of calculating appropriate buffer bounds to the programmer. Frequently, programmers miscalculate these bounds or else do not realize that the library function can overflow the buffer. Several alternative libraries, that claim to provide a *safe* implementation of functions in the C library have been developed; some are discussed here. **Libsafe** [62] is a dynamically loaded library that intercepts calls to functions in glibc that are considered dangerous. These routines include `strcpy()`, `strcat()`, `getwd()`, `gets()`, `scanf()`, `realpath()`, `sprintf()` and others. The call to one of these routines is replaced with a wrapper function that attempts to compute the maximum size for each local buffer. The size of a buffer is computed as the distance from the start of the buffer to the saved frame pointer above it. If the size limitation allows it, the intercepted function is executed normally; otherwise, the program is terminated, thus turning a buffer overflow into a denial of service attack. Although Libsafe protects against a stack-smashing attack, it does not prevent overwriting of local variables, which can be used to change a code pointer indirectly. This method also fails if the executable has been compiled with `-fomit-frame-pointer` option, as the frame pointer is no longer saved on the stack. This option is frequently enabled during compilation to increase performance in the resulting executable.

Libverify [13] is an enhancement of Libsafe. During each function call, the return address is copied onto a special *canary stack* stored on the heap. When the function returns, the return address on the stack is compared to the address on the *canary stack* - if the two match, the program is allowed to proceed; otherwise, the program is terminated and an alert is issued. Libverify protects against a larger class of attacks than Libsafe; however, attacks using function pointers still remain undetected and lack of protection for the *canary stack* on the heap is a major limitation.

TIED-LibsafePlus [10] is another enhancement of Libsafe. TIED (Type Information Extractor and Depositor) extracts size information of all global and automatic buffers defined in the program from the debugging information produced by the compiler and inserts it back in the program binary as a data structure available at runtime. LibsafePlus is a dynamic library that provides wrapper functions for unsafe C library functions such as `strcpy()`. These wrapper functions check the source and target buffer sizes using the information made available by TIED and perform the requested operation only if it is safe to do so. For dynamically allocated buffers, the sizes and starting addresses are recorded at run time.

LibFormat [53] works by intercepting calls to `printf` family of functions, and aborts any process if the format string is writable and contains `%n` format specifier. This technique is quite effective in defending against real format string attacks, but in most cases writable format strings containing `%n` format specifier are legal, and consequently it generates many false alarms.

3.4 Operating System Extensions

Various methods for thwarting exploits based on buffer overflows using operating system extensions have been proposed and deployed. A commonly used tactic is inducing entropy in the process address space and the underlying instruction set. Several methods based on address space and intruction set randomization have been proposed in the past years to defend against a wide range of memory corruption attacks. This section discusses some of the approaches.

3.4.1 Randomized Addresses

Most exploits expect the memory segments to always start at a specific address and attempt to overwrite the return address of a function, or some other interesting address with an address that points into their own code. If the base address is generated randomly when the program is executed, it is harder for the exploit to direct the execution-flow to its injected code.

PaX [3] is a patch for the Linux kernel that implements least privilege protections for memory pages. It offers prevention against unwarranted code execution via memory management access controls (NOEXEC) and address space layout randomization (ASLR). The NOEXEC component of PaX aims to prevent the injection and execution of arbitrary code in an existing process memory space. The NOEXEC implementation consists of three features that ultimately apply access controls on mapped pages of memory. The first feature of NOEXEC applies executable semantics to memory pages. Executable semantics can be thought of as applying least-privilege concepts to the MMU. The application of these semantics to create non-executable pages on the IA-32 architecture can take two forms, based on the paging (PAGEEXEC) and segmentation logic (SEGMEXEC) of IA-32. Once the logic required to create non-executable pages has been merged into the kernel, the next step is to apply the new features. This can be done by making the memory that holds the stack, heap, anonymous memory mappings and any section not specifically marked as executable in an ELF file, non-executable by default. Finally, the functionality of `mmap()` and `mprotect()` are modified to prevent the conversion of the default memory states to an insecure state during execution (MPROTECT). PAGEEXEC is an implementation of non-executable pages, which is derived from the paging logic of IA-32 processors. The IA-32 family of processors lack native hardware support for marking pages of memory non-executable. However, the implementation of a split Translation Lookaside Buffer (TLB) in Pentium and AMD K7+ CPUs can be leveraged to emulate non-executable page support. The purpose of the TLB is to provide a cache for virtual-to-physical address translation, which speeds up instruction or data fetching within the CPU. A split TLB actually has two separate translation buffers, one for instruction fetches (ITLB) and one for data fetches (DTLB). The ITLB/DTLB loading is the key feature to getting non-executable pages, as protected pages can be marked as either *non-present* or *requiring supervisor level access.* In both cases, access to the pages will generate a page fault. The page fault handler can then decide if it was an instruction fetch or data access. If it is an instruction fetch, it means that there was an execution attempt in a non-executable page, and the process can then be terminated accordingly. However, if the fault is triggered during data access, the pages can be changed temporarily to provide user-level access and then restored to enable the fault handler for future accesses. SEGMEXEC is an alternate implementation of non-executable pages that is derived from the segmentation logic of IA-32 processors. Linux runs in protected mode with paging enabled on IA-32 processors, which means that each address translation requires a two-step process. The logical address must first be converted to a linear address from which the correct physical address may be determined. This is usually transparent to users of Linux, primarily because it creates identical segments for both code and data access that cover the range of `0x00000000 - 0xffffffff` and does not require translation between logical and virtual memory addresses because they share the same value. PaX leverages the segmentation logic to create separate address ranges for the data (non-executable) and code segments. The 3 GB of userland memory space is divided in half, and each segment is assigned one of the halves. The data segment lies in the `0x00000000 - 0x5fffffff` range and the code segment lies in the `0x60000000 - 0xbfffffff` range. Since the code and data segments are separated, accesses to the memory ranges can be monitored by the kernel and a page fault is generated if instruction fetches are initiated in the non-executable pages. Address Space Layout Randomization (ASLR) is the concept that attempts to render exploits that depend on predetermined memory addresses useless by introducing a certain amount of randomness to the layout of the virtual memory space. By randomizing the locations of the stack, heap, loaded libraries and executable binaries, ASLR effectively reduces the probability that an exploit that relies on hardcoded addresses within those segments will successfully redirect code execution to the supplied buffer. For the Intel x86 architecture, PaX ASLR provides 16, 16 and 24 bits of randomization for the executable, mapped and stack areas respectively. However, many successful derandomization attacks against PaX have been studied in the past [56, 28], that defeat it in a matter of minutes.

Bhatkar et al describe **Address Obfuscation** [16, 17] techniques that make it harder for an attacker to exploit one of the vulnerabilities described earlier. But as opposed to PaX the implementation is not done at the kernel level, but at the library level. They too support the same kind of randomization that PaX ASLR does, but they extend this idea. Firstly the order of static variables, local variables in a stackframe and the shared library functions is randomized. This makes it harder for an attacker to overwrite adjacent variables. However, some objects can not be moved as the system expects them to be in a specific order. To make it harder for an attacker to overwrite these, random sized gaps are introduced between objects. For implementation purposes the authors have chosen to implement the randomization at link-time; this has an important shortcoming: the randomization is static, *i.e.* once the program has been randomized it will, for every execution, always contain the same randomization. This makes it easier for an attacker to figure out the base address, relative distances between variables and order, either by examining the program binary or by exploiting a vulnerability that causes the victim program to print out parts of its memory. For this reason the authors suggest to use a different method of implementation that does randomization at the beginning of execution (*i.e.* different randomizations each time the program is run) and that supports rerandomization at run time.

While not a kernel modification, **Transparent Runtime Randomization (TRR)** [65] can be seen as an operating system modification. It modifies the dynamic program loader in Linux, which is used to load programs in to memory for execution. It randomizes many of the memory locations where an attacker would place his code or that an attacker wouldmodify to execute code: the user stack set up by the kernel is randomized by allocating a new stack at a random location below the current one. The contents is copied to the new one and the pointers to it are adjusted accordingly. Finally the stack pointer is set to point to the new stack and the old stack is freed. The heap is randomized by randomly growing it. The location of the shared libraries are randomized by allocating a random-sized region before loading the shared libraries into the programs address space. The Global Offset Table (`GOT`), which is used to locate the information needed to execute position independent code (PIC) and is used by the PIC to access data, is loaded into a fixed position. To randomize the `GOT`, all positions that access the `GOT` directly must be rewritten to access this new position.

3.4.2 Randomized Instruction Sets

Another technique that can be used to prevent the injection of attacker-specified code is the use of randomized instruction sets. Instruction set randomization prevents an attacker from injecting any foreign code into the application by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. Attackers are unable to guess the decryption key of the current process, so their instructions, after theyve been decrypted, will cause the wrong instructions to be executed. This will prevent attackers from having the process execute their payload and will have a large chance crashing the process due to an invalid instruction being executed. Both implementations that we will describe here incur a significant run-time performance penalty when unscrambling instructions because they are implemented in emulators, but it is entirely possible, and in most cases desirable, to implement them at the hardware level and thus reducing the impact on run time performance.

Barrantes et al implement a proof-of-concept implementation of these randomized instructions by using emulators that emulate a processor. Their implementation **Randomized Instruction Set Emulation (RISE)** [14] is built on Valgrind an x86-to-x86 binary translator that is used to detect memory leaks. RISE scrambles the instructions at load-time and unscrambles them before execution. By scrambling at load-time the binaries can remain on the disk unmodified and as such the scrambled binary can not be examined by a potential attacker.

The approach by Kc et al [38] is also a proof-of-concept implementation using an emulator. It mainly differs from RISE in the fact that binaries are scrambled on disk instead of at load time. The key is then extracted from the header at load time and stored in the process control block (PCB) . The key is then stored in a special register that can only be written to with a special instruction (and can't be read from). Storing the encrypted binaries on disk has some major drawbacks though: an attacker that has local access could examine the encrypted binary and extract the key. More importantly the current implementation can only work with statically linked binaries because dynamically loaded libraries would have to be encrypted with the same key as the executable which means that any programs using the libraries would also have to use the same key. As a result the entire system would probably be using just a single key for all programs.

