A Dissertation

On

"Boundary Exploitation Overflow Preventive"

Submitted in the partial fulfillment of the requirements

For the award of Degree of

Master of Technology

In

Software Technology

By

Kushal Ahuja
(2K12/SWT/007)

Under the guidance of

Vinod Kumar

Department of Computer Science & Engineering, DTU

DEPARTMENT OF SOFTWARE ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

BAWANA ROAD, DELHI

DECLARATION

I hereby want to declare that the thesis entitled "**Boundary Exploitation Overflow Preventive"** which is being submitted to the **Delhi Technological University**, in partial fulfillment of the requirements for the award of degree in **Master of Technology in Software Technology** is an authentic work carried out by me. The material contained in this thesis has not been submitted to any institution or university for the award of any degree.

____

Kushal Ahuja

Department of Computer Science & Engineering

Delhi Technological University,

Delhi.

CERTIFICATE

Delhi Technological University
(Government of Delhi)
Bawana Road, New Delhi-42

This is to certify that the thesis entitled **"Boundary Exploitation Overflow Preventive"** done by **Kushal Ahuja** (Roll Number: **2K12/SWT/007**) for the partial fulfillment of the requirements for the award of degree of **Master of Technology** Degree in **Software Technology** in the **Department of Computer Science & Engineering**, Delhi Technological University, New Delhi is an authentic work carried out by her under my guidance.

Project Guide:
Vinod Kumar
Associate Professor,
Department of Computer Science & Engineering
Delhi Technological University, Delhi

ACKNOWLEDGEMENT

I take this opportunity to express my deep sense of gratitude and respect towards my guide **Vinod Kumar, Associate Professor , Department of Computer Science & Engineering.**

I am very much indebted to him for his generosity, expertise and guidance i have received from him while working on this project. Without his support and timely guidance the completion of the project would have seemed a far –fetched dream. In this respect I find myself lucky to have my guide. He have guided not only with the subject matter, but also taught the proper style and techniques of documentation and presentation. I would also like to take this opportunity to present my sincere regards to **Ms. Raju Gupta**, for extending her support and valuable Guidance.

Besides my guides, I would like to thank entire teaching and non-teaching staff in the Department of Computer Science & Engineering, DTU for all their help during my tenure at DTU. Kudos to all my friends at DTU for thought provoking discussion and making stay very pleasant.

Kushal Ahuja
M.Tech Software Technology
2K12/SWT/007

ABSTRACT

Buffer overflow continues to be one of the leading vulnerabilities that plague the software industry. Buffer overflow as the name suggests results because software may potentially allow operation, such as reading or writing, to be performed at addresses not intended by the developer. Buffer overflow typically affects unsafe languages such as C and C++ as these languages don't perform bound checks on arrays and pointer references and they focus more on programming efficiency and code length than on the security aspects. Languages like Java that perform bound check are not prone to buffer overflows arising out of unbounded copy. Range of possible buffer overflow exploits is based on degree of control by attacker achieved. It may range from "Denial of Service" attack (resulting in system crash) to "Arbitrary Code Execution" (to hijack control of your system).

In this report, we typically explore the kinds of programming vulnerabilities which result into Buffer Overflow, how an attacker could exploit them, how a best programmer could detect them and inhibit or prevent exploitation of those vulnerabilities. This report details one method which is based on "Execution Space Protection" to prevent buffer overflow vulnerability from being exploited. To understand this method, report is complemented with basic Process Memory Layout Details, Linux Internals like system calls, system call table, Interrupt Descriptor table (IDT), Virtual memory area, and Basic Kernel Module Programming. With all this knowledge simulated, we'll go through design details of a Kernel Module to protect buffer overflow vulnerability from being exploited. This kernel module works by overwriting the system call table function pointers with its own function. Doing so would direct the control to the module function whenever a system call is made and we can do the necessary processing to know whether system call originated from writable region of memory. If so, we can kill the system call without letting it hijack control of our system.

Broad Academic Area of Work: System/Network Security

Keywords: Buffer Overflow, Stack Frame, Shell Code, System Call Table.

Signature of the Student	**Signature of the Supervisor**
Name: ________________	**Name:** ___________________
Date:	**Date:**
Place:	**Place:**

Table of Contents

ABSTRACT ..... **iv**

1 Introduction **1**

1.1 Objective 1

1.2 What is a Buffer? 1

1.3 What is a Buffer Overflow? 1

1.4 What is a Buffer Overflow Exploit? 2

1.4.1 Stack Based: 2

1.4.2 Heap Based: 3

1.5 Why are Buffer Overflows so Common? 3

2 Process Memory Layout **5**

2.1 Typical Memory Layout 5

2.2 Stack Frame 6

3 Buffer Overflow Exploitation, Detection and Prevention **9**

3.1 Buffer Overflow Vulnerabilities 9

3.1.1 Unbounded Transfer 9

3.1.2 Improper Termination 10

3.1.3 Buffer Underwrite 11

3.1.4 Integer Overflow/Underflow 11

3.2 What It Takes to Write an Exploit? 12

3.2.1 Calculate Length to Return Address on the Stack 12

3.2.2 Create Attack Code 14

3.3 Buffer Overflow Detection 14

3.3.1 Static Analysis 14

3.3.2 Compile and Runtime Analysis 15

3.4 Buffer Overflow Protection 15

3.4.1 Canary-Based defenses 16

3.4.2 Address Space Layout Randomization (ASLR) 16

3.4.3 Executable Space Protection (ESP) 17

3.4.4 Safe C Lib 17

4 Kernel Module Programming **19**

4.1 What is a Kernel Module? 19
4.2 Writing a Kernel Module 19
4.3 Compiling and Building Modules 22
4.4 Loading and Unloading Modules 24
5 Functional Specifications for BOEP 26
6 Hooking System Call Table 28
6.1 System Call 28
6.2 System Call Table 29
6.3 X86 Exceptions 29
6.4 System Call Handler and Service Routine 30
6.5 Interrupt Descriptor Table 32
6.6 Issuing a System Call via *int 0x80* Instruction 33
7 Protection System Call Handler 35
8 Summary, Conclusion and Future Work 37
8.1 Summary 37
8.2 Conclusion 37
8.3 Future Work 37
Bibliography 38
Appendix A 39

List of Figures

1.1 ‘A’ and ‘B’, Two Adjacent Memory Locations........1

1.2 ‘B’ Overwritten By Unsafe Buffer Copy On ‘A’........2

2.1 Typical Memory Layout........5

2.2 Typical Stack Frame........7

3.1 Overwrite Stack Frame with Shell Code and a New Return Address........10

3.2 Return-to-Stack Technique........13

3.3 Jump-to-Register Technique........14

3.4 Compile and Runtime BO Detection Analysis Flow Diagram........15

3.5 Stack Guard (Stack-Smashing Protection)........16

5.1 BOEP Execution Flow Chart........26

6.1 Invoking a System Call........30

6.2 Trap Gate Descriptor in IDT........31

A.1 The Process Descriptor and Task List........38

List of Tables

2.1 Function Symbol Table…………………………………………………………...6

3.1 ASLR Implementation Options on Various OS…………………………………….17

6.1 System Call Implementation Directory for Various Kernel Services……………...….28

6.2 Various x86 Exceptions…………………………………………………….………28

6.3 Linux IDT Descriptors……………………………………………………………...31

6.4 IDT System Gate Descriptor after set_system_gate(0x80, &system_call)…………..…..32

List of Acronyms

ASLR	Address Space Layout Randomization
BO	Buffer Overflow
BOEP	Buffer Overflow Exploit Prevention
EIP	Extended Instruction Pointer
ESP	Execution Space Protection / Extended Stack Pointer
IDT	Interrupt Descriptor Table
IDTR	Interrupt Descriptor Table Register
VMA	Virtual Memory Area

Chapter 1

1 Introduction

Buffer Overflow – The software may potentially allow operations, such as reading and writing at addresses not intended by the developer.

Common Weakness Enumeration (http://cwe.mitre.org)

1.1 Objective

As the project title reflects, through this project we will describe the buffer overflow vulnerabilities arising out of unbounded buffer copy, improper termination etc. and how an attacker can hijack control of our system by exploiting these software vulnerabilities. We will also brief about the present techniques to prevent buffer overflows from being exploited and detail about one which is implemented as a kernel module as a part of this project.

1.2 What is a Buffer?

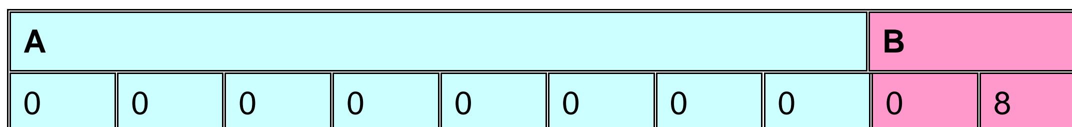
A buffer can be formally defined as "a contiguous block of computer memory that holds more than one instance of the same data type." In C and C++, buffers are usually implemented using arrays and memory allocation routines like malloc() and new. An extremely common kind of buffer is simply an array of characters.

1.3 What is a Buffer Overflow?

In computer security and programming, a buffer overflow, or buffer overrun, is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer. The result is that the extra data overwrites adjacent memory locations. The overwritten data may include other buffers, variables and program flow data, and may result in erratic program behavior, a memory access exception, program termination (a crash), incorrect results or — especially if deliberately caused by a malicious user — a possible breach of system security.

A buffer overflow occurs when data written to a buffer, due to insufficient bounds checking, corrupts data values in memory addresses adjacent to the allocated buffer. Most commonly this occurs when copying strings of characters from one buffer to another.

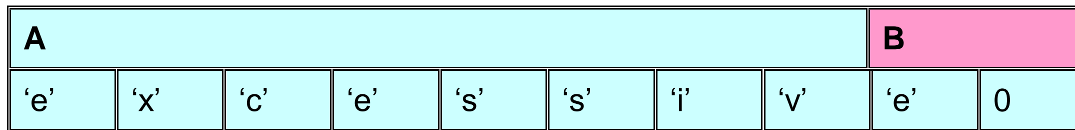
Example:
Suppose, a program has defined two variables items which are adjacent in memory: an 8-byte-long string buffer, A, and a 2-byte integer, B. Initially, A contains nothing but zero bytes, and B contains the number length of buffer A i.e. 8.



A								**B**	
0	0	0	0	0	0	0	0	0	8

Figure 1.1: ‘A’ and ‘B’, Two Adjacent Memory Locations

Now, the program attempts to store the character string "excessive" in the A buffer, followed by a zero byte to mark the end of the string. By not checking the length of the string, it overwrites the value of B:



A								B	
‘e’	‘x’	‘c’	‘e’	‘s’	‘s’	‘i’	‘v’	‘e’	0

Figure 1.2: ‘B’ Overwritten by Unsafe Buffer Copy on ‘A’

Although the programmer did not intend to change B at all, B's value has now been replaced by a number formed from part of the character string. In this example, on a big-endian system that uses ASCII, "e" followed by a zero byte would become the number 25856. If B was the only other variable data item defined by the program, writing an even longer string that went past the end of B could cause an error such as a segmentation fault, terminating the process.

