
Buffer Overflows

Internet Security [1] VU

Matthias Neugschwandtner

Christian Platzer, Gilbert Wondracek, Edgar Weippl

inetsec@iseclab.org

News and Announcements

*Int. Secure Systems Lab
Vienna University of Technology*

- Challenge 4 starts at 18:00 today

Outline

Int. Secure Systems Lab
Vienna University of Technology

- Overview: Buffer Overflows
- Memory Management & x86 Process Layout
- Stack / Frames / x86 Function Calls
- Buffer Overflow Exploitation
- Shell Code 101
- Defenses

Buffer Overflows

- One of the most widely used attack methods of compromising a machine
 - if successful, allows execution of *arbitrary code* in the context of the *exploited program*
- Goal / Steps
 - 1) inject instructions into memory of vulnerable program
 - 2) exploit program vulnerability to change control flow (flow of execution), and
 - 