3.5 Compiler Modifications

Some of the attack detection tools work by inserting appropriate checks or instrumentation at compile time. The instrumentation depends on the particular approach and the attacks the tool is trying to prevent.

StackGuard [26] and **ProPolice** [30] take a similar approach to protecting an executable against a stack-smashing attack. During compilation, the prologue and epilogue of each function call is modified so that a *canary* value is inserted between the stack frame, which contains saved return IP and frame pointer, and local variables. Upon return from a function, this value is checked to ensure that it has not changed; if it has, there must have been an overflow attack that could have damaged the return IP - thus, the program is terminated with an error. StackGuard provides more sophisticated options, including a terminator canary (consisting of four bytes - NULL, $\backslash r$, $\backslash n$, EOF). Such a canary provides better protection when the buffer is overflowed using a string library function, as these bytes serve as termination characters. By default, a random value picked at compile time is used for the *canary.* A major limitation of this approach is that only continuous overflows that overwrite the canary on the stack are detected. Many other indirect attacks can still change the value of the return IP. ProPolice tries to mitigate this problem by rearranging local variables such that character stack buffers are directly below the canary value. While this method helps prevent some indirect stack attacks, it does not protect any heap buffers. Read overflows and underflows also remain undetected.

PointGuard [25] attempts to protect pointers by encrypting them while they are in memory and decrypting them when they are loaded into registers where they are safe from overwriting. While an attacker will still be able to modify pointers stored in memory, when the pointers that the attacker provides are decrypted, they will point to different (possibly inaccessible) memory locations. Without the encryption key, the attacker is unable to use pointers that will decrypt to a predictable value. The encryption is extremely simple so as not to cause a significant overhead to execution time of the program. A random value is generated at program start-up and every pointer is XOR'ed with this value, this will generally make it impossible for attackers to bypass the protection as the process will most likely crash when a pointer is decrypted wrongly and subsequently used. When the program is restarted a new key will be generated, making it very hard for attackers to guess the key. However, if attackers are able to cause the program to print out pointer information through some other vulnerability then they might be able to guess the decryption key and would then be able to encrypt their own pointers correctly.

FormatGuard [24] offers counter measures for the format string attacks. These countermeasures are based on the observation that most format string attacks have more specifiers in the format string than arguments passed to the format function. FormatGuard tries to prevent format string attacks by counting the amount of arguments that a format string expects and compares this to the amount of arguments that were actually passed to the function. If more were expected than provided it is likely that someone is attempting to exploit to a format string vulnerability and the program will be terminated.

3.6 Runtime Detection of Attacks

A different approach to mitigating the problem of memory corruption attacks is to stop the attacks that exploit them at runtime. Such tools either use instrumentation inserted into the executable at compile time or monitor the binary executable itself to determine when an out-of-bounds access has occurred. By stopping the attack in its tracks, these tools convert the attack into a denial of service.

Fine-grained Bounds Checking

TinyCC and CRED both provide fine-grained bounds checking based on the *referent object* approach, developed by Jones and Kelly [37]. As buffers are created in the program, they are added to an object tree. For each pointer operation, the instrumentation first finds the object to which the pointer currently refers in the object tree. The operation is considered illegal if it references memory or results in a pointer outside said object. When an illegal operation is detected, the program is halted. **TinyCC** [15] is a small and fast C compiler that uses a re-implementation of Jones and Kelly code; however, it is much more limited than gcc and provides no error messages upon an access violation - the program simply segfaults. **CRED** [54], on the other hand, is built upon Jones and Kelly code, which is a patch to gcc. Thus, CRED is able to compile almost all programs that gcc can handle. In addition, CRED adheres to a less strict interpretation of the C standard and allows illegal pointers to be used in pointer arithmetic and conditionals, while making sure that they are not dereferenced. This treatment of out-of-bounds pointers significantly increases the number of real world programs that can be compiled with this instrumentation, as many programs use out-of-bounds pointers in testing for termination conditions.

An interesting approach to mitigating buffer overflows via fine-grained bounds checking has been developed by Martin Rinard, as part of failure-oblivious computing. Instead of halting program execution when an out-of-bounds access has been detected, a program compiled with the failure-oblivious compiler will instead ignore the access and proceed as if nothing has happened. This approach prevents damage to program state that occurs due to out-of-bounds writes and thus keeps the program from crashing or performing actions it is not intended to do. Instrumentation likewise detects out-of-bounds reads and generates some value to return - for example, returning NULL will stop read overflows originating in string functions. The paper discusses several other values that may drive the program back into a correct execution path. This compiler is implemented as an extension to CRED and thus also uses *referent object* approach.

Insure++ [49] is a commercial product that also provides fine-grained bounds checking capabilities. The product is closed-source so little is known about its internal workings. According to its manual, Insure examines source code and inserts instrumentation to check for memory corruption, memory leaks, memory allocation errors and pointer errors, among other things. The resulting code is executed and errors are reported when they occur. The main limitation of all these tools is that an overflow within a structure is not recognized until the border of the structure has been reached. CRED is capable of detecting overflows within structures correctly, but only when structure members are accessed directly - if an access is made through an alias pointer, CRED will detect the overflow only when it reaches the border of the structure. These tools also cannot detect overflows within uninstrumented library functions. TinyCC seems to have some incorrect wrappers for library functions and cannot handle large programs, which is a significant drawback. Another limitation of TinyCC is that no error messages are provided - the program simply terminates with a segmentation fault. Insures major flaw is the performance slowdown that it imposes on executables - some programs run up to 250 times slower.

3.6.1 Executable Monitoring

Another method of detecting and preventing buffer overflows at runtime is through executable monitoring. **Chaperon** [49] is an executable monitor that is part of the Insure toolset from Parasoft, and **Valgrind** [46] is an open-source project aimed at helping developers find buffer overflows and prevent memory leaks. Tools such as Chaperon and Valgrind wrap the executable directly and monitor calls to `malloc()` and `free()`, thus building a map of heap memory in use by the program. Buffer accesses can then be checked for validity using this memory map.

There are several limitations to this approach. Since no functions are explicitly called to allocate and deallocate stack memory, these tools can monitor memory accesses on the stack only at a very coarse-grained level, and will not detect many attacks exploiting stack buffers. Another limitation is the large performance overhead imposed by these tools. Since Valgrind simulates program execution on a virtual x86 processor, testing is 25-50 times slower than `gcc`.

Program shepherding [39] is an approach that determines all entry and exit points of each function, and ensures that only those entry and exit points are used during execution. Thus, an attacker cannot inject code and jump to it by changing a code pointer - this will be noticed as an unspecified exit point and the program will be halted. Like operating system approaches discussed above, this method is transparent to the program running on the system. However, it is much more heavy-weight and incurs higher performance overhead. In addition, attacks that do not rely on changing code pointers will still succeed, as they do not change entry and exit points of a function.

A similar approach is used by **StackShield** [9]. One of the security models supported by this tool is a Global Return Stack. Return IP for each function that is entered is stored on a separate stack, and the return pointer is compared with the stored value when the function call is completed. If the values match, the execution proceeds as intended; however, if the values do not match, the value on the return stack is used as the return address. Thus, the program is guided back to its intended path. This approach detects attacks that try to overwrite the return IP or provide a fake stack frame. However, this method does not stop logic-based attacks or attacks based on overwriting a function pointer. Another security model offered by StackShield is Return Change Check. Under this model a single global variable is used to store the return IP of the currently executing function, and upon exit the return IP saved on the stack is compared to the one saved in this variable. If they do not match, the program is halted and an error message is displayed. This security model also tries to protect function pointers by making the assumption that all function pointers should point into the text segment of the program. If the program tries to follow a function pointer that points to some other memory segment, it is halted and an alert is raised. While this method does not prevent logic-based attacks from proceeding, it makes attacking the system more difficult. Another limitation of this approach is that read overflows are not detected.

3.6.2 Software Fault Injection

A popular approach for testing program robustness is a dynamic analysis technique focused on injecting faults into software execution. This technique creates anomalous situations by injecting faults at program interfaces. These faults may force a particular program module to stop responding or start providing invalid or malformed data to other modules. By observing system behavior under such anomalous conditions, certain inherent vulnerabilities can be discovered. Tools such as **FIST** [34] and **Fuzz** [44] have been used to perform such testing, with some promising results.

The main drawback of software fault injection is that it requires considerable developer intervention and effort. The code must be manually prepared for testing and the developer needs to analyze the output of each test case reporting a system crash to determine where the error has occurred. Automating program execution with fault injection is also difficult, as the program is likely to crash or hang due to inserted faults.

3.7 Information Flow Tracking

The idea behind information flow tracking or *taint analysis* based approaches is that in order for an attacker to change the execution of a program illegitimately, he must cause a value that is normally derived from a trusted source to instead be derived from his own input. For example, values such as jump addresses and format strings should usually be supplied by the code itself, not from external untrusted inputs. However, an attacker may attempt to overwrite these values with his own data. This technique works by labeling the input data from untrusted sources such as network as *unsafe* or *tainted*. The data derived from such tainted data is itself marked as tainted. An attack is detected when program control branches to a location specified by the unsafe data.

Static taint analysis has been used to find bugs in C programs [57, 33, 68] or to find potential sensitive data in Scrash [19]. Perl [5] does run time taint checking to see whether data from untrusted sources are used in security-sensitive ways such as as argument to a system call.

TaintBochs [22], Suh et al [60], Chen et al [66] and Minos [27] perform taint tracing at the hardware level. **TaintBochs** [22] is built on top of the open source IA-32 simulator Bochs. It is a tool based on whole-system simulation analyzing how sensitive data are handled in large programs. Suh et al [60] propose an information tracking approach that tracks spurious information flows and restricts their usage. **Minos** [27] is a micro-architecture that implements Biba's low watermark integrity checking on individual words to detect attacks at run time. Chen et al [66] defeat memory corruption attacks using an architectural support that detects attacks when a tainted pointer is dereferenced. Their approach can defeat non-control data attacks. The main limitation of these systems is that they require specialized hardware. Without this custom hardware, hardware emulators can be used but performance becomes unacceptable.