1.4 What is a Buffer Overflow Exploit?

A buffer overflow may happen accidentally during the execution of a program. When this happens, however, it is very unlikely that it will lead to a security compromise of the system. Most often the clobbering of information in areas adjacent to the buffer will cause the program to crash or produce obviously incorrect results. In a buffer overflow attack, on the other hand, the objective of the attacker is to use the vulnerability to corrupt information in a carefully designed way in order to execute attack code previously planted by the attacker. Buffer overflows can be triggered by inputs specifically designed to execute malicious code or to make the program operate in an unintended way. As such, buffer overflows cause many software vulnerabilities and form the basis of many exploits. Buffer overflow typically affects unsafe languages such as C and C++ as these languages don’t perform bound checks on arrays and pointer references and they focus more on programming efficiency and code length than on the security aspects. Languages like Java which perform bound check are not prone to buffer overflows arising out of unbounded copy. Range of possible buffer overflow exploits is based on degree of control by attacker achieved. It may range from “Denial of Service” attack (essentially results in system crash) to “Arbitrary Code Execution” (to hijack control of your system). The techniques to exploit a buffer overflow vulnerability vary per architecture, operating system and memory region. For example, there are two main types of buffer overflow attacks as per memory region:

1.4.1 Stack Based:

Stack-based buffer overflows are by far the most common. In a stack-based buffer overrun, the program being exploited uses a memory object known as a stack to store user input. Normally, the stack is empty until the program requires user input. At that point, the program writes a return memory address to the stack and then the user's input is placed on top of it. When the stack is processed, the user's input gets sent to the return address specified by the program. However, a stack does not have an infinite potential size. The programmer who develops the code must reserve a specific amount of space for

the stack. If the user's input is longer than the amount of space reserved for it within the stack, then the stack will overflow. This in itself isn't a huge problem, but it becomes a huge security hole when combined with malicious input. While buffer overflow examples can be rather complex, it is possible to have very simple, yet still exploitable, stack-based buffer overflows:

```
#define BUFSIZE 256
int main(int argc, char **argv) {
char buf[BUFSIZE];
strcpy(buf, argv[1]);
}
```

Above was a very classic example of buffer overflow. Typically, a technically inclined and malicious user may exploit stack-based buffer overflows to manipulate the program in one of several ways:

- By overwriting a local variable that is near the buffer in memory on the stack to change the behavior of the program which may benefit the attacker.
- By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user input filled buffer.
- By overwriting a function pointer, or exception handler, which is subsequently executed.

1.4.2 Heap Based:

A buffer overflow occurring in the heap data area is referred to as a heap overflow and is exploitable in a different manner to that of stack-based overflows. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data. Exploitation is performed by corrupting this data in specific ways to cause the application to overwrite internal structures such as linked list pointers. The canonical heap overflow technique overwrites dynamic memory allocation linkage (such as malloc meta data) and uses the resulting pointer exchange to overwrite a program function pointer. A simple example showing Heap-based buffer overflow is:

```
#define BUFSIZE 256
int main(int argc, char **argv) {
char *buf;

buf = (char *)malloc(BUFSIZE);
strcpy(buf, argv[1]);
}
```

1.5 Why are Buffer Overflows so Common?

In nearly all computer languages, both old and new, trying to overflow a buffer is normally detected and prevented automatically by the language itself (say, by raising

an exception or adding more space to the buffer as needed). But there are two languages where this is not true: C and C++. Often C and C++ will simply let additional data be scribbled all over the rest of the memory, and this can be exploited to horrific effect. What's worse, it's actually more difficult to write correct code in C and C++ to always deal with buffer overflows; it's very easy to accidentally permit a buffer overflow. These might be irrelevant facts except that C and C++ are very widely used; for example, 86% of the lines of code in Red Hat Linux 7.1 are in either C or C++. Thus, there's a vast amount of code that's vulnerable to this problem because the implementation language fails to protect against it.

This isn't easily fixed in the C and C++ languages themselves. The problem is based on fundamental design decisions of the C language (particularly how pointers and arrays are handled in C). Since C++ is a mostly compatible superset of C, it has the same problems. Fundamentally, any time your program reads or copies data into a buffer, it needs to check that there's enough space before making the copy. An exception is if you can show it can't happen -- but often programs are changed over time that make the impossible possible.

Another problem is that C and C++ have very weak typing for integers and don't normally detect problems manipulating them. Since they require the programmer to do all the detecting of problems by hand, it's easy to manipulate numbers incorrectly in a way that's exploitable. In particular, it's often the case that you need to keep track of a buffer length, or read a length of something. But what happens if you use a signed value to store this -- can an attacker cause it to "go negative" and then later have that data interpreted as a really large positive number? When numeric values are translated between different sizes, can an attacker exploit this? Are numeric overflows exploitable? Sometimes the way integers are handled creates a vulnerability.

Chapter 2

2 Process Memory Layout

Here, in this chapter we would discuss about how the process memory is laid out when a process is loaded into the memory. We will go in more details of a memory unit called "Stack" and develop our understanding on Stack Frame by having some discussion about assembly equivalent of C code.

2.1 Typical Memory Layout

When a program is executed, its various compilation units are mapped in memory in a well-structured manner. The kernel arranges pages into blocks that share certain properties, such as access permissions. These blocks are called *memory regions*, *segments*, or *mappings*. Figure below reflects typical memory layout.

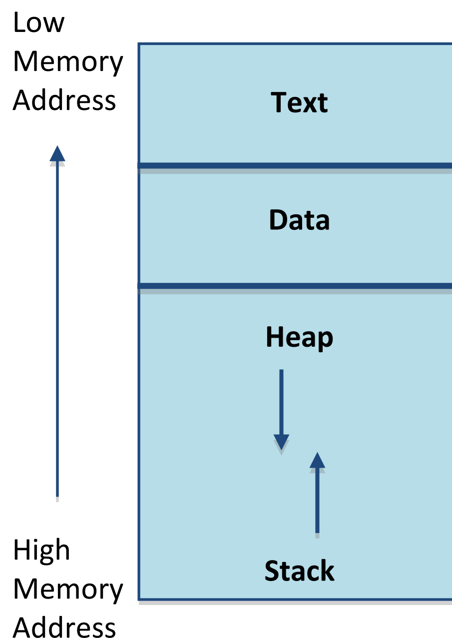


Figure 2.1: Typical Memory Layout

Text Segment - contains primarily the program code, i.e., a series of executable program instructions. Other than program code, it contains string literals, constant variables, and other read-only data. The code execution is non-linear, it can skip code, jump, and call functions on certain conditions. Therefore, we have a pointer called EIP, or instruction pointer. The address where EIP points to always contains the code that will be executed next.

Data Segment - is an area of memory containing both initialized and uninitialized global data. Its size is provided at compilation time.

Stack Segment - is used to pass data (arguments) to functions and as a space for variables of functions. It is shared by the stack (which is a LIFO data structure) and heap that, in turn, is allocated at run time. The stack is used to store function call-by arguments, local variables and values of selected registers allowing it to retrieve the program state. The heap holds dynamic variables. To allocate memory, the heap uses the malloc function or the new operator.

2.2 Stack Frame

Let us try to explore more about the stack area shown in Fig. 2.1, with the help of a sample program when compiled using GNU compiler:

```
main() {
       int i, j, k;
       i = 400
       j = 500
       k = add(i+j)
}
int add(int a, int b) {
      int c;
      c = a+b;
}
```

Compiler takes source code and emits assembly code. The following steps are involved in compilation of the above code into GNU assembly equivalent:

1. Identify executable and non-executable statements within the source code.
2. Construct a local variable table and resolve all non-executable statements.
3. Convert executable statements into assembly equivalents as per GNU assembly template.

In step (1), when we look up the source code, we'll find two kinds of statements:

1) Executable - which need CPU time
2) Non-executable - which don't need CPU time. These statements like local variable declarations find their place on stack. Stack is a LIFO structure.

In step (2), we create a table for non-executable statements called local variables table or symbol table. Columns of this table are Symbol Name, Type, Composition(memory space needed), address. Every function has its own symbol table. For above sample code's main() function, symbol table appears like:

Table 2.1: Function Symbol Table

Symbol Name	Type	Composition	Address
i	int	4	-12 (%ebp)
j	int	4	-8 (%ebp)
k	int	4	-4 (%ebp)
Total		**12**	

Extended Base Pointer (ebp) and Extended Stack Pointer (esp), are the CPU registers that are referenced throughout the assembly code. EBP contains virtual address, the address at compile time. On top of EBP, resides the stack. Local variables and arguments are addressed with respect to EBP. Arguments stay in the high memory region and local variables in low memory region with respect to EBP. Therefore, EBP is also known as Stack Frame Pointer. For the sample code, first variable is at -4 offset, second at -8 and third at -12 offset from EBP as stack grows upwards.

In step (3), we have to convert executable code into assembly equivalent. Template for assembly equivalent looks something like below:

```
Function add():
        Prologue;
        Function body;
        Epilogue;
```

Prologue (denotes opening brace '{') and epilogue (denotes closing brace '}') are fixed for every function.

```
Prologue:
   pushl          %ebp      } 'enter' instruction on x86
   movl %esp, %ebp
Epilogue:
   movl   %ebp,   %esp
   popl           %ebp      } 'leave' instruction on x86

   ret
```

Function call to "add" ('call' instruction in x86) would push the arguments on the stack typically from right to left. Then the Return Address from where 'main' will resume its execution after 'add' returns, is pushed on to the stack. Then the function prologue gets executed and the stack is allocated for the local variables of 'add'. When 'add' finishes its execution, epilogue, which is just reverse of prologue, gets executed and 'ret' instruction pops back the return address from stack for 'main' to resume. So the stack frame for 'add' appears like:

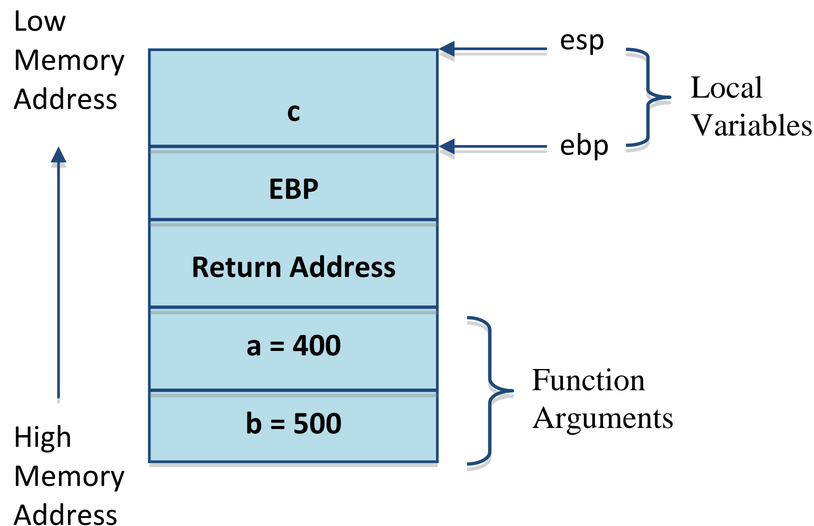


Figure 2.2: Typical Stack Frame

EBP which is pushed onto the stack as part of prologue, saves the Frame Pointer for caller function. That is why, in epilogue we pop it back before returning back to ‘main’.

Chapter 3

3 Buffer Overflow Exploitation, Detection and Prevention

3.1 Buffer Overflow Vulnerabilities

Besides unbounded methods like strcpy(), strcat(), sprintf(), gets() and memcpy() etc., which are so called the reasons of buffer overflows, there are other reasons as well which may make a program vulnerable to be exploited by the attacker. Let's discuss some few of them here.

3.1.1 Unbounded Transfer

Let us consider a sample code snippet that represents unbounded copy. It is a security hole that can be exploited by the attacker.

```
void foo (char *user_arg)  {
        char buf[FIXED_SIZE];
        -----
        -----
        strcpy(buf, user_arg)
}
int main(int argc, char *argv[])  {
        -----
        foo(argv[1])
}
```

This sample code has all the characteristics to indicate a potential buffer overflow vulnerability: a local buffer and an unsafe function that writes to memory, the value of the first instruction line parameter with no bounds checking employed. As we are interactively accepting the input from the user and performing no bound check, attacker may take advantage out of this vulnerability. We are copying the user input 'user_arg' to a string 'buf' of fixed size without really validating it. The issue here is that the user may input a lot longer input and could go past the buffer 'buf'. This may result in overwriting other local variables (if present) as well as the Return Address. The attacker may input the shell code (attack code) to spawn a shell and then may manage to modify the return address stored on the stack in a way so that it points to the shell code. So if we are executing as root, instead of resuming our execution from 'main' after 'foo' finishes its execution, attacker manages to spawn a shell and hijack control of our system. Now as he is executing as root, he has all the privileges and do whatever he wants. Above said is clearly reflected in the figure below.

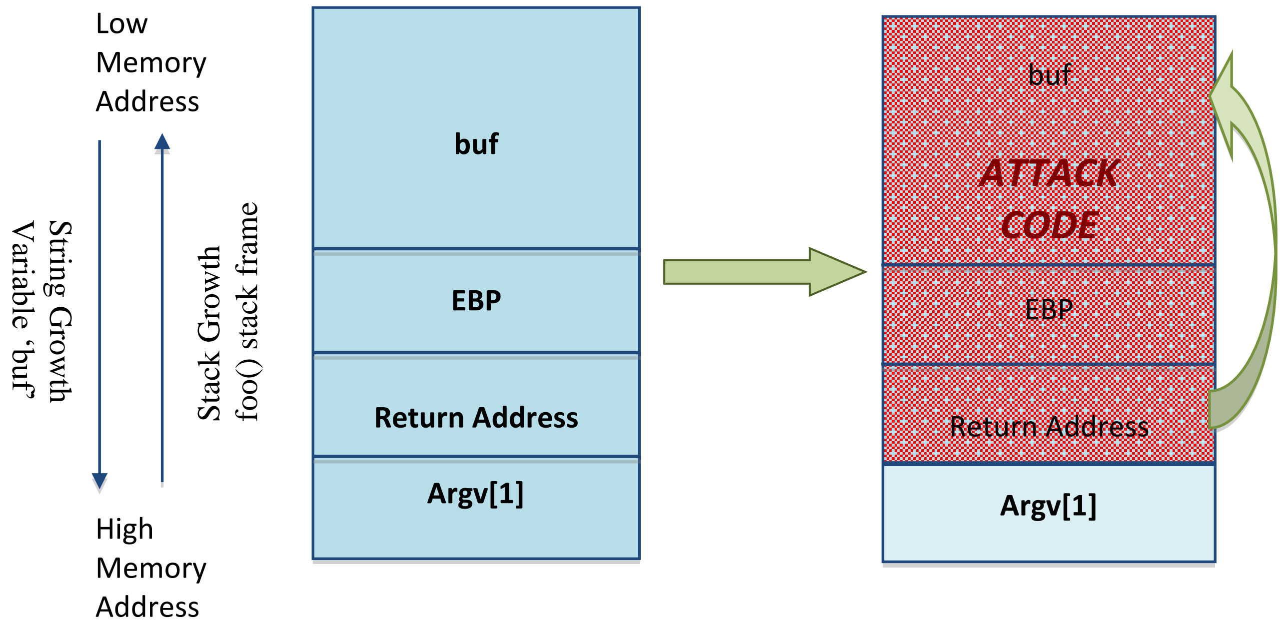


Figure 3.1: Overwritten Stack Frame with Shell Code and a New Return Address

Summarizing, we see that a buffer overflow attack usually consists of three parts:

1) The planting of the attack code into the target program;
2) The actual copying into the buffer which overflows it and corrupts adjacent data structures;
3) The hijacking of control to execute the attack code;

3.1.2 Improper Termination

(a) Mismatch between functions with different assumptions may cause buffer overflow to happen. Let's understand it taking another code snippet. The following code assumes that read() null terminates the string:

```
#define MAXLEN 1024
………

char output_buf[MAXLEN]
………

read(file, input_buf, MAXLEN);          //doesn't null terminate
strcpy(output_buf, input_buf)                   //requires null terminated input
```

Here data past (input_buf+MAXLEN) will be copied to output_buf until '\0' is found and it may cause buffer overflow as reflected in figure 3.1.

(b) Wrong assumptions on proper loop termination may cause buffer overflow to happen. *Blaster Worm (*also known as LOVSAN) was a computer worm which spread by exploiting the logic flaw (buffer overflow) in the DCOM RPC service for which a patch was already released. But worm was programmed to start a SYN flood against port 80 of windowsupdate.com, thereby creating a Distributed Denial of Service (DDoS) attack against the site. Below given is the vulnerable code snippet:

```
pwszServerName = wszMachineName;
LPWSTR pwsTemp = pwszPath+2;
While( *pwszTemp != L’’\\’ )
        *pwsServerName++ = *pwszTemp++
```

Here in this code snippet which Blaster worm exploited, pwszServerName is pointing to a local buffer ‘wszMachineName’ on stack. This code is looking for a Back Slash ‘\’ in wszMachineName and may get incremented past the buffer boundary if ‘\’ is not found.

3.1.3 Buffer Underwrite

Buffer underwrite signifies that the buffer area is written to memory locations prior the targeted buffer. This vulnerability was present in PHP 5.2.0 which allowed arbitrary code execution. Below given is the code snippet from header() method in PHP which first performs a whitespace trimming on the parameter.

```
/* cut of trailing spaces, linefeeds and carriage-returns */
  while(isspace(header_line[header_line_len-1]))
      header_line[--header_line_len]='\0';
```

The code trims the trailing whitespace by moving backward through the header and overwriting NULL bytes over the end. Unfortunately the trimming does not work correctly on an all whitespace string, because the move backward does not stop at the beginning of the string. Therefore the trimming operation will write NULL bytes in front of the allocated buffer when the bytes before the buffer start contain ASCII characters belonging to the whitespace charset. So it may lead to an exploit when memory is unlinked and may cause Heap corruption.

3.1.4 Integer Overflow/Underflow

An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc. Integer overflows generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also

high. Integer overflows can sometimes trigger buffer overflows that can be used to execute arbitrary code. The following code excerpt from OpenSSH 3.3 demonstrates a classic case of integer overflow:

```
nresp = packet_get_int(); if (nresp < 0) {
response = xmalloc(nresp*sizeof(char*));
for (i = 0; i > nresp; i++) response[i] = packet_get_string(NULL);
}
```

If nresp has the value 1073741824 and sizeof(char*) has its typical value of 4, then the result of the operation nresp*sizeof(char*) overflows, and the argument to xmalloc() will be 0. Most malloc() implementations will happily allocate a 0-byte buffer, causing the subsequent loop iterations to overflow the heap buffer response.

3.2 What It Takes to Write an Exploit?

As we said earlier, by exploiting the kind of vulnerabilities detailed above, attacker may inject its own code and execute it. As a programmer conversant with security issues, at the first line of caution we should first be able to avoid introducing software vulnerabilities described above by reading attacker's mind. To be able to do it, we should have an idea of how exploits may be written around the vulnerable code. Let's go through exercise of writing an exploit for a stack-based buffer overflow.

3.2.1 Calculate Length to Return Address on the Stack

First of all attacker needs to know how far the return address is placed on the stack from the start of buffer which expects user input. If the attacker has the source code of the attacked program it may be possible to determine exactly how big the buffer is and how far it is from the return address, determining how big the payload string must be. Access to the source code is nowadays quite common for many Operating Systems, e.g. Linux, OpenBSD, Free BSD, and even Solaris.
However, there is no need to have access to the source, or even knowledge of the exact details of how the attacked program works. The address of the attack code can be guessed, and through various techniques an approximate guess will do.

Return-to-Stack Technique:

The attack code could start with a long list of no operation instructions, so that control could be passed to any of these in order to correctly execute the crucial part of the attack code which spawns the shell and comes after the no ops. This collection of no-ops is referred to as the "NOP-sled" because if the return address is overwritten with any address within the no-op region of the buffer it will "slide" down the no-ops until it is redirected to the actual malicious code. This technique was already used in the Morris worm. Similarly, the tail of the payload string could consist of a repeated list of the guessed address of the attack code that we want to overwrite the return address with.

Because of the popularity of this technique many vendors of Intrusion prevention systems will search for this pattern of no-op machine instructions in an attempt to detect shell code in use. It is important to note that a NOP-sled does not necessarily contain only traditional no-op machine instructions; any instruction that does not corrupt

the machine state to a point where the shell code will not run can be used in place of the hardware assisted no-op. As a result it has become common practice for exploit writers to compose the no-op sled with randomly chosen instructions which will have no real effect on the shell code execution.