Binary instrumentation for taint-tracking was first used in **TaintCheck** [47]. While effective in attack detection, their approach slows down programs significantly (by about 37x). **TaintTrace** [21] achieved significantly faster taint-tracking by using more efficient instrumentation based on DynamoRIO, combined with simple static analysis for eliminating redundant register saves and a shadow memory data structure that speeded up metadata access. **LIFT** [51] achieved significant additional performance benefits by using better static analysis and faster instrumentation techniques. Although all of these approaches detect a wide range of memory corruption attacks, none of them defends against non-control data attacks, rendering them vulnerable to privilege escalation.

3.8 Static Analysis

Static analysis is generally used during the implementation or audit phases of an application, although it can also be used by compilers to decide if a specific run time check is necessary or not. Static analyzers do not offer any protection by themselves but are able to point out what code might be vulnerable. They operate by examining the source code of a particular application.

They can be as simple as just searching the code for specific known vulnerable functions, such as `strcpy()`, `gets()`, etc., or as complicated as building a complete model of the running application.

Splint [41, 31] is a lightweight static analysis tool, that analyzes the program source code based on annotations that describe assumptions made about specific objects (i.e. specific value, lifetime, etc.) or pre and post conditions that must be met for a specific function. By providing some built-in annotations for specific *vulnerable* functions, *e.g.* for `strcpy()`: $/* \ @requires \ maxSet(s1) \leq maxRead(s2) \ @*/$, *i.e.* the size of s1 (destination) is larger than the amount of data read from s2 (source). Splint can be used with a minimum of effort by a programmer to verify source code for commonly encountered implementation errors. Format string vulnerabilities are detected using taint analysis. All user input is considered tainted and an error is reported if a tainted variable is used where an untainted one is expected. The definitions of the format functions are then changed to expect an untainted format strings as arguments.

Boon [63] is a static analysis based tool for detecting buffer overflows. It models strings in C as integer ranges: each string is represented by two integers: the number of bytes allocated for the string and the number of bytes currently in use. A constraint language based on these ranges is defined and the specific string operations are modeled in this language (i.e. the constraints that must hold are specified). The program is subsequently parsed and a system of integer range constraints is generated for each statement in the input program. As each string is represented by two variables and all string operations are modeled with regard to these values, the safe state is when $\forall s \mid len(s) < alloc(s)$. To simplify analysis, the authors decided to use flow insensitive analysis (control-flow is ignored when generating constraints), this brings about some problems for functions like strcat which could be executed in a loop and have a state that relies on the previous execution. To alleviate this problem any call to strcat is flagged as a possible vulnerability, which will most likely generate some false positives. The current technique also has some limitations when analyzing pointer operations: it can not handle pointer aliasing correctly; ignores function pointers, doubly indirected pointers and unions. After generation of constraints, the constraint system is solved by finding a minimal bounding box solution that encloses all of the possible execution paths. The places where the constraints might not hold are reported as possible overflows.

White-listing [52] is another approach that defends against format string attacks. It uses source code transformation to automatically insert code and maintains checks against the whitelist containing safe `%n` writable address ranges via knowledge gained from static analysis. It requires source code and recompilation of the program to be protected.

3.9 Anomaly Detection

Anomaly detection based techniques work by building a database of *normal* for a process. Deviation from this normal is considered as an anomaly an flagged as an intrusion. In many cases the execution of system calls is monitored and if they do not correspond to a previously gathered pattern, an anomaly is recorded. Once a threshold for anomalies is reached, the anomaly can be reported and subsequent action can be taken (e.g. the program is terminated or the system call is denied).

Forrest et al [32] define a method for anomaly detection based on short sequences of system calls. A sliding window is used to record system calls during the training period: in that window a sequence of system calls of a specific size is recorded. The window is slided across a trace of the system calls that an application performs. If the program deviates from these sequences of calls in subsequent runs an anomaly is detected. The smaller the size of the window, the more options an attacker has for forming valid sequences of system calls while still being able to execute useful code. Some sequences that could never exist in the actual program would also not be recorded as an anomaly when using the short sequence of calls technique. Programs that use many different system calls and in many orders might also cause a problem. The allowable sequences might be so large that the attacker is not really restricted anymore.

Sekar et al [55] suggest using finite state automata to model system call sequences. They build an automaton during the training period that models the sequence of system calls that are performed during program execution. The advantage of using an automaton is that control-flow structures like loops and branches are captured in the model, allowing the automaton to make more accurate decisions about the validity of system calls. The main problem when building automata for modeling is deciding if two specific system calls are the same or not, which can lead to the building of inefficient automata. To be able to detect if these two calls are the same, the automaton needs to contain more information about the current program state than just the system call, so whenever a system call is added to the automaton the location in the program where the system call was called is also recorded. This allows the program to decide if two calls to the same system call are actually the same call. Once the automaton has been built, it can be used for anomaly detection during subsequent runs of the program. At runtime execution of system calls is intercepted and the location from where the call was made is looked up. If a transition can be made from the current state to a new state in the automaton with the intercepted system call then the automaton is placed in the new state. If no transition can be made, an anomaly is recorded. The automaton is then resynched with the program using the program location. When an anomaly is recorded it isn't immediately reported as this might cause too many false positives, instead an anomaly threshold is defined. When reached, action can be taken. The authors implement this using weighted anomalies, some are more likely to have been caused by a malicious user and should therefor weigh more heavily on the *anomaly index*.

Chapter 4

FormatShield

In this chapter, we describe FormatShield, a comprehensive solution to format string attacks. FormatShield works by intercepting calls to vulnerable functions in `libc` and identifying call sites in a running process that are vulnerable to format string attacks. By using binary rewriting, the list of exploitable program contexts at the vulnerable call sites is dumped in the program binary, which is made available at runtime. Attacks are detected when malicious input is found at vulnerable call sites with an exploitable program context. It does not require source code and recompilation of the program to be protected and, therefore, can be used to protect legacy or proprietary programs for which source code may not be available. It does not take into consideration the presence of any specific format specifier such as %n in the format string, and thus, it can defend against both types of format string attacks, i.e. arbitrary memory read attempts and arbitrary memory write attempts. It is capable of defending against non-control data attacks. It does not rely on the target of the format specifiers, and thus can protect against both, attacks that target control information such as return address on the stack, and those which target program specific security sensitive non-control information. Also, our experiments show that it incurs a very nominal performance penalty of less than 4%.

4.1 Approach Description

FormatShield works by identifying call sites in a running process that are potentially vulnerable to format string attacks. A potentially vulnerable call site is identified when a format function is called with a probable legitimate user input as a format string argument. Further, a probable legitimate user input can be identified by checking whether the format string is writable and without any format specifiers. The format string argument of a non-vulnerable call site, such as in `printf("%s", argv[1])`, would lie in a non-writable memory segment, while that of a vulnerable call site, such as in `printf(argv[1])`, would lie in a writable memory segment. The key idea here is to augment the program binary with the program context information at the vulnerable call sites. A program context represents an execution path within a program and is specified by the set of return addresses on the stack. Since all execution paths to the vulnerable call site may not lead to an attack, FormatShield considers only those with an exploitable program context. For example, Figure 4.1 shows a vulnerable code fragment. Here the vulnerability lies in the function `output()`, which passes its argument as the format string to printf. Although `output()` has been called from three different call sites in `main()`, note that only one of these three, *i.e.* `output(argv[1])`, is exploitable. Here, the contexts, *i.e.* the set of return addresses on the stack, corresponding to all the three calls will be different, and thus the context corresponding to `output(argv[1])` can

```
void output(char *str) {
    printf(str);
}

int main(int argc, char **argv) {
    output("alpha");
    .....
    output("beta");
    .....
    output(argv[1]);
    .....
}
```

Figure 4.1: Only the third call to `output()` is exploitable

easily be differentiated from those of other two calls to `output()`. As the process executes, the exploitable program contexts are identified. The next time, if the vulnerable call site is called with an exploitable program context with format specifiers in the format string, a violation is raised. When the process exits, the entire list of exploitable program contexts is dumped into the binary as a new loadable read-only section, which is made available at runtime for subsequent runs of the program. If the section already exists, the list of exploitable program contexts is updated in the section. The program binary is updated with context information over use and becomes immune to format string attacks.

4.2 Implementation

FormatShield is implemented as a shared library that intercepts calls to the vulnerable functions in `libc`, preloaded using `LD_PRELOAD` environment variable. This section explains the design and implementation of FormatShield. First we explain how FormatShield identifies vulnerable call sites in a running process. Then we describe the binary rewriting approach used to augment the binary with the context information.