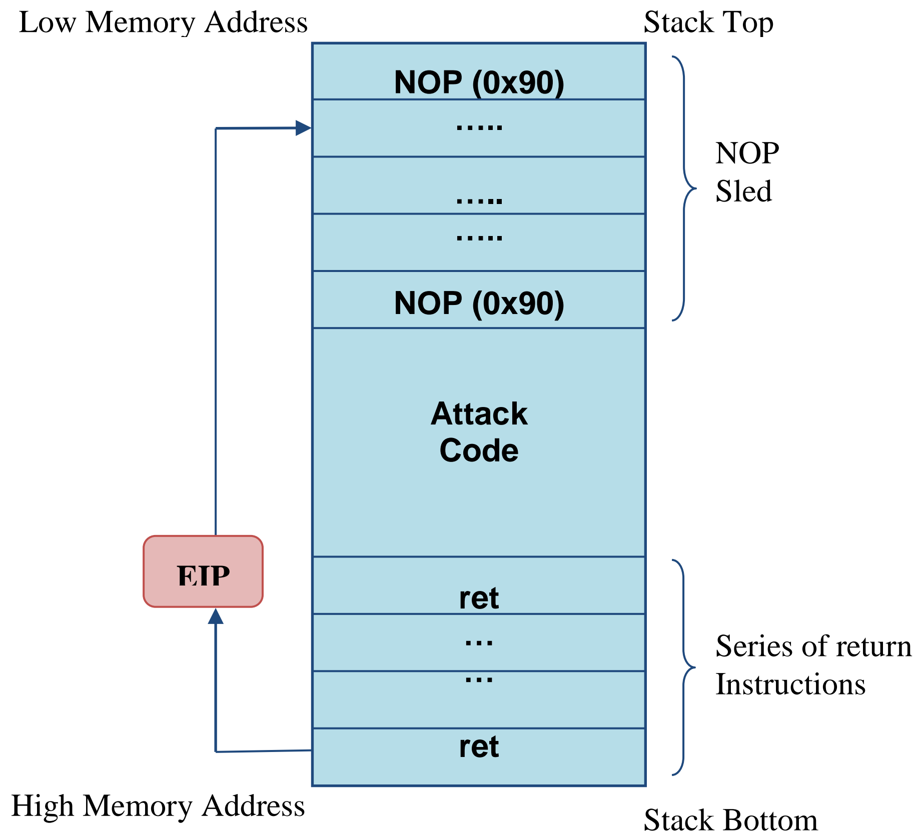


Figure 3.2: Return-to-Stack Technique

While this method greatly improves the chances that an attack will be successful, it is not without problems. Exploits using this technique still must rely on some amount of luck that they will guess offsets on the stack that are within the NOP-sled region.[8] An incorrect guess will usually result in the target program crashing and could alert the system administrator to the attacker's activities. Another problem is that the NOP-sled requires a much larger amount of memory in which to hold a NOP-sled large enough to be of any use. This can be a problem when the allocated size of the affected buffer is too small and the current depth of the stack is shallow (i.e. there is not much space from the end of the current stack frame to the start of the stack). Despite its problems, the NOP-sled is often the only method that will work for a given platform, environment, or situation; as such it is still an important technique.

Jump-to-Register Technique:

The "jump to register" technique allows for reliable exploitation of stack buffer overflows without the need for extra room for a NOP-sled and without having to guess stack offsets. The strategy is to overwrite the return pointer with something that will cause the program to jump to a known pointer stored within a register which points to the controlled buffer and thus the shellcode. For example if register A contains a pointer to

the start of a buffer then any jump or call taking that register as an operand can be used to gain control of the flow of execution.
In practice a program may not intentionally contain instructions to jump to a particular register. The traditional solution is to find an unintentional instance of a suitable opcode at a fixed location somewhere within the program memory.

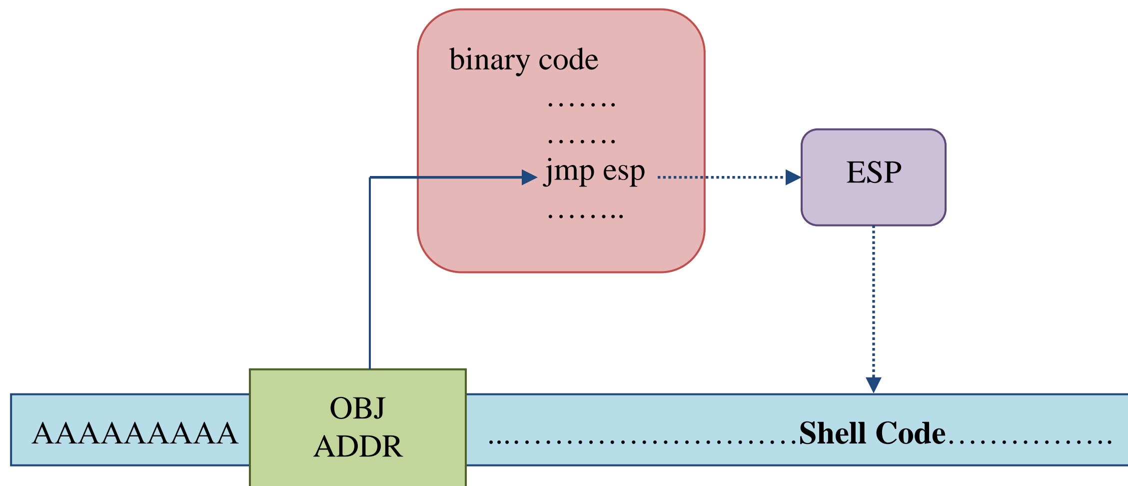


Figure 3.3: Jump-to-Register Technique

Return To libc Technique:

Attacker may jump to a useful/interesting libc function address such as system(), exec() etc. which could help him spawning a shell. Ensure correct arguments are pushed on to the stack. Advantage of this technique is that, the attacker needs not to create a shell code.

3.2.2 Create Attack Code

It is commonly referred to as shell code as well, because generally the intent of code is to spawn a shell. It is a part of large buffer or payload that overflows the program. Some key notes which an attacker has to take care of, while writing attack code are:

- It should be assembled in CPU's native instruction set.
- It should make use of OS specific system calls.
- It should be properly encoded i.e. It should not have null characters '\0' as part of shell code because the unbounded functions like strcpy() etc. just copy till null character.

3.3 Buffer Overflow Detection

Static analysis as well as runtime analysis of the code can protect a programmer from introducing buffer overflow vulnerabilities in the production environment where the siftwarehas to be actually deployed.

3.3.1 Static Analysis

There are tools available like Klockwork and Coverity which contain rules to check for secure coding violations. CiscoProduct Security Group evaluated both the static

analysis tools against violations detailed in CERT's secure coding guidelines and ISO safe C technical doc and observed that Coverity had lower false positive rates, more detailed and intelligent messages, and was able to detect elusive bugs that span multiple functions.

3.3.2 Compile and Runtime Analysis

Compilers options GCC 4.0+ and -D_FORTIFY_SOURCE=1/2, are provided which can perform light weight checks to detect common buffer overflows. These options may infact warn at compile time if hey could detect potential buffer overflow at compile time and replace copy functions (variants of memcpy, strcpy, strat, sprintf, gest etc.) with runtime checking versions which take the length of the destination object. These compiler options enables a programmer to abort the program if overflow is detected at runtime.

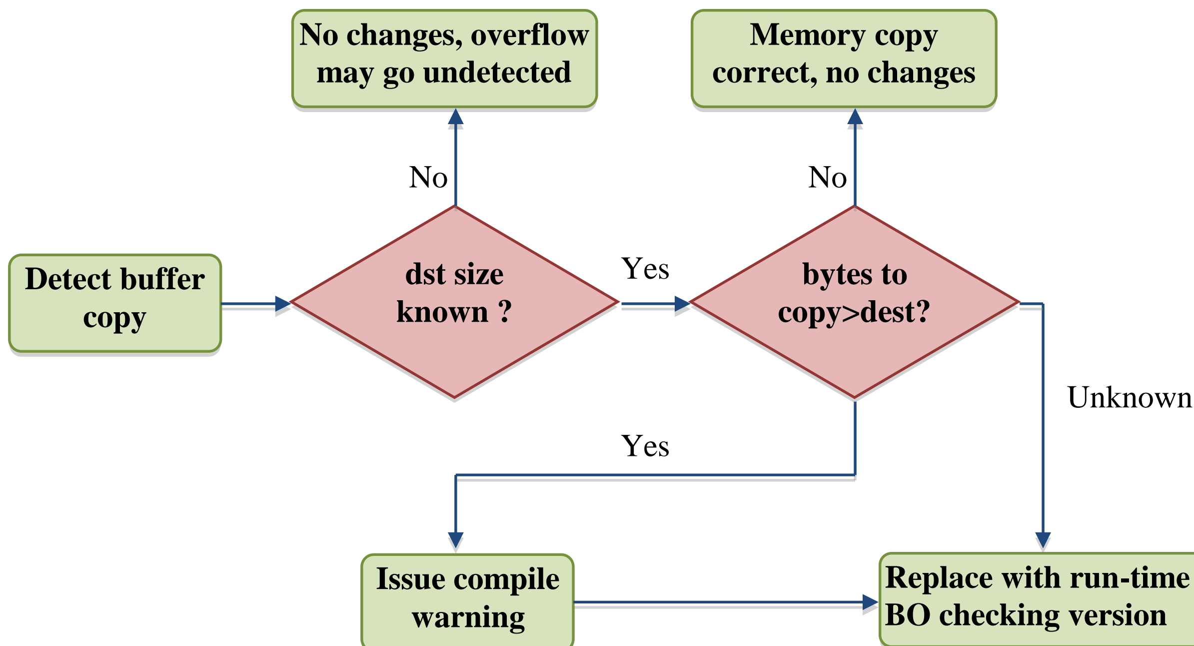


Figure 3.4: Compile and Runtime BO Detection Analysis Flow Diagram

3.4 Buffer Overflow Protection

Of course, it's hard to get programmers to not make common mistakes, and it's often difficult to change programs (and programmers!) to another language. So why not have the underlying system automatically protect against these problems? At the very least, protection against stack-smashing attacks would be a good thing, because stack-smashing attacks are especially easy to do.

In general, changing the underlying system so that it protects against common security problems is an excellent idea, and we'll encounter that theme in later articles too. It turns out there are many defensive measures available, and some of the most popular measures can be grouped into these categories:

- Canary-based defenses. This includes StackGuard (as used by Immunix), ssp/ProPolice (as used by OpenBSD), and Microsoft's /GS option.
- Non-executing stack defenses. This includes Solar Designer's non-exec stack patch (as used by OpenWall) and exec shield (as used by Red Hat/Fedora).
- Other approaches. This includes libsafe (as used by Mandrake) and split-stack approaches.

3.4.1 Canary-Based defenses

Researcher Crispen Cowan created an interesting approach called StackGuard. Stackguard modifies the C compiler (gcc) so that a "canary" value is inserted in front of return addresses. The "canary" acts like a canary in a coal mine: it warns when something has gone wrong. Before any function returns, it checks to make sure that the canary value hasn't changed. If an attacker overwrites the return address (as part of a stack-smashing attack), the canary's value will probably change and the system can stop instead.