4.2.1 Identifying vulnerable call sites

During process startup, FormatShield checks if the new section (named `fsprotect`) is present in the binary of the process. This is done by resolving the symbol `fsprotect`. If present, the list of exploitable program contexts is loaded. During process execution, whenever the control is transferred to a vulnerable function intercepted by FormatShield, it checks if the format string is writable. This is done by looking at the process memory map in `/proc/pid/maps`. If the format string is non-writable, corresponding equivalent function (such as `vprintf()` for `printf()`) in `libc` is called, since a non-writable format string cannot lead to an attack. However, if the format string is writable, FormatShield identifies the current context of the program, and checks if this context is in the list of exploitable program contexts. The current context of the program, i.e. the set of return addresses on the stack, is retrieved by following the chain of stored frame pointers on the stack. Instead of storing the entire set of return addresses on the stack, FormatShield computes a lightweight hash of the return addresses. If the current context is not present in the list of exploitable contexts, FormatShield checks if the format string is without any format specifiers. If the format string does not contain any format specifiers, it is identified as a legitimate user input, and the current context is added to the list of exploitable contexts. Otherwise, if the format string contains format specifiers, it is not added to the list of exploitable contexts. In either case, FormatShield calls the equivalent function in `libc`. Note that, if the format string contains format

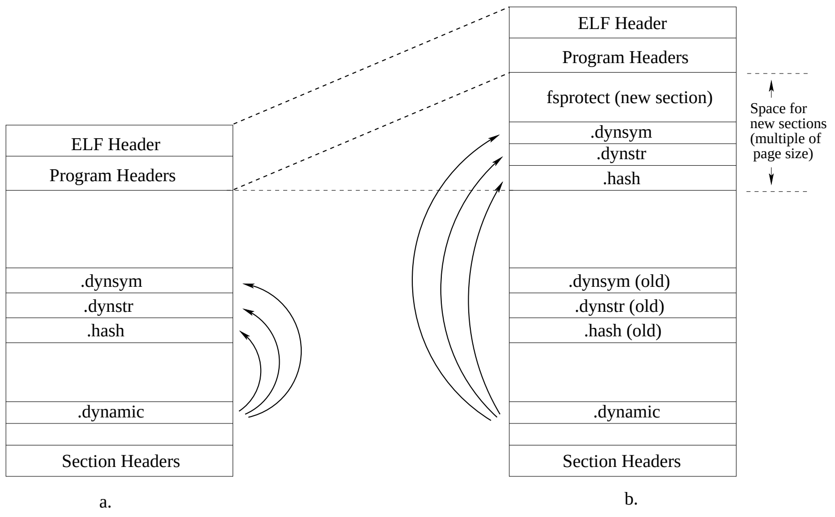


Figure 4.2: ELF binary **a.** before rewriting **b.** after rewriting

specifiers, it could be a case when an exploitable context is not yet identified by FormatShield and is being exploited by an attacker. However, FormatShield takes a safer step of not identifying it as an attack, since an exploitable context is highly likely to be identified before an attack, either over use or by exercising "synthetically". If the current context is in the list of exploitable contexts, FormatShield checks if there are any format specifiers in the format string. If the format string does not contains any format specifiers, FormatShield calls the equivalent function in `libc`. Otherwise, if the format string contains format specifiers, FormatShield raises a violation. On detecting an attack, the victim process is killed, and a log is written to `syslog`. Note that the default action to terminate the process can be used as a basis to launch denial of service (DoS) attack against the victim process by an attacker. However, silently returning from the vulnerable function without terminating the process may lead to an application specific error. For e.g., if the vulnerability lies in a call to `vfprintf()`, skipping the call may lead to no output being printed to terminal if the string is being printed to `stdout`, which may not be fatal. However, if the string is being printed to a file, skipping the `vfprintf()` call may lead to a corrupted file. Terminating the victim process would create "noise" that a conventional host-based intrusion detection system can detect the intrusion attempt.

4.2.2 Binary Rewriting

FormatShield uses an approach (Figure 4.2) similar to that used by TIED [10] to insert context information in the program binary. FormatShield currently supports only ELF [23] executables. In the ELF binary format, `.dynsym` section of binary contains symbols needed for dynamic linking, `.dynstr` section contains the corresponding symbol names, and `.hash` section holds a hash look up table to quickly resolve symbols. `.dynamic` section holds the addresses of these three sections. The information to be inserted is a list of hashes of stored return addresses corresponding to exploitable contexts at different vulnerable call sites in the program. During process exit, the entire list is dumped into the executable as a new read-only loadable section. If the section is already present, the context information in the section is updated. A typical ELF binary loads at virtual address `0x08048000`. To add a new section (Figure 4.5 and 4.6), FormatShield extends the binary towards lower addresses, i.e. lower than address `0x08048000`. This is done to ensure that the addresses of existing code and data do not change. To make the context information available at run time, a new dynamic symbol (Figure 4.3 and 4.4) is added to the `.dynsym` section and the corresponding address is set to that of the new section. Since this requires extending `.dynsym`, `.dynstr` and `.hash` sections which cannot be done without changing the addresses of other sections, FormatShield creates an extended copy of these sections, i.e. `.dynsym`, `.dynstr` and `.hash`, and changes their addresses in `.dynamic` section. The address of the new section is so chosen such that the sum of the sizes of the four new sections is a multiple of page size. The space overhead is of the order of few kilobytes (less than 10 KB for most binaries). [1]

```
DYNAMIC SYMBOL TABLE:
00000000       DF    *UND*       000000e7    __libc_start_main
00000000       DF    *UND*       00000039    printf
080484a4  g    DO    .rodata     00000004    _IO_stdin_used
00000000  w    D     *UND*       00000000    __gmon_start__
```

Figure 4.3: Dynamic symbol table before rewriting the binary

```
DYNAMIC SYMBOL TABLE:
00000000       DF    *UND*       000000e7    __libc_start_main
00000000       DF    *UND*       00000039    printf
080484a4  g    DO    .rodata     00000004    _IO_stdin_used
00000000  w    D     *UND*       00000000    __gmon_start__
08047114  g    DO    fsprotect   0000000a    fsprotect
```

Figure 4.4: Dynamic symbol table after rewriting the binary. A new dynamic symbol named `fsprotect` is added while rewriting the binary which points to the new section at address `0x08047114`.

```
....
[003] 0x08048148 a------ .hash      sz:00000040 link:04
[004] 0x08048170 a------ .dynsym    sz:00000080 link:05
[005] 0x080481C0 a------ .dynstr    sz:00000076 link:00
....
```

Figure 4.5: Sections before rewriting the binary

```
....
[003] 0x080480E8 a------ .hash      sz:00000044 link:04
[004] 0x08048030 a------ .dynsym    sz:00000096 link:05
[005] 0x08048090 a------ .dynstr    sz:00000088 link:00
....
[026] 0x08047114 a------ fsprotect  sz:00003868 link:00
[027] 0x08048170 a------            sz:00000080 link:00
[028] 0x080481C0 a------            sz:00000076 link:00
[029] 0x08048148 a------            sz:00000040 link:00
```

Figure 4.6: Sections after rewriting the binary. A new loadable read-only section named `fsprotect` is added which holds the context information. The `.dynsym`, `.dynstr` and `.hash` sections shown are extended copies of the original ones. The original `.dynsym`, `.dynstr` and `.hash` are still loaded at their original load addresses.

[1]As per ELF specification [23], loadable process segments must have congruent values of the load address, modulo the page size.

4.2.3 Implementation Issues

Address Space Randomization

Address Space Randomization (ASR) [4, 16, 17] involves randomizing base addresses of various segments so as to make it difficult for an attacker to guess an address. Since the base addresses of various code segments are randomized, the absolute memory locations associated with the set of return addresses will change from one execution of the program to the next. To compensate for this, we decompose each return address into a pair {`name, offset`}, where `name` identifies the executable or the shared library, and `offset` identifies the relative distance from the base of the executable or shared library.

4.3 Evaluation

We conducted a series of experiments to evaluate the effectiveness and performance of FormatShield. All tests were run in single user mode on a Pentium-4 3.2 GHz machine with 512 MB RAM running Linux kernel 2.6.18. All programs were compiled with `gcc` 4.1.2 with default options and linked with `glibc` 2.3.6.

4.3.1 Effectiveness

We tested FormatShield on five programs with known format string vulnerabilities:

- `wuftpd` version 2.6.0 and earlier suffer from a format string vulnerability [61] in the "`SITE EXEC`" implementation. A remote user can gain a root shell by exploiting this vulnerability.
- `tcpflow` 0.2.0 suffers from a format string vulnerability [59], that can be exploited by injecting format specifiers in command line arguments. A local user can gain a root shell by exploiting this vulnerability.
- `xlock` 4.16 suffers from a format string vulnerability [18] when using command line argument `-d`, that can be used by a local user to gain root privileges.
- `rpc.statd` (`nfs-utils` versions 0.1.9.1 and earlier) suffers from a format string vulnerability [35], which allows a remote user to execute arbitrary code as root.
- `splitvt` version 1.6.5 and earlier suffer from a format string vulnerability when handling the command line argument `-rcfile`. A local user can gain a root shell[2] by exploiting this vulnerability.

The above programs were trained "synthetically" with legitimate inputs before launching the attacks so as to identify the vulnerable call sites and the corresponding exploitable contexts. FormatShield successfully detected all the above attacks, and terminated the programs to prevent execution of malicious code. The results are presented in Table 4.1. To check the effectiveness of FormatShield on non-control data attacks, we modified the publicly available exploit for the wuftpd 2.6.0 format string vulnerability [61] to overwrite the cached copy of user ID `pw->pw_uid` with 0 so as to to disable the server's ability to drop privileges. Such an attack has been tested in [20]. FormatShield successfully detected the write attempt to the user ID field and terminated the child process.

[2]The attack gives a root shell if the program is installed suid root, otherwise it gives a user shell.

Vulnerable program	**CVE #**	**Results without FormatShield**	**Results with FormatShield**
`wuftpd`	CVE-2000-0573	Root Shell	Process Killed
`tcpflow`	CAN-2003-0671	Root Shell	Process Killed
`xlock`	CVE-2000-0763	Root Shell	Process Killed
`rpc.statd`	CVE-2000-0666	Root Shell	Process Killed
`splitvt`	CAN-2001-0112	Root Shell	Process Killed

Table 4.1: Results of effectiveness evaluation

4.3.2 Performance Testing

To test the performance overhead of FormatShield, we performed micro benchmarks to measure the overhead at function call level, and then macro benchmarks to measure the overhead at application level.

Micro benchmarks

To measure the overhead per function call, we ran a set of simple benchmarks consisting of a single loop containing a single `sprintf` call. A six character writable string was used as the format string. With no format specifiers, FormatShield added an overhead of 12.2%. With two `%d` format specifiers, overhead was found to be 4.6%, while with two `%n` format specifiers the overhead was 3.3%. We also tested `vsprintf` using the same loop. The overheads were found to be 15.5%, 1.9% and 3.4% for no format specifiers, two `%d` format specifiers, and two `%n` format specifiers respectively. The overheads were found to be much less than those with the previous approaches. Table 4.2 compares micro benchmarks of FormatShield with those of FormatGuard and White-Listing.