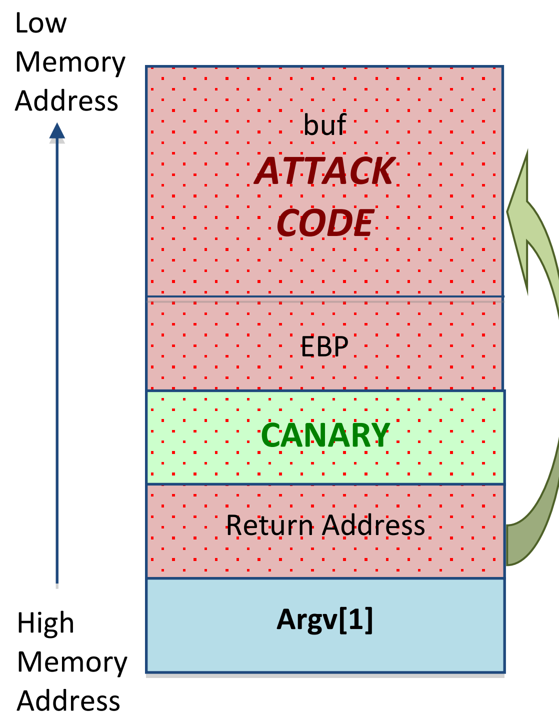


Figure 3.5: Stack Guard (Stack-Smashing Protection)

Limitations:

- It doesn’t protect against overflows on heap.
- It doesn’t protect against certain types of return-to-libc attacks. E.g. a function pointer on the stack was overwritten instead of return address.

3.4.2 Address Space Layout Randomization (ASLR)

Address space randomization defends against attacks that require prior knowledge of addresses by preventing an attacker from being able to easily predict target addresses. For example, attackers trying to execute return-to-libc attacks must locate the code to

be executed; while other attackers trying to execute shell code injected on the stack (return-to-stack) have to first find the stack. In both cases, the related memory addresses are obscured from the attackers; these values have to be guessed, and a mistaken guess is not usually recoverable due to the application crashing. In essence, this technique randomly arranges positions of key data areas like Stack Base, Heap Base, mmap() manages memory (libraries, shared memory etc.) and main executable/data/bss segments in a process's address space.

Limitations:

- NOP Sled - Shellcode injected may have a series of NOP instructions to reduce the accuracy needed to pinpoint exact address.
- Crash and retry attack - Even 16-bit randomization can be broken within minutes.
- Memory fragmentation - This technique may reduce the maximum size of memory mapping an application can create.

Table 3.1: ASLR Implementation Options on Various OS

Operating System	ASLR Option
Linux 2.4 or earlier	PaX kernel patch is available
Linux 2.6.x	Part of kernel distribution
Windows Vista and Server 2008	Kernel support, enable apps using /DYNAMICFLAG linker flag

3.4.3 Executable Space Protection (ESP)

This protection mechanism takes advantage of NX bit on certain processors to enforce non-executable pages or segments.

- Data segments such as heap or stack re marked as non-executable.
- Code segment or shared libraries are marked as read-exeutable.
- Certain restrictions might be applied to API calls such as mmap() and mprotect() that affect memory permissions

This technique aborts the program if attempts were made that violate memory permissions. This mechanism defends against execution of arbitrary code.

Limitations:

- This technique doesn't protect against return-to-libc attacks because in these attacks no need to inject code and they simply rely on jumping to system() call etc.
- Some applications may need to generate code at run-time or have inline assembly, so need to turn off ESP for these kind of applications.

3.4.4 Safe C Lib

Major enhancements to standard C library has been made which ensures correct bounds checking and parameter validation. Question my poop up in reader's mind why

not to go with functions such as strncpy() which take the destination buffer size as an argument? Answer to this question lies below:

- It is difficult to get right (off by 1).
- These functions don't null terminate. Programmer has to himself take care of null terminating the destination buffer.

Sample correct usage:

```
strncpy (dst, src, dstsize-1);
dst[dstsize-1] = '\0';
```

Below is the prototype for strcpy_s(), the safe C lib implementation of strcpy() method:

In above prototype,

- S1 and s2 shall not be null pointers

```
strcpy_s(char *s1, rsize_t s1max, const char *s2)
```

- No overlap copy between s1 and s2
- s1max shall be greater than length of s2
- s1[0] is set to NULL if there is run-time violation detected (if s1 is not null)
- Returns 0 if no run-time violation detected.

Chapter 4

4 Kernel Module Programming

In the following chapter, we will discuss about what is a kernel module and how to write and add code in kernel space. We will present a sample ‘hello world” kernel module to make it easy for the reader to understand.

4.1 What is a Kernel Module?

One of the good features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that we can add functionality to the kernel (and remove functionality as well) while the system is up and running without needing to reboot the system. Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types(or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program. These are special shared objects (like .so files) having extension ‘.ko’ where ‘k’ indicates that it is a kernel object. Kernel modules are nothing but C files which can be loaded and unloaded into the kernel upon demand. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image.

4.2 Writing a Kernel Module

While writing module code, we can’t make use of any APIs, system calls and user-space libraries like libc etc.

Skeleton for Kernel Module Writing:

1. Every kernel module should start with a declaration like:

```
#define MODULE
#define _ _KERNEL_ _
```

   The reason we need to declare “#define MODULE” is that when we gave a .c file to gcc (GNU C Compiler), it assumes it to be user-space code. gcc would link it against ANSI C library, which is not intended for a kernel module as it should not use any user-space libraries. So to tell gcc that it is not an application but a kernel-space module, we make this declaration.

#define _ _*kernel*_ _ is responsible for unlocking all the kernel-space prototypes in the header files which are common between user-space and kernel-space like in *signal.h* there are various prototypes which are accessible to user-space applications and various others which should be accessed only by kernel code.

2. Every kernel module includes many header files depending upon which particular subsystem the kernel code is added to like Process Management, Memory Management, Network Management, Drivers etc. But there are some header files which need to included in each kernel module always.

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>          //Resolves kernel symbol calls
#include <linux/init.h>
```

module.h – is interface header file of kmod subsystem which handles loading and unloading of kernel module. Kmod subsystem is analogous to dynamic loader for shared objects (.so).
version.h – to ensure that a particular module should be loadable in a particular kernel version, this header file helps in stamping the module with a specific kernel version.
kernel.h – responsible for resolving the kernel symbols by providing the list of existing kernel functions.
init.h – is another kmod subsystem header file which provides prototypes for entry and exit methods for a kernel module.

3. Every kernel module has entry and exit functions. On loading a module in kernel-space, entry function is executed which should check if all the required resources, memory etc. for this kernel module are there. So all the acquisitions are done at entry point. Similarly when a module is unloaded, exit function is executed which takes care of releasing all the acquired resources. The functions names could be of our own choice but signatures should match.

```
int init_module (void) {            //user defined entry function
        printk("Module Loaded");
        return 0;
}

void cleanup_module (void) {        //user defined exit function
        printk("Module Unloaded");
}
```

Now these entry and exit functions should be registered with kernel functions like:

```
module_init (init_module);
module_exit (cleanup_module);
```

4. Module Comments - Whenever we need a device driver to specific hardware needs, the chances are more that we get the driver from a third party. Say you acquired the driver from somewhere and now you are facing some problem with it, to whom you would report this problem. So for a module or driver writer, it is always appropriate and encouraged he adds his name, description, email-ID etc. in the module just like comments in the source code. There are some macros to do it, which go along the module in the kernel space. These are known as module comments. These comments are kind of mandatory because for linux kernel, we find hundreds of drivers contributed by number of parties. Below are the macros which help the module author to provide information about himself to help the module user:

   ```
   MODULE_AUTHOR (“Kushal Ahuja”);
   MODULE DESCRIPTION (“ Buffer Overflow Exploit Prevention”);
   ```

   Linux kernel is open-source which gets released with GPL (GNU Public License) licensing policy. Every module must as well declare the open-source licensing policy which it want to be released with. All the licensing policies are defined in *module.h* which even details about what all the licensing policies mean. Kernel macro to declare licensing policy for a kernel module is:

   ```
   MODULE_LICENSE(“GPL”);
   ```

5. Till now we have discussed about the generic layout of a kernel module. The rest is kernel module body which may comprise of functions, variables, data structures etc. But all this code should not use any user-space functions and system calls.

Based on the detailed steps provided above let’s write a “hello world” kernel module:

```
// hello.c
#include <linux/module.h>               // required by all modules
#include <linux/version.h>
#include <linux/kernel.h>               // required by printk()
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kushal Ahuja");
MODULE DESCRIPTION (" Buffer Overflow Exploit Prevention");

// Entry function
static int hello_init(void) {
	printk("Hello world!\n");           // A logging mechanism for kernel
	return 0;   // A non-zero return means hello_init failed; the module can't be loaded.
}

// Exit function
static void hello_cleanup(void) {
	printk("Goodbye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

4.3 Compiling and Building Modules

Regardless of the origin of your kernel, building modules for 2.6.x requires that you have configured and built kernel tree on your system. This requirement is a change from previous kernel versions, where a current set of header files was sufficient. 2.6 modules are linked against object files found in the kernel source tree ; the result is a more robust module loader, but also the requirement that those object files be available. So our first order of business is to come up with a kernel source tree (either from the kernel.org network or distributors' kernel source package), build a new kernel and install it on our system.

After we are done with writing our kernel module, module author has to turn the module source code into an executing subsystem within the kernel. The build procedure for module differs significantly from that used for user-space applications; the kernel is a large , standalone program with detailed and explicit requirements on how its pieces are put together. The build process also differs from how things were done with previous versions of kernel; the new build system is simpler to use and produces more correct results, but it looks every different from what came before. The kernel build system is a complex beast, and we just look at a tiny piece of it. The files in the *Documentation/kbuild* directory in the kernel source are required reading for anybody wanting to understand all that is really going on beneath the surface.

There are some prerequisites that we must get out of our way before we can build kernel modules. The first is to ensure that we have sufficiently current versions of the

compiler, module utilities, and other necessary tools. The file *Documentation/Changes* in the kernel documentation directory always lists the required tool versions; we should consult it before going any further.

Now once we have everything set up, we have to create a makefile for our module. For our “hello world” example shown in above section, a single line in the makefile will suffice:

```
obj-m := hello.o
```

If instead, we have a module called *module.ko* that is generated from two or more sources say file1.c and file2.c, the correct incantation would be:

```
obj-m := module.o
module-objs := file1.o file2.0
```

Readers who are familiar with the *make*, but not with the 2.9 kernel build system, are likely to be wondering how this makefile works. The answer, of course, is that the kernel build system handles the rest. The assignment above (which takes advantage of the extended syntax provided by GNU *make*) states that there is one module to be built from the object *hello.o*. The resulting module is named hello.ko after being built from the object file.

For the above makefile to work, it must be invoked within the context of the larger kernel build system. If our kernel source tree is located in ~/kernel-2.6 directory, the *make* command, required to build module would be:

```
make –C ~/kernel-2.6 M=`pwd` modules
```

The command starts by changing its directory to the one provided with the *–C* option i.e our kernel source directory. There it finds the kernel’s top-level makefile. The *M*= option causes that makefile to move back into our module’s source directory before trying to build the *modules* target. The target in turn refers to the list of modules found in *obj-m* variable, which we have set to hello.o in our example.