Benchmark	**Format Specifiers**	**FormatGuard**	**White-Listing**	**FormatShield**
`sprintf`	nil	7.5%	10.2%	12.2%
	2 %d	20.9%	28.6%	4.6%
	2 %n	38.1%	60.0%	3.3%
`vsprintf`	nil	No protection	26.4%	15.5%
	2 %d	No protection	39.8%	1.9%
	2 %n	No protection	74.7%	3.4%

Table 4.2: Micro benchmarks

Macro benchmarks

To test the overhead at the application level, we used `man2html` since it uses `printf` extensively to write HTML-formatted man pages to standard output. The test was to translate 4.6 MB of man pages. The test was performed multiple times. It took `man2html` 0.468 seconds to convert without FormatShield, and 0.484 seconds with FormatShield. Thus, FormatShield imposed 3.42% run-time overhead.

4.4 Discussion

4.4.1 False Positives and False Negatives.

Although, during the entire testing phase, we did not find any false positives or false negatives, we have identified a case which could lead to a false positive or false negative. It is when a format string is dynamically constructed as a result of a conditional statement and then passed to a format function. A false positive can be there if one outcome of the condition creates a format string with format specifiers and the other outcome creates one without format specifiers. Similarly, a false negative can be there when one outcome of the condition reads user input into the format string and the other outcome creates a format string with format specifiers. This is because the context remains the same at the call site in both the cases. However, during the entire testing phase, we did not encounter any such code constructs. Also, there could be a false negative when format specifiers are present at the vulnerable call site but the corresponding context is not yet identified.

4.4.2 Limitations

FormatShield requires frame pointers to obtain the set of stored return addresses on the stack, which are available in most cases. However, it may not be able to protect programs compiled without frame pointers, such as those compiled with `-fomit-frame-pointer` flag of `gcc`. Also, FormatShield requires that exploitable contexts of the vulnerable call sites are identified before it can detect attacks. This may require the program to be trained, either by deploying or by exercising "synthetically". Another limitation of FormatShield is that it requires programs to be dynamically linked (since library call interpositioning works only with dynamic linked programs). However, this is not a problem if we consider Xiao's study [64] according to which 99.78% applications on Unix platform are dynamically linked. Also, since FormatShield keeps updating the context information in the program binary till it becomes immune to format string attacks, it may interfere with some integrity checkers.

Chapter 5

Coarse Grained Dynamic Taint Analysis

Several approaches have been proposed in the past years to detect memory corruption attacks. Approaches such as StackGuard [26] and Libsafe [62] target specific attacks such as buffer overflows. Many of them even require source code and recompilation of the program to be protected. Other approaches such as Address Space Randomization (ASR) [4, 16, 17, 65] and Instruction Set Randomization (ISR) [14, 38] are generic, but are easily defeated by brute force attacks in a matter of few minutes [56, 28, 58]. Several recent work [66, 42, 47, 60, 40, 51, 21, 22, 67, 27] demonstrated that information flow tracking, or *taint analysis* is a promising and effective technique for detecting a wide range of security attacks that corrupt control data such as return addresses or function pointers, even against zero-day attacks. The idea behind taint analysis is that in order for an attacker to change the execution of a program illegitimately, he must cause a value that is normally derived from a trusted source to instead be derived from his own input. For example, values such as jump addresses and format strings should usually be supplied by the code itself, not from external untrusted inputs. However, an attacker may attempt to overwrite these values with his own data. This technique works by labeling the input data from untrusted sources such as network as “unsafe” or *tainted*. The data derived from such tainted data is itself marked as tainted. An attack is detected when program control branches to a location specified by the unsafe data. Taint analysis is capable of detecting a wide range of memory corruption attacks. However, previous taint analysis based approaches do not defend against non-control data attacks, in which an attacker targets a program specific security critical data, such as a variable storing user privileges, instead of control information. Also, the overheads incurred by previous approaches slow down the program execution several times, making them unsuitable for use in production environments.

In this section, we discuss *Coarse Grained Dynamic Taint Analysis*, a very low-overhead information flow tracking technique. We propagate *taint* on variables, rather than individual words of memory. Our experiments with several exploits show that Coarse Grained Dynamic Taint Analysis effectively detects various types of memory corruption attacks, including non-control data attacks. Also, as it incurs modest performance overhead, it is practical for use in production environments.

5.1 Coarse Grained Dynamic Taint Analysis

This section presents an overview of our approach and the details of the binary instrumentation framework that is required to implement the taint analysis.

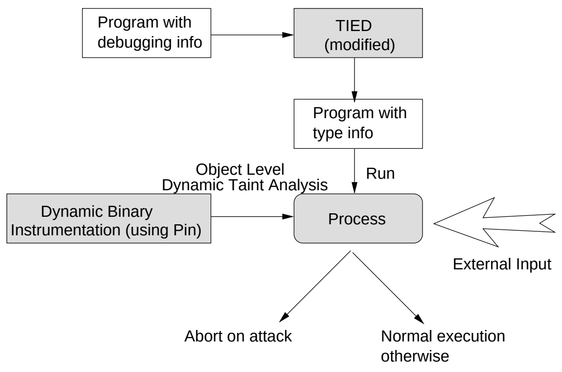

Program with debugging info
TIED (modified)
Program with type info
Object Level Dynamic Taint Analysis
Run
Dynamic Binary Instrumentation (using Pin)
Process
External Input
Abort on attack
Normal execution otherwise

Figure 5.1: Approach Overview

5.1.1 Approach Overview

Our approach is based on the observation that in a program, variables are the entities used by a programmer to accomplish any task, such as receiving user input, reading or writing files, arithmetic, etc. As a program executes, these variables might be copied among themselves (such as `a=b`), values of some variables might be derived from other variables (such as `c=a+b`), others might be received from user input (such as `gets(str)`), etc. An attacker may cause the value of one variable to illegitimately affect the value of another variable or control data by providing a kind of input that the programmer did not expect, such as an excessively large input, an out of bounds value, an input with format specifiers, etc.

By tracking the source of each variable, we propagate taint on these variables. We extract variable addresses and their sizes from the debugging information. By using binary rewriting, this information is inserted in the program binary as a new loadable read-only section, available at run time. At run time, we associate a *tag* with each variable, that can have two values: 0 ("safe") or 1 ("unsafe" or "tainted"). The program is then dynamically instrumented to perform information flow tracking and to detect attacks that alter program control flow to unsafe data. All data (variables) received from untrusted sources, such as network or command line arguments, are marked as tainted. As the program executes, the taint is propagated among variables. Any operation that results in an variables being written, causes the tag of the destination variable to be affected by that of the source varaible. Any attempt to branch to an address specified by a tainted variable detects an attack. The overview of our approach is shown in Figure 5.1.

Tainting Policy. Our tainting policy for variables is simple, if a particular variable's state is changed directly by a tainted variable then we mark the current variable as tainted as well. This can be expressed as follows. If a variable Z is being modified by a function, F that takes one or more tainted variable as parameters, then, Z becomes tainted as well. The function F can be any function that directly affects Z. Formally, if Z is affected by F directly i.e., in the same statement then, Z is tainted if one or more parameters of F are also tainted. Note that, this policy excludes implicit flows that often occur due to programmer mistakes, e.g., changing a variable based on logical condition on a tainted value. Moreover, [67] has noted that tracing implicit flows is a hard problem.

Justification. Based on our tainting policy, we argue that our approach captures the semantics of the previous taint analysis approaches in [51, 47, 21] and hence, is justified. Earlier approaches propagate taint on every byte of memory. In our approach, note that, the taint tag is propagated on all bytes associated with a single variable. This implies that, even if a single byte of this variable is modified by tainted data, the entire range of bytes associated with the variable become tainted.

```
struct s {
        char a;
        int b;
};
struct s foo[20];
```

Figure 5.2: Variables within a structure

In fact, our tainting process is a super-set of the byte-level tainting process in [51, 47, 21] as an entire sequence of bytes is tainted even if a single byte is tainted. Furthermore, we propagate taint in all operations where the tainted variables are accessed thereby ensuring that all data affected by tainted data is marked tainted as well. Thus, based on these arguments, our approach appropriately captures the semantics of previous approaches. Later, using experimental evaluation we validate this justification thoroughly.

5.1.2 Framework Overview

Our framework consists of the following steps. First, our approach identifies variables in the application being checked. We perform static analysis during this phase to determine the instructions that refer to the variables at run-time. Second, we describe the taint tag management and storage issues in detail. Third, we describe the dynamic instrumentation framework that we use to rewrite the application at run-time. We show the different code instructions that are instrumented at run-time. Finally, we describe the exploit detection policy and the exploits detected using the instrumented code.

Identifying Variables. The first step in our approach is to extract the variables from the executable and make it available to our taint tracking framework at run-time. Note that, for each executable, this is a one-time process. The extracted information is added to the executable and stored along with it. Our variable extraction uses an enhanced version of the TIED[1] [10] framework which is suitable for our purpose. Since the program is compiled with debugging information, it contains variables location and size information that can be used at run time. TIED extracts location and size information of all the buffers in the program and dumps it in the executable as a separate loadable read-only section. For global variables, the addresses are known at compile time itself, while for local variables only offset from the frame pointer is known at compile time. We have modified TIED so that it dumps the location (virtual address for the global variables and offset from frame pointer for local variables) and size information of all the variables in the executable. The members of arrays, structures and unions are also explored to detect the variables that lie in them. Figure 5.2 demonstrates a typical case of variables within structures. TIED detects all 40 variables in this case. This location and size information is loaded at run time to identify variables in the virtual address space of the program. Figure 5.3 shows a code fragment and the corresponding variables that are created at run time. For the example code, the previous approaches would read the input up to 1024 bytes, mark the tag for all the bytes read as tainted and then propagate the taint to the destination. However, in our approach we mark the variable `src` as tainted and taint is propagated to `dest` variable. Therefore, for the same code, our approach marks an entity as tainted only once and taint propagation is required just once, thus saving taint marking as well as propagation effort to a large extent.

Static Analysis. We statically analyze the program to identify instructions referring to these variables in memory at program run-time. Global variables are identified by looking at the address being referred by the instruction. Local variables and function arguments are identified by looking

[1]Type Information Extractor and Depositor

```
void readdata(int fd) {
        char dest[1024];
        char src[1024];

        read(fd, src, sizeof(src));
        strcpy(dest, src);
}
```

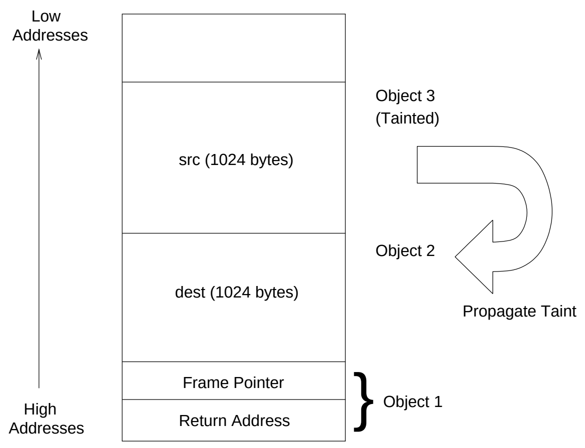


Figure 5.3: Variables in a stack frame

at the offset from frame pointer. A negative offset identifies a local variable and a positive offset identifies a function argument.

Tag Management. We associate a one bit tag with each variable and register, which can have only two values (0 for "safe" and 1 for "unsafe"). As a single bit tag is maintained for an variable that may span several bytes, the space overhead is considerably lower than approaches that use a single-bit tag per byte of memory. Instead of maintaining tags for the entire program's address space, we maintain tags only for writable parts of the program. This is done since the non-writable parts of the address space cannot be affected by the user input and are therefore assumed to be "safe". Hence, the tag for any variable belonging to a read-only part of the address space is assumed to be 0 or "untainted".

We store tags for variable in a separate memory region, called *tag space* (or *shadow memory*), which is addressable via a one-to-one direct mapping between an variable and the tag-bit in the program's virtual address space. Such a direct mapping makes it straightforward and fast (with only one memory access and a few arithmetic instructions) to get the tag value for a given variable. At program load time, our prototype uses the information dumped by TIED to identify variables in the process address space. For every variable, our approach maintains a (`offset`, `bit`) tuple, which is used to locate its tag bit. The `offset` refers to the difference between the tag byte that holds the tag bit for an variable and the beginning of the tag space, and the `bit` holds the bit position in that byte. This tuple is determined using static analysis of the program during load time. At program load time, instructions that refer to an variable are located, and are instrumented so as to propagate taint at runtime. For local variables, for which memory is reserved on the stack only when the corresponding function is called, this step is deferred till the corresponding function is called.

Dynamic Binary Instrumentation Framework and Tracking. To enable taint tracking, when the program is executed, we dynamically instrument the code. The instrumentation process supplements the code with additional instructions to enable variable tracking and taint propagation. The instrumentation needs to be done every time the program is restarted. Our dynamic binary instrumentation approach is built on top an existing dynamic binary instrumentation framework called Pin [43]. Pin provides efficient instrumentation by using a just-in-time (JIT) compiler to insert and optimize code. In addition to some standard techniques for dynamic instrumentation systems including code caching and trace linking. Pin implements register reallocation, liveliness analysis, code inlining and instruction scheduling to optimize jitted code.

```
movb $0x10,%eax
mov  $0xfffffffe,%ecx
rol  $0x03,%ecx
and  %ecx,$0xa8000000(%eax)
mov  $0x0400,%ebx
```

Figure 5.4: Instrumentation for the instruction `mov $0x0400,%ebx`

A tag space of one bit is maintained for each variable that was identified during static analysis. At program initialization, all tags are cleared to zero. Data (variables) received from untrusted sources such as network or standard input are marked as tainted. As the program executes, other variables may be tagged as tainted via information flow. A tainted variable may become untainted if its value is reassigned from some untainted variable. In accordance to our formal definition of *taint policy* (cf. Section 5.1.1), we instrument all data movement and arithmetic instructions.

- For data movement instructions such as `mov`, `lods`, `stos`, `push`, `pop`, etc, the tag value of the source operand is propagated to that of the destination. For data movement instructions which move a constant value in an variable, we instrument the instruction to mark the variable as untainted at run time.
- For arithmetic instructions, such as `add`, `sub`, `xor`, etc, the tag value of the destination is the OR of the tag values of the source operands, since the value of the destination is affected by both the source operands.
- For instructions which do not involve any second operand (explicit or implicit), such as `inc`, `dec`, etc, the tag value does not change.

In Figure 5.4, we show the instrumentation code generated for the instruction `mov $0x0400,%ebx`. As this instruction is associated with moving data it needs to be instrumented. Since a constant value `0x0400` is being copied to `ebx` register, the register needs to be marked as untainted. To do that, a value (`0x10` in this example) is copied to register `eax`. This value is an offset from the beginning of the tag space (`0xA8000000` in this example). Another value `0xfffffffe` is copied to `ecx` register. This value is then rotated left by 3 bits to create a mask, which is then ANDed with the byte at address `A800000000 + 0x10 = 0xA800000010`, to mark the corresponding taint bit for the `ebx` register as untainted. The original instruction is then executed.

Other instructions are identified using the rules described above. Note that, these rules are not applicable for a few special instructions for which the result is untainted irrespective of whether the operand is tainted or not. For example, in x86 architecture, the instructions such as “`xor %eax, %eax`” or “`sub %eax, %eax`” clear the `eax` register irrespective of its previous value. In our approach, we identify such instructions and mark the corresponding register as untainted. We instrument calls to functions that allocate memory such as `malloc()` to identify corresponding variable in global shadow memory. Calls to functions that copy memory such as `memcpy()` or `strcpy()` are instrumented to propagate taint from the source variable to the destination variable. Calls to functions such as `memset()` are instrumented to mark the corresponding argument as untainted if the source byte lies in an untainted variable. For `mmap` or `mmap2` system calls, we check the return value to see if the allocation was successful. For each successful allocation, we identify variables in the global shadow memory. In Table 5.1, we show a categorized list (partial) of instructions that are identified for instrumentation.

Table 5.1: Instructions instrumented for Taint Tracking

Instruction type / System or library call	**Instrumentation**
Data movement instructions, *e.g.* `mov`, `lods`, `stos`, etc.	tag[dest] = tag[source]
Arithmetic instructions, *e.g.* `add`, `sub`, `xor`, etc.	tag[dest] = tag[a] OR tag[b]
Instructions without any second operand, *e.g.* `inc`, `dec`, etc.	No change in tag value
Calls that copy memory, *e.g.* `memcpy`, `strcpy`, etc.	tag[dest] = tag[source]
Calls that set memory, *e.g* `memset`	tag[dest] = tag[source] if source is not constant, else tag[dest] = 0

5.1.3 Exploit Detection Policy

We use a configurable exploit detection policy to detect any exploitation attempts. The exploit detection policy specifies the sources that are untrusted, for example, network, files, standard input, command line arguments, environment variables, etc. Any data (variable) received from an untrusted source is marked as tainted. The policy also defines the checks that are applied when the program is run. Based on our exploit detection policy, we instrument all control altering instructions such as `call`, `jmp`, `ret`, etc. and check the branch address. A branch address belonging to a tainted variable detects a control hijacking attack. To detect format string attacks, we instrument calls to functions in the `printf`, `syslog`, and `warn/err` family. A format string belonging to a tainted variable detects a format string attack. The policy also checks whether any non-control variables are marked as tainted in an illegitimate manner. For example, a stack overflow attack that does not overwrite the stored return address, but only overwrites the next variable on the stack is noted in the exploit detection policy. Table 5.2 lists the exploit detection policies that are required for detecting different types of attacks.

The following discussion illustrates the attacks and the checks that are enforced by our exploit detection policy for detecting these attacks.

- *Branch Addresses.* The policy checks if any address is used as a target of any branch instruction such as `jmp` or `call`. This defeats all those attacks which overwrite return addresses, function pointers, global offset table (`GOT`) entries, etc. in order to redirect the program control flow to the injected code or to some library function such as `execve()`. It also defeats *frame faking attacks* that overwrite the stored frame pointer in order to create a fake stack frame when the function returns.

- *Format string arguments.* The policy checks if any format function such `fprintf()` or `syslog()` is called with a tainted format string argument. Any such case is identified as an exploitation attempt irrespective of the format specifiers used. Therefore, it can not only defeat attempts to overwrite arbitrary memory locations, but even arbitrary memory read attempts. Note that such a check can not only detect an exploitation attempt, but can also pinpoint the vulnerability even before it is exploited. This is because the format string argument will still be tainted even if the input is legitimate.

- *System call arguments.* The policy checks if any arguments used by a system call are overwrit-

Table 5.2: Attacks detected by different policies

Policy	**Attacks Detected**
Tainted branch addresses	Classical stack smashing attacks, common buffer overflows and frame faking attacks
Tainted format string arguments	Format string attacks
Tainted system call arguments	Any kind of code injection attacks
Tainted control data, *e.g.* tainted boundary tags in heap or `longjmp` buffers etc.	Heap overflows and stack overflows
Tainted Non-control data, *e.g.* variable containing user privileges etc.	All kinds of non-control data attacks

ten by tainted data. In Linux x86 architecture, a system call is implemented using software interrupt 0x80. The system call number is usually passed in the `eax` register. We instrument all `int 0x80` instructions to check the value of `eax` register at run time. For `execve` system call, we check at run time if its arguments are tainted. Such checks will not only defeat code injection attacks, but even those attacks in which an attacker overwrites data that is later used as an argument to a system call.