To make it easy to build a kernel module, below is the sample makefile, which could simply be run with *make* command.

```
obj-m  := hello.o

KDIR  := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)

all:
            $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
           rm –rf  *.o *.ko
```

4.4 Loading and Unloading Modules

After the module is built, next step is to load it into the kernel. *'insmod'* is the program that loads the data into the kernel, which, in turn performs a function similar to that of ld, in that links any unresolved symbols in the module to the symbol table of the kernel. Interested readers may want to look at how the kernel supports i*nsmod*: it relies on a system call defined in *kernel/module.c*. The function sys_init_module allocates kernel memory to hold a module (this memory is allocated using *vmalloc*); it then copies module text into that memory region, resolves the kernel references in the module via the kernel symbol table, and calls the module's initialization function to get everything going.
The *modprobe* utility is worth a quick mention. *modprobe*, like *insmod*, loads a module into the kernel. It differs in that it will look at the module to be loaded to see whether it references any symbols that are not currently defined in the kernel. If any such references are found, *modprobe* looks for other modules in the current module search path that define the relevant symbols. When *modprobe* finds those modules (which are needed by the module being loaded),it loads them into the kernel as well. If you use *insmod* in this situation instead, the command fails with an "unresolved symbols" message left in the system logfile.

```
insmod hello.ko
modprobe hello.ko
```

To see all the modules inserted into the kernel, we can use:

```
cat  /proc/modules                //file which lists all the modules
lsmod                             // linux command to list all the inserted kernel modules
```

Now, to unload the module, we can use:

```
rmmod hello
```

NOTE: Any messages logged by printk() kernel method don't appear on the console, instead, get logged into a special file. *dmesg* is the command to read those messages from the log file.

Chapter 5

5 Functional Specifications for BOEP

Salient feature of the our kernel module implementation is that it would allow buffer overflow exploit to write beyond the bounds of a program buffer but it would prevent it from impairing our system security.

Below is the general set of features, the module should provide:

1. Locate the system call table, save current state of it and overwrite it with our own function pointer at load-time.
2. Whenever a system call is invoked from user space, the control should be passed to the module function with which we replaced the system call table.
3. The module function should ensure that if the system call originated from the writable region of memory, it should be killed and if not, control should be given back to the actual system call service routine.
4. On unloading, the system call table should be restored to its original state.

Let's represent what we said above, in a form of a flowchart to help us understand the program flow:

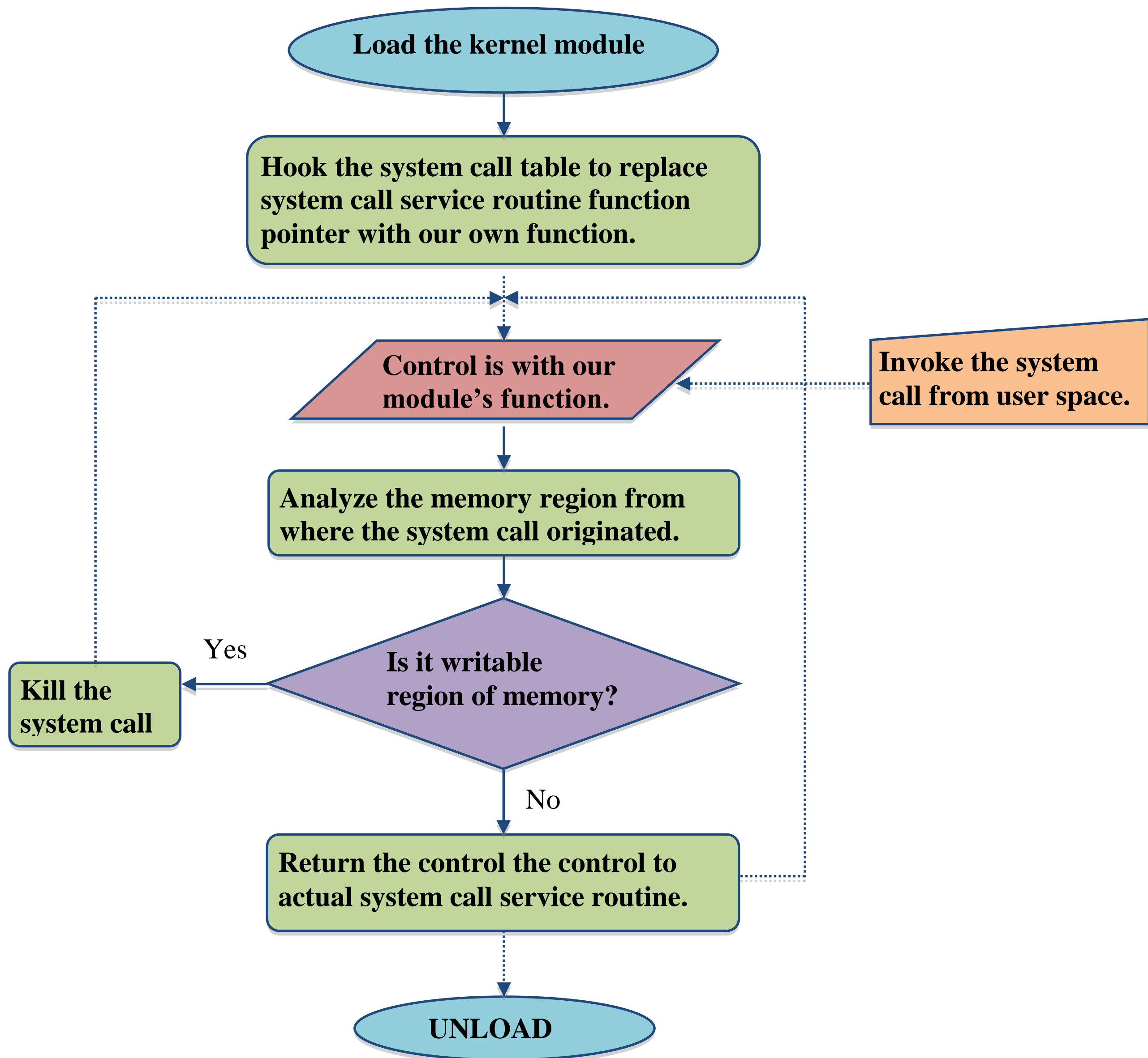


Figure 5.1: BOEP execution flow chart

In the upcoming chapters, we'll detail about, how we can go around implementing these features in our kernel module. Hold on till then.

Chapter 6

6 Hooking System Call Table

This module should generically detect and prevent buffer overflow attacks on Linux by determining if a system call originated from a writable region of memory. If so, it kills the system call. Doing it, requires knowing the system call table address, so that we may hook the table to point to our module function. So, we should be able to locate sys_call_table without the exported symbol. Till linux 2.4 kernel, there was an exported symbol that could give the system call table address like:

```
extern void *sys_call_table[]
```

From linux 2.6 onwards, this functionality was removed for three primary reasons:

1. It made it too easy for a programmer to accidentally trash the entire system with a single module.
2. It made it too easy for a programmer to subvert the entire system, including security etc. with a single module.
3. It was felt that the existing kernel functions where more than adequate for normal module programming.

Before going any further, we need to have some basic knowledge about the what is a system call, system call table because kernel module implementation would require overwriting the function pointers in system call table with our module function.

6.1 System Call

The role of kernel is to collect the requirements from user and to run the application by providing them kernel resources. Kernel abstracts the application from all hardware issues like resource management etc. This communication from user-space application to kernel-space is made possible through system calls. So, system calls are kernel space functions that serve as an interface between kernel and the applications.

A unique number identifies each system call in linux kernel. To see it, go to kernel source directory say *~/kernel-2.6*. In file *include/asm-i386/unistd.h* under kernel source tree, we'll find a list of system calls and corresponding identifiers. The identifiers start with 0 and run through some finite number 293 or so. An example entry for "read" system call in unistd.h is:

```
#define __NR_read      3
```

The macro NR_syscalls contains the total number of system calls for a kernel.

6.2 System Call Table

Now under the kernel source tree, in *arch/i386/kernel/syscall_table.S*, we'll find a core data structure *sys_call_table*. In the runtime, it becomes the table of all the system calls. In this table, address of all the function calls (function pointers) pointing to the implementation of system call, are stored. Functions pointers for a particular system call reside at an offset equivalent to the identifier for system call specified in unistd.h. An example entry for "read" system call function pointer in syscall_table.S is:

```
.long sys_read
```

System call implementation is kept under different directories depending on the nature of task, the system call is supposed to perform e.g. system call that belong to memory, go into memory management module under kernel source tree *~/kernel-2.6/mm*.

Table 6.1: System Call Implementation Directory for Various Kernel Services

Kernel Services	System Call Implementation Directory
Memory Management	~/kernel-2.6/mm
Network Management	~/kernel-2.6/net
Device Management	~/kernel-2.6/drivers
Generic Services	~/kernel-2.6/kernel

The function prototype for sys_read() under *~/kernel-2.6/fs/read_write.c* appears like:

```
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
```

The system call resides in the kernel space, so it should not use any API/library calls, and so should our module function because it is going to be executed on behalf of user-space system call.

6.3 X86 Exceptions

To overwrite the system call table, we are going to make use of x86 programmed exception vector *int 0x80.* Let us first discuss about the different kind of software exceptions available in x86. Exceptions, are caused either by programming errors or by anomalous conditions that must be handled by the kernel. In the first case, the kernel handles the exception by delivering to the current process one of the signals familiar to every Unix programmer. In the second case, the kernel performs all the steps needed to recover from the anomalous condition, such as a Page Fault or a request via an assembly language instruction such as ***int*** for a kernel service.

Table 6.2: Various x86 Exceptions

Exceptions	Description
Processor-detected exceptions	Generated when the CPU detects an anomalous condition while executing an instruction. These are further divided into three groups, depending on the value of the *eip* register that is saved on the Kernel

	Mode stack when the CPU control unit raises the exception.
Faults	Can generally be corrected; once corrected, the program is allowed to restart with no loss of continuity. The saved value of *eip* is the address of the instruction that caused the fault, and hence that instruction can be resumed when the exception handler terminates.
Traps	Reported immediately following the execution of the trapping instruction; after the kernel returns control to the program, it is allowed to continue its execution with no loss of continuity. The saved value of eip is the address of the instruction that should be executed after the one that caused the trap. A trap is triggered only when there is no need to reexecute the instruction that terminated. The main use of traps is for debugging purposes. The role of the interrupt signal in this case is to notify the debugger that a specific instruction has been executed (for instance, a breakpoint has been reached within a program). Once the user has examined the data provided by the debugger, she may ask that execution of the debugged program resume, starting from the next instruction.
Aborts	A serious error occurred; the control unit is in trouble, and it may be unable to store in the eip register the precise location of the instruction causing the exception. Aborts are used to report severe errors, such as hardware failures and invalid or inconsistent values in system tables. The interrupt signal sent by the control unit is an emergency signal used to switch control to the corresponding abort exception handler. This handler has no choice but to force the affected process to terminate.
Programmed Exception	Occur at the request of the programmer. They are triggered by *int* or *int3* instructions; the into (check for overflow) and bound (check on address bound) instructions also give rise to a programmed exception when the condition they are checking is not true. Programmed exceptions are handled by the control unit as traps; they are often called *software interrupts* . Such exceptions have two common uses: to implement system calls and to notify a debugger of a specific event.