- *Other control data.* The policy also checks whether certain other control data such as boundary tags in heap or `longjmp` buffer are tainted. Such tags are inserted at points where these control data are used. For example, for boundary tags in heap, we insert checks in calls to `free()`, and for `longjmp` buffers, we insert checks in calls to `longjmp()` and `siglongjmp()`.

- *Non-control data.* The policy checks if any non-control data is marked as tainted in an illegitimate manner, such as by an overflow of an variable. Such checks are implemented by checking bounds of the variable being written, as done by previous approaches such as LibSafe [62] or LibSafePlus [10].

In the next section, we evaluate the effectiveness of our taint analysis approach against these attacks. We validate our technique through extensive testing over synthetic and real-time exploits.

5.2 Evaluation

We have implemented a prototype of our approach on Linux x86 architecture. We use TIED to dump variable size and location information into the program executable. Our prototype then uses Pin to instrument the code by adding instructions to perform information flow tracking on these variables and to detect attacks. The instrumentation is performed only once, at the program load time. However, the instrumented code may run many times. Pin caches the generated instrumented code, so that it need not be instrumented again when required.

We conducted a series of experiments to evaluate the effectiveness and performance of our approach. All tests were run in single user mode on a Pentium-4 3.2 GHz machine with 512 MB RAM running Linux kernel 2.6.18. All programs were compiled with gcc 4.1.2 and linked with glibc 2.3.6.

5.2.1 Effectiveness Evaluation

We tested the effectiveness of our approach using several synthetic and actual exploits. The exploits were selected from a wide range of attacks including stack smashing attack, heap overflow, format string attack, double free and non-control data attacks. Our approach successfully detected all the exploitation attempts and terminated the victim program to prevent execution of malicious code. The results are presented in Table 5.3.

Synthetic Exploits.

In this section, we evaluate our approach using synthetic exploits on stack overflows, heap overflows, and format string vulnerabilities. Our approach successfully detected all the attacks without any false negatives.

- *Stack Smashing Attack.* To test the effectiveness of our approach against the classical stack smashing attack, we wrote a program with a buffer overflow vulnerability. It uses `strcpy()` to copy a command line argument to a local buffer. We tried inserting a long command line argument in order to overflow the buffer and to overwrite the stored return address. Our approach successfully detected the tainted return address (which is treated like any other control data in our approach) when the function returned.

- *Heap Overflow Attack.* In a similar test, we verified our approach by detecting a heap overflow. We wrote a program with a heap overflow vulnerability. It allocates a buffer in heap, uses `strcpy()` to copy the first command line input into the buffer, and then frees the buffer. We injected a huge input in the first command line argument in order to overflow the buffer and overwrite the boundary tags. Our approach successfully detected the tainted boundary tags when the buffer was freed.

- *Format String Attack.* We used our approach on a custom written program with a format string vulnerability. The program uses `printf()` to display the first command line argument. The exploit for the program uses the format string attack to overwrite the global offset table (`GOT`) entry for `exit()` function. The exploitation attempt was successfully detected as the format string argument was found to be tainted. Note that for a format string attack, our approach can defeat all exploitation attempts of writing to and reading from arbitrary memory locations irrespective of the format specifier used.

- *Buffer Overflow Attack.* To test our approach against non-control data attacks, we tried overflowing a buffer in a structure in order to overwrite a character array stored next to it. The character array is used to hold the filename of the temporary file being written. The attack was detected by the fine grained policy as the character array was found to be illegitimately written by a tainted buffer.

Actual Exploits.

We tested our approach on five exploits on publicly known vulnerabilities. All the attacks were successfully detected by our approach.

- *Samba `call_trans2open()` Buffer Overflow.* Samba version 2.2.8 and earlier suffer from a stack smashing vulnerability [6] in `call_trans2open()` function. Successful exploitation of the vulnerability allows a remote attacker to gain a root shell on the machine running vulnerable version of samba. Our approach detected the tainted stored return address and defeated the exploitation attempt.

- *BIND 8 Buffer Overflow.* BIND version 8.2 and earlier suffer from a buffer overflow [48] in the
`nslookupComplain()` routine, which allows a remote attacker to gain root access on the affected machine. Our approach correctly detected that the return address is tainted and defeated the attack.

- *xlockmore Format String Vulnerability.* xlock version 4.16 suffers from a format string vulnerability [18] when using the command line argument `-d`, that can be used by a local user to gain root privileges. The exploit overwrites the stored return address with the address of the injected shellcode. Our approach successfully identified that the stored return address was tainted and defeated the exploitation attempt.

- *PHP `Session_Decode()` Double Free Memory Corruption Vulnerability.* PHP version 4.4.5 and 4.5.6 suffer from a double free vulnerability [29], that can be used by a local user to execute arbitrary code in the context of the webserver or to cause denial of service conditions. The exploit overwrites the pointer to a destructor with a junk value to cause denial of service. Our approach successfully identified that the pointer to destructor was tainted and defeated the exploitation attempt.

- *ghttpd `Log()` Buffer Overflow Vulnerability.* A stack overflow vulnerability [2] exists in ghttpd version 1.4.3 and lower which allows a remote user to execute arbitrary code with the privileges of the web server. The overflow occurs in `Log()` when the argument to a GET request overruns a 200-byte stack buffer. In order to test our approach against non-control data attacks, we modified the publicly available exploit to overwrite the stored `ESI` register on the stack, which is a later copied to a pointer to the URL requested by the client. By overwriting the stored `ESI` register, it is made to point to the URL `/cgi-bin/../../../../bin/sh` in order to execute a shell. Our approach correctly detected the argument to the `execlp()` was tainted and defeated the attack.

- *wuftpd `SITE EXEC` Format String Vulnerability.* wuftpd version 2.6.0 and earlier suffer from a format string vulnerability [61] in `SITE EXEC` implementation that allows arbitrary code execution. We modified the publicly available exploit for this vulnerability to overwrite the cached copy of user ID `pw->pw_uid` with 0 so as to disable the server's ability to drop privileges. The attack was detected as the format string argument was found to be tainted.

- *SIDVault `Simple_Bind()` Function Remote Buffer Overflow Vulnerability.* SIDVault LDAP server version 2.0e and earlier suffer from a buffer overflow [7], that can be used by a remote user to gain root privileges. The exploit overwrites the stored return address with the address of the injected shellcode. Our approach successfully identified that the stored return address was tainted and defeated the exploitation attempt.

Note on SIDVault. We would like to mention here that our approach was tested on SIDVault LDAP server buffer overflow attack [7], which is a proprietary software available without source code. Whereas our approach is applicable to such proprietary software available without source code, other approaches that rely on source code such as Xu et al's approach [67] will not be able to defend against attacks in such cases.

Table 5.3: Results of effectiveness evaluation

CVE#	Program	Attack Type	Overwrite Target	Description	Detected
Control Data Attacks					
-	Synthetic	Stack Smashing	Stored return address	Stack Smashing Attack	✓
-	Synthetic	Heap Overflow	Boundary tags	Heap Overflow Attack	✓
-	Synthetic	Format String Attack	GOT entry	Format String Attack	✓
CVE-2003-0201	samba	Buffer Overflow	Stored return address	Buffer overflow in `call_trans2open()` function	✓
CVE-2001-0010	bind	Buffer Overflow	Stored return address	Buffer overflow in `nslookupComplain()` routine	✓
CVE-2000-0763	xlock	Format String Attack	Stored return address	Format string vulnerability when using `-d` command line switch	✓
CVE-2007-1711	PHP	Double free attack	Pointer to destructor	Double free vulnerability using `session_decode()` function	✓
CVE-2007-4566	SIDVault	Buffer Overflow	Stored return address	Buffer Overflow in `Simple_Bind()` function	✓
Non-Control Data Attacks					
-	Synthetic	Buffer Overflow	Array in a structure	Non-control data attack	✓
CVE-2001-0820	ghttpd	Buffer Overflow	Stored `ESI` register	Non-control data attack	✓
CVE-2000-0573	wuftpd	Format String Attack	Cached copy of user ID	Non-control data attack	✓

5.2.2 Performance Evaluation

To test the performance overhead of our approach, we used our approach on several CPU intensive programs:

- **bc** - bc is an interactive algebraic language with arbitrary precision, with several extensions including multi-character variable names, and full Boolean expressions. The test was to calculate the factorial of 600.

- **Enscript** - Enscript converts ASCII files to PostScript, HTML, RTF or Pretty-Print and stores generated output to a file or sends it directly to the printer. The test was to convert a 5.5 MB text file to postscript.

Table 5.4: Comparison of overhead incurred by our approach with that of Xu et al's approach

Program	**Test**	**Overhead**	
		Xu et al's approach [67]	**Our approach**
bc-1.06	Find factorial of 600	61%	42.84%
enscript-1.6.4	Convert a 5.5 MB text file into a PS file	58%	28.63%
bison-1.35	Parse a Bison file for C++ grammer	78%	32.02%
gzip-1.3.3	Compress a 12 MB file	106%	45.38%

Table 5.5: Feature Comparison with previous approaches

Feature	**TaintCheck [47]**	**TaintTrace [21]**	**LIFT [51]**	**Chen et al's approach [66]**	**Xu et al's approach [67]**	**Our approach**
Works without additional hardware	✓	✓	✓	×	✓	✓
Works without source code	✓	✓	✓	✓	×	✓[2]
Detects non control data attacks	×	×	×	✓[3]	✓	✓

- **Bison** - Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a program to parse that grammar. The test was to parse a bison file for C++ grammer.

- **gzip** - Gzip is the standard file compression utility. The test was to compress a 12 MB file.

We performed all the tests 10000 times and took the averages of the results. We compare our results to that of Xu et al's approach, which has the lowest overhead among all the previous approaches. Our approach incurred an average overhead of 37.2% while that of Xu et al's approach [67] was 76%, which clearly shows an improvement by an order of magnitude. We present the test results in Table 5.4 and in Table 5.5 we present a detailed feature comparison of our approach with previous approaches.