In the above table, as explained int 0x80, is a mechanism to make a system call. In other words, when a system call is made from user-space, int 0x80 serves as a way to switch from user-space to kernel-space.

6.4 System Call Handler and Service Routine

As discussed above, When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function. The result is a jump to an assembly language function called the *system call handler*. Because the kernel implements many different system calls, the User Mode process must pass a parameter called the *system call number* to identify the required system call; the eax register is used by Linux for this purpose. General execution flow while invoking a system call is:

- The system call number as seen from *unistd.h,* is loaded into eax.
- All the parameters for system call are pushed into CPU registers. But to pass the parameters in registers, two conditions must be satisfied:

1. The length of each parameter cannot exceed the length of a register (32 bits for 32-bit architecture).
2. The number of parameters must not exceed six, besides the system call number passed in eax, because 80 x 86 processors have a very limited number of registers.

However, system calls that require more than six parameters exist. In such cases, a single register is used to point to a memory area in the process address space that contains the parameter values. Of course, programmers do not have to care about this workaround. As with every C function call, parameters are automatically saved on the stack when the wrapper routine is invoked. This routine will find the appropriate way to pass the parameters to the kernel.

- Now, as the signature is ready, int 0x80 is invoked to switch from user-space to kernel-space.
- In kernel-space, eax is read back to see the service routine to be executed to serve the user-space system call.

The system call handler, which has a structure similar to that of the other exception handlers, performs the following operations:

- Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language).
- Handles the system call by invoking a corresponding C function called the *system call service routine*.
- Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language).

The name of the service routine associated with the xyz( ) system call is usually sys_xyz(); there are, however, a few exceptions to this rule. Figure below explains the execution described above.

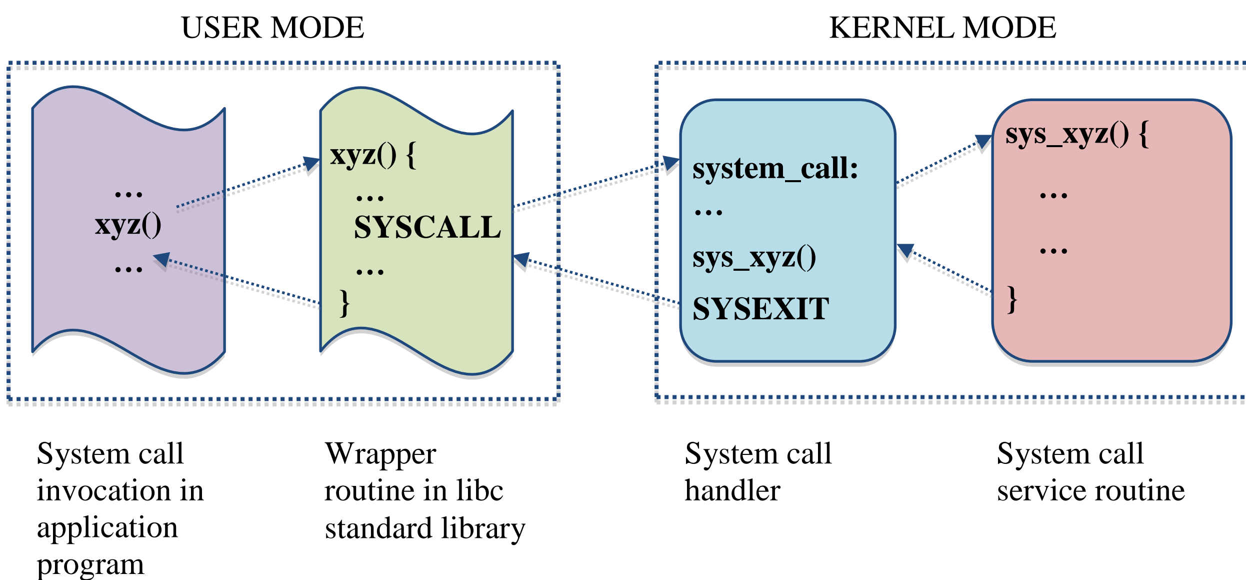


Figure 6.1: Invoking a System Call

The terms "SYSCALL" and "SYSEXIT" are placeholders for the actual assembly language instructions that switch the CPU, respectively, from User Mode to Kernel Mode and from Kernel Mode to User Mode. SYSCALL in our case, corresponds to “int 0x80” assembly instruction. To reiterate, to associate each system call number with its corresponding service routine, the kernel uses a *system call dispatch table*, which is stored in the sys_call_table array and has NR_syscalls entries (294 in the Linux 2.6.14 kernel). The nth entry contains the service routine address of the system call having number *n*.

6.5 Interrupt Descriptor Table

A system table called *Interrupt Descriptor Table* (IDT) associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler. The IDT must be properly initialized before the kernel enables interrupts. Each entry corresponds to an interrupt or an exception vector and consists of an 8-byte descriptor. Thus, a maximum of 256 x 8 = 2048 bytes are required to store the IDT. The idtr CPU register allows the IDT to be located anywhere in memory: it specifies both the IDT base physical address and its limit (maximum length). It must be initialized before enabling interrupts by using the *lidt* assembly language instruction. It can be read back using *sidt* assembly instruction. The IDT may include three types of descriptors; Task Gate Descriptor, Interrupt Gate Descriptor and Trap Gate Descriptor. We are interested in knowing about Trap Gate Descriptor and here we go.

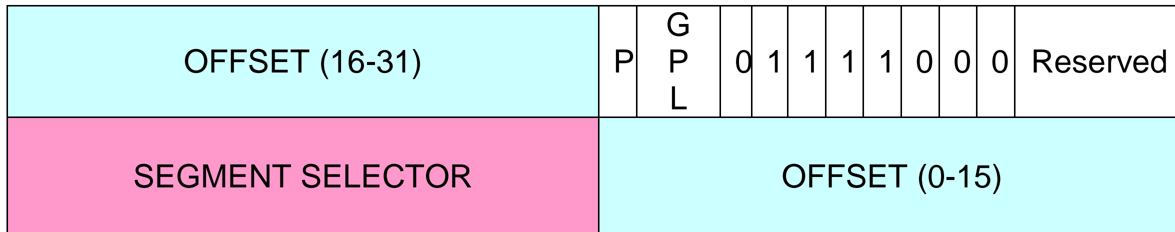


Figure 6.2: Trap Gate Descriptor in IDT

Linux uses a slightly different breakdown and terminology from Intel when classifying the interrupt descriptors included in the Interrupt Descriptor Table:

Table 6.3: Linux IDT Descriptors

Interrupt Descriptor	Architecture Dependent Function for IDT descriptor	Description
Interrupt Gate	set_intr_gate(n, addr)	Inserts an interrupt gate in the *n*th IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to addr, which is the address of the interrupt handler. The DPL field is set to 0.
System Gate	set_system_gate(n, addr)	Inserts a trap gate in the *n* th IDT entry. The Segment Selector inside the gate is set to the kernel code's Segment Selector. The Offset field is set to addr, which is the address of the exception handler. The DPL field

		is set to 3.
System Interrupt Gate	set_system_intr_gate(n, addr)	Inserts an interrupt gate in the *n* th IDT entry. The Segment Selector inside the gate is set to the kernel code's xxx`Segment Selector. The Offset field is set to addr, which is the address of the exception handler. The DPL field is set to 3.
Trap Gate	set_trap_gate(n, addr)	Similar to the previous function, except the DPL field is set to 0.
Task Gate	set_task_gate(n, addr)	Inserts a task gate in the *n* th IDT entry. The Segment Selector inside the gate stores the index in the GDT of the TSS containing the function to be activated. The Offset field is set to 0, while the DPL field is set to 3.

As per linux breakdown, we are interested in System Gate which actually inserts a hardware level Trap Gate at system boot up. To take advantage of exceptions, the IDT must be properly initialized with an exception handler function for each recognized exception. As it is clear in above table, *set_system_gate(128, &system_call);* establishes a exception handler to handle system calls issued from user space.

6.6 Issuing a System Call via *int 0x80* Instruction

As discussed in above sections, The vector 128 in hexadecimal, 0x80 is associated with the kernel entry point. The *trap_init( )* function, invoked during kernel initialization, sets up the Interrupt Descriptor Table entry corresponding to vector 128 as follows:

```
set_system_gate(0x80, &system_call);
```

Above call loads following values in the IDT, system gate descriptor:

Table 6 .4: IDT System Gate Descriptor after set_system_gate(0x80, &system_call)

Descriptor Field	Value Loaded
Segment Selector	The _ _KERNEL_CS Segment Selector of the kernel code segment.
Offset	The pointer to the system_call( ) system call handler.
Type	Set to 15. Indicates that the exception is a Trap and that the corresponding handler does not disable maskable interrupts.
DPL (Descriptor Privilege Level)	Set to 3. This allows processes in User Mode to invoke the exception handler

Therefore, when a User Mode process issues an *int $0x80* instruction, the CPU switches into Kernel Mode and starts executing instructions from the *system_call* address.

The *system_call()* function checks further for the validity of given system call number by comparing it to *NR_syscalls*. If it is larger than or equal to *NR_syscalls*, the function returns –*ENOSYS*. Otherwise, the specified system call is invoked:

```
call *sys_call_table(, %eax, 4)
```

Because, each element in the system call table is 32-bits (4 bytes) long, the kernel multiplies the given system call number by 4 to arrive at its location in the system call table.
With all above information, we should be able to hook on the system call table and direct the control to our own kernel module function whenever a system call is issued from user mode.

Chapter 7

7 Protection System Call Handler

After hooking the system call table, core stuff left is to know, whether the memory region from where the system call is launched is writable or not. Because generally processor doesn't allow to execute from writable region of memory, we'll kill the system, if it happens so.