[2] Requires debugging information

[3] Can only detect non-control data attacks where a pointer is tainted

[4] Performance results were not available

5.3 Discussion

In this section, in the context of our approach, we discuss ramifications of different security threats and address concerns regarding protection of the shadow memory.

Ramifications of Varied Security Threats. In the present day scenario, attacks on web applications, such as XSS and SQL injection, do contribute to the majority of the attacks today, however, the impact of such attacks is limited than that of memory corruption attacks. For e.g., for a server having both an XSS vulnerability and a buffer overflow vulnerability, exploitation of XSS flaw may lead to credit card details of a user being stolen, while exploitation of buffer overflow may lead to credit card details of all the users being stolen in a single attempt. Furthermore, exploitation of XSS would give an attacker only a limited (user-level) access on the server while exploitation of buffer overflow would give him superuser access (typically) on the server and it would even give him access to the corporate intranet. It is important to note that, memory corruption attacks have been used by worms such as CodeRed and Slammer, which plague the Internet in a matter of few minutes. Defending from web application attacks is easier due to the availability of the source code. However defending from memory corruption attacks is not easy due to source code unavailability, performance considerations and other factors. Construction of exploits for memory corruption attacks requires high expertise and experience. However, construction of exploits for XSS and SQL injection is fairly easy. This is easily justified by the fact that a single zero-day exploit for a Microsoft platform costs between 10000 - 100000 USD, in the underground market. However, several freeware tools are available for exploiting XSS and SQL injection vulnerabilities.

Protection of Shadow Memory. For protecting the shadow memory we used an approach similar to that used in [51]. Our shadow memory mapping strategy can prevent attackers from directly modifying shadow memory as long as the attackers' code is under the control of Pin. The reason is that when attackers issue an instruction to access an address `addr` in the shadow memory, the instrumented code will access memory at `addr+shadow_base`, which is beyond the boundary of the shadow memory area. This will cause an invalid memory access exception. Note that an attacker might try to access the shadow memory by using a large value of `addr` so that `addr+shadow_base` overflows the 32-bits of the integer used to hold the sum. To counter such attacks, we check if the sum `addr+shadow_base` leads to an overflow condition.

Implicit Information Flows. Implicit information flow occurs when the values of certain variables are related by virtue of program logic, even though there are no assignments between them. For example, the following code snippet illustrates such a case.

```
if(x==0) y=0; if(x==1) y=1;
```

The above code fragment is equivalent to the assignment `y=x`. Therefore, if `x` is tainted (untainted), `y` is tainted (untainted) as well, although there is no direct assignment from `x` to `y`. But since the value of `y` is assigned from a constant, it would be marked as untainted, irrespective of the tag of `x`. Our approach supports some of such basic implicit information flows, such as the use of translation tables, which are sometimes used for decoding using table lookups, such as `y = table[x]`. In such a case, the array index `x` determines the value of `y`. To handle such cases, our approach marks the result of an array access as tainted (untainted) whenever the index is tainted (untainted). Much more complex implicit flows can be supported using a notion of noninterference, which however is too conservative and leads to high false positives.

Limitations. Our approach is built on top of TIED. Although TIED does not require source code of the programs, it requires them to be compiled with debugging information. A limitation of our approach is that it does not work for stripped binaries, i.e. programs without debugging and other symbol table information. However, we believe it is not such a major limitation as many operating systems distributions, for example Microsoft Windows and Fedora family of operating systems make debugging information available as separate packages [8, 1]. Requirement for debugging information may also ease reverse engineering in some cases. However, we argue that our approach requires only information related to the variables. The debugging information related to code can safely be removed from the program binaries. Even in the presence of debugging information, obfuscations can still be applied to the code present in the binary to make it harder to reverse engineer. Moreover, some approaches [11, 12] have been proposed in the past that are able to discover variables in stripped binaries with a high accuracy. We believe using a similar technique would make our approach independent of the debugging information present in the executables, making it work even for stripped binaries.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Memory errors in C and C++ programs have been known for decades and are one of the oldest classes of software vulnerabilities. Researchers have been working on memory error protection mechanisms for decades. Nonetheless, it seems that this kind of vulnerability is far from being completely defeated.

This thesis presented two techniques to defeat memory corruption attacks. Recognizing the effectiveness of anomaly-based detection techniques and taint-tracking, we proposed two approaches that are able to deal effectively with memory corruption attacks. In particular, our first approach, *FormatShield* uses anomaly detection to defeat format string attacks. By identifying vulnerable call sites and the associated exploitable contexts, it detects malicious input at the such vulnerable call sites with an exploitable context. It defends against all types of format string attacks, *i.e.*, arbitrary memory read and arbitrary memory write attempts, and incurs minimal performance overhead. *Coarse Grained Dynamic Taint Analysis*, the second approach proposed by this thesis, takes advantage of taint tracking and security policies to defeat a wide range of memory corruption attacks. The proposed technique first uses the debugging information present in a program binary to dump location and size information of the variables back in the binary. Then, by propagating taint on these variables, together with security policies, our approach is able to defeat any kind of memory corruption attacks, including non-control data attacks.

6.2 Future Work

Although Coarse Grained Dynamic Taint Analysis defends against several types of memory corruption attacks, it requires programs to be compiled with debugging information. One of the future works in our approach involves extracting variable information effectively without the need for debugging information. Also, we are exploring ways of incorporating more application context information so as to detect other attacks on critical variables that are identified by the application developer. Some approaches [11, 12] have been proposed in the past that are able to discover variables in stripped binaries with a high accuracy. We believe using a similar technique would make our approach independent of the debugging information present in the executables, making it work even for stripped binaries.

Bibliography

[1] Fedora 7 Debuginfo Packages. http://ftp.cs.pu.edu.tw/Linux/Fedora/updates/testing/7/i386/debug/.

[2] ghttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960.

[3] PaX. http://pax.grsecurity.net.

[4] PaX Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[5] Perl security manual page. http://www.perldoc.com.

[6] Samba 'call_trans2open' remote buffer overflow vulnerability. http://www.securityfocus.com/bid/7294.

[7] SIDVault Simple_Bind Function Multiple Remote Buffer Overflow Vulnerabilities. http://www.securityfocus.com/bid/25460.

[8] Windows Symbol Packages. http://www.microsoft.com/whdc/DevTools/Debugging/symbolpkg.mspx.

[9] Stack shield - a "stack smashing" technique protection tool for linux. 2006. http://www.angelfire.com/sk/stackshield/.

[10] K. Avijit, P. Gupta, and D. Gupta. TIED, Libsafeplus: Tools for Runtime Buffer Overflow Protection. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

[11] G. Balakrishnan and T. W. Reps. Analyzing Memory Accesses in x86 Executables. In *CC*, pages 5–23, 2004.

[12] G. Balakrishnan and T. W. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, pages 1–28, 2007.

[13] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[14] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 281–289, New York, NY, USA, 2003. ACM.

[15] F. Bellard. Tcc: Tiny C compiler. October 2003. http://www.tinycc.org.

[16] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[17] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[18] bind. xlockmore User Supplied Format String Vulnerability. http://www.securityfocus.com/bid/1585.

[19] P. Broadwell, M. Harren, and N. Sastry. Scrash: A System for Generating Secure Crash Information. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[20] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control Data Attacks are Realistic Threats. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[21] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[22] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[23] T. I. S. Committee. Executable and Linking Format (ELF) Specification. 1995. version 1.2.

[24] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. Formatguard: Automatic Protection from Printf Format String Vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[25] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[26] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, Berkeley, CA, USA, 1998. USENIX Association.

[27] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[28] T. Durden. Bypassing PaX ASLR Protection. *Phrack Magazine*, June 2002. http://www.phrack.org/phrack/59/p59-0x09.

[29] S. Esser. PHP Session_Decode Double Free Memory Corruption Vulnerability. http://www.securityfocus.com/bid/23121.

[30] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. June 2000. Technical report, IBM Research Divison, Tokyo Research Laboratory.

[31] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *Software, IEEE*, 19(1):42–51, Jan/Feb 2002.

[32] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1996. IEEE Computer Society.

[33] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM.

[34] A. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 104–114, May 1998.

[35] D. Jacobowitz. Multiple Linux Vendor rpc.statd Remote Format String Vulnerability. http://www.securityfocus.com/bid/1480.

[36] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[37] R. W. M. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1997.

[38] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.

[39] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[40] L. C. Lam and T. cker Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[41] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[42] Z. Lin, N. Xia, G. Li, B. Mao, and L. Xie. Transparent Run-Time Prevention of Format-String Attacks via Dynamic Taint and Flexible Validation. In *ISC*, pages 17–31, 2006.

[43] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, pages 190–200, 2005.

[44] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[45] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-Safe Retrofitting of Legacy Code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, USA, 2002. ACM.

[46] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[47] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[48] A. Osborne and J. McDonald. ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/2302.

[49] Parasoft. Insure++: Automatic runtime error detection. 2004. http://www.parasoft.com.

[50] N. Provos. Improving Host Security with System Call Policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[51] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[52] M. F. Ringenburg and D. Grossman. Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 354–363, New York, NY, USA, 2005. ACM.

[53] T. Robbins. Libformat. http://www.wiretapped.net/ fyre/software/libformat.html.

[54] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[55] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2001. IEEE Computer Society.

[56] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[57] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[58] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[59] @stake Inc. tcpflow 0.2.0 format string vulnerability. August 2003. http://www.securityfocus.com/advisories/5686.

[60] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[61] tf8. Wu-Ftpd Remote Format String Stack Overwrite Vulnerability. http://www.securityfocus.com/bid/1387.

[62] T. Tsai and N. Singh. Libsafe: Transparent System-Wide Protection against Buffer Overflow Attacks. 2002.

[63] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[64] Z. Xiao. An Automated Approach to Software Reliability and Security. 2003. Invited Talk, Department of Computer Science, University of California at Berkeley.

[65] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent Runtime Randomization for Security. In *SRDS*, 2003.

[66] J. Xu and N. Nakka. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[67] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[68] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.