To carry forward this task, the most important kernel data structure is *struct task_struct*, defined by *<linux./sched.h>* (Appendix A). Although kernel modules don't execute sequentially as applications do, most actions performed by the kernel are done on behalf of specific process. Kernel code can refer to the current process by accessing the global item *current*, defined in *<asm/current.h>*, which is an instance of data structure *struct task_struct*. The *current* pointer refers to the process that is currently executing. So during the execution of system call, such as open or read, the current process is the one that invoked the call. Kernel code can use process-specific information by using *current*, if it needs to do so. For example, the following statement prints the process ID and the command name of the current process by accessing certain fields in *struct task_struct*:

```
printk("The process is \"%s\" (pid %i)\n", current->comm, current->pid);
```

Similarly, from the *current* pointer, we can access the eip, using *KSTK_EIP()* kernel macro and can verify which virtual memory page from the address space of process, it belongs to. *find_vma()* (defined in *<linux/mm/mmap.c>* is a function which can analyze to which virtual memory page, an address belongs and returns a pointer to *struct vm_area_struct*, defined in *<linux/mm.h>*. Below given is the signature for *vma_find()*:

```
struct vm_area_struct * find_vma (struct task_struct * task, unsigned long addr)
```

vma_find() sets the attributes in *struct vm_area_struct* to appropriate values, which talks about the attributes of virtual memory area (VMA) of process's virtual address space, wherein the given address falls. One among the attributes is, *vm_flags*, indicating the permissions about the VMA . Some values associated to *vm_flags* are:

```
#define VM_READ          0x0001
#define VM_WRITE         0x0002
#define VM_EXEC          0x0004
#define VM_STACK_FLAGS   0x0177
```

Below is the vm_area_struct, directly taken from <linux/mm.h>:

```
/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task.  A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */
struct vm_area_struct {
    struct mm_struct * vm_mm;   /* The address space we belong to. */
    unsigned long vm_start;     /* Our start address within vm_mm. */
    unsigned long vm_end;       /* The first byte after our end address
                   within vm_mm. */

    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;      /* Access permissions of this VMA. */
    unsigned long vm_flags;     /* Flags, listed below. */

    struct rb_node vm_rb;

    /*
     * For areas with an address space and backing store,
     * linkage into the address_space->i_mmap prio tree, or
     * linkage to the list of like vmas hanging off its node, or
     * linkage of vma in the address_space->i_mmap_nonlinear list.
     */
    union {
        struct {
            struct list_head list;
            void *parent;   /* aligns with prio_tree_node parent */
            struct vm_area_struct *head;
        } vm_set;

        struct raw_prio_tree_node prio_tree_node;
    }
```

Provided all the above information, we should be able to kill the system call, if it seems to be originating from the writable region of memory.

Chapter 8

8 Summary, Conclusion and Future Work

8.1 Summary

As part of this report, we learnt about what are buffer overflow vulnerabilities, how they are exploited, and how we can prevent them from being exploited. We dived into the linux kernel and learnt its inner workings. Then, we used kernel module programming to prevent the buffer overflow attacks from hijacking our systems.

8.2 Conclusion

Buffer overflow vulnerabilities can be minimized by following secure coding practices but still it's difficult to completely eradicate them because to err is human. Moreover, it's a tedious job to sit and scrutinize all the existing applications for possible buffer overflow vulnerabilities. This is where the Buffer Overflow Exploit Prevention module becomes useful; you can simply load it at run-time in your kernel and just forget about your system security, which can otherwise be impaired by an attacker.

8.3 Future Work

The future work in this direction can involve:

1. Implementing an *ioctl* interface to interact with the kernel module from user space to pass it on a set of set of system calls and corresponding actions like kill or ignore.
2. Expanded reporting to the user space.
3. Finding a way to deal with the "system calls" that come from kernel.

Bibliography

[1] Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman. Linux Device Drivers. Sebastopol, USA: O'REILLY, Third Edition, 2005

[2] Robert Love, Linux Kernel Development. Indianapolis, USA: Pearson Education, Second Edition, 2005

[3] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel. Sebastopol, USA: O'Reilly, Third Edition, 2005

[4] John Qian. "How to Kill Buffer Overflows – What is it". Product Security Workshop. <http://wwwin-enged.cisco.com/etools/videolibrary_public/cgi-bin/>

[5] David A. Wheeler. "Countering Buffer Overflows". Secure Programmer. <**http://www.ibm.com/developerworks/linux/library/l-sp4.html**>

[6] Common Weakness Enumeration. <**http://cwe.mitre.org/**>

[7] Analysis of Buffer Overflow Attacks. <**http://www.windowsecurity.com/articles/Analysis_of_Buffer_Overflow_Attacks.html**>

[8] Istvan Simon. "A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks". <**http://www.mcs.csuhayward.edu/~simon/security/boflo.html**>

Appendix A

The *task_struct* is a huge data structure, at around 1.7 Kbytes on a 32-bit machine. The size, however, is quite small considering that the structure contains all the information that the kernel has and needs about a process. The process descriptor contains the data that describes the executing program's open files, the process's address space, pending signals, the process's state and much more.

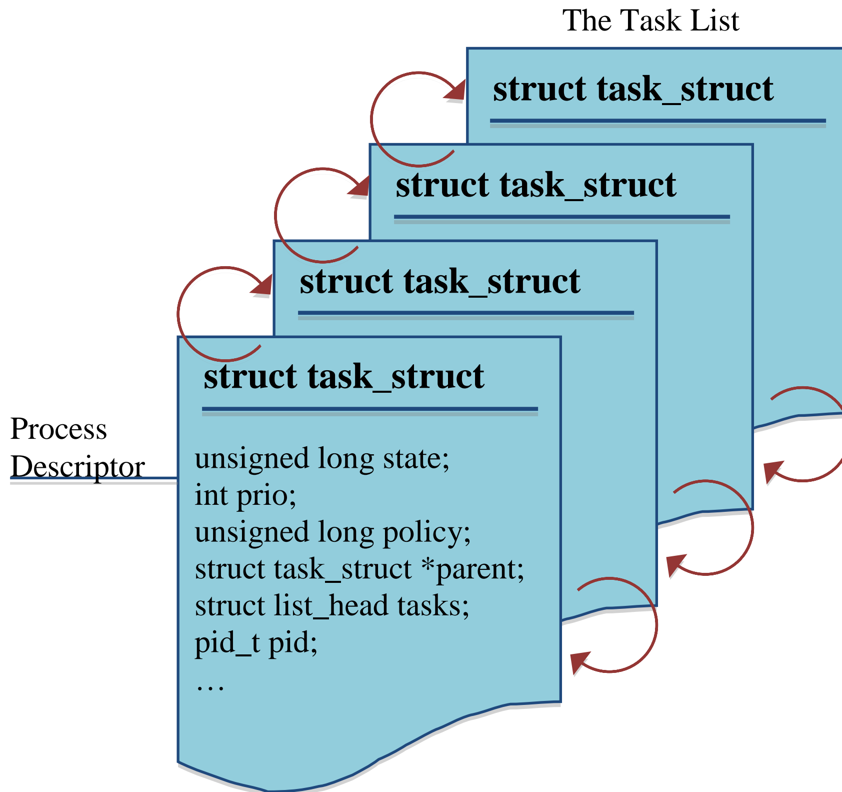


Figure A.1: The Process Descriptor and Task List

```
<include/linux/sched.h>

struct task_struct {
   volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
   struct thread_info *thread_info;
   atomic_t usage;
   unsigned long flags;    /* per process flags, defined below */
   unsigned long ptrace;

   int lock_depth;     /* BKL lock depth */

#if defined(CONFIG_SMP) && defined(__ARCH_WANT_UNLOCKED_CTXSW)
   int oncpu;
#endif
   int prio, static_prio;
   struct list_head run_list;
   prio_array_t *array;
```

```
    unsigned short ioprio;

    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    unsigned long long sched_time; /* sched_clock time spent running */
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;

#ifdef CONFIG_SCHEDSTATS
    struct sched_info sched_info;
#endif

    struct list_head tasks;
    /*
     * ptrace_list/ptrace_children forms the list of my children
     * that were stolen by a ptracer.
     */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

/* task state */
    struct linux_binfmt *binfmt;
    long exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */
    /* ??? */
    unsigned long personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->parent->pid)
     */
    struct task_struct *real_parent; /* real parent process (when being debugged) */
    struct task_struct *parent; /* parent process */
    /*
     * children/sibling forms the list of my children plus the
     * tasks I'm ptracing.
     */
    struct list_head children;  /* list of my children */
    struct list_head sibling;   /* linkage in my parent's children list */
    struct task_struct *group_leader;   /* threadgroup leader */

    /* PID/PID hash table linkage. */
    struct pid pids[PIDTYPE_MAX];

    struct completion *vfork_done;      /* for vfork() */
    int __user *set_child_tid;      /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */

    unsigned long rt_priority;
```

```
   cputime_t utime, stime;
   unsigned long nvcsw, nivcsw; /* context switch counts */
   struct timespec start_time;
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
   unsigned long min_flt, maj_flt;

   cputime_t it_prof_expires, it_virt_expires;
   unsigned long long it_sched_expires;
   struct list_head cpu_timers[3];

/* process credentials */
   uid_t uid,euid,suid,fsuid;
   gid_t gid,egid,sgid,fsgid;
   struct group_info *group_info;
   kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
   unsigned keep_capabilities:1;
   struct user_struct *user;
#ifdef CONFIG_KEYS
   struct key *thread_keyring; /* keyring private to this thread */
   unsigned char jit_keyring;  /* default keyring to attach requested keys to */
#endif
   int oomkilladj; /* OOM kill score adjustment (bit shift). */
   char comm[TASK_COMM_LEN]; /* executable name excluding path
                 - access with [gs]et_task_comm (which lock
                   it with task_lock())
                 - initialized normally by flush_old_exec */
/* file system info */
   int link_count, total_link_count;
/* ipc stuff */
   struct sysv_sem sysvsem;
/* CPU-specific state of this task */
   struct thread_struct thread;
/* filesystem information */
   struct fs_struct *fs;
/* open file information */
   struct files_struct *files;
/* namespace */
   struct namespace *namespace;
/* signal handlers */
   struct signal_struct *signal;
   struct sighand_struct *sighand;

   sigset_t blocked, real_blocked;
   struct sigpending pending;

   unsigned long sas_ss_sp;
   size_t sas_ss_size;
   int (*notifier)(void *priv);
   void *notifier_data;
   sigset_t *notifier_mask;

   void *security;
   struct audit_context *audit_context;
   seccomp_t seccomp;

/* Thread group tracking */
   u32 parent_exec_id;
   u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
```

```
    spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
    spinlock_t proc_lock;

/* journalling filesystem info */
    void *journal_info;

/* VM state */
    struct reclaim_state *reclaim_state;

    struct dentry *proc_dentry;
    struct backing_dev_info *backing_dev_info;

    struct io_context *io_context;

    unsigned long ptrace_message;
    siginfo_t *last_siginfo; /* For ptrace use.  */
/*
 * current io wait handle: wait queue entry to use for io waits
 * If this thread is processing aio, this points at the waitqueue
 * inside the currently handled kiocb. It may be NULL (i.e. default
 * to a stack based synchronous wait) if its doing sync IO.
 */
    wait_queue_t *io_wait;
/* i/o counters(bytes read/written, #syscalls */
    u64 rchar, wchar, syscr, syscw;
#if defined(CONFIG_BSD_PROCESS_ACCT)
    u64 acct_rss_mem1;  /* accumulated rss usage */
    u64 acct_vm_mem1;   /* accumulated virtual memory usage */
    clock_t acct_stimexpd;  /* clock_t-converted stime since last update */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *mempolicy;
    short il_next;
#endif
#ifdef CONFIG_CPUSETS
    struct cpuset *cpuset;
    nodemask_t mems_allowed;
    int cpuset_mems_generation;
#endif
    atomic_t fs_excl;   /* holding fs exclusive resources */
};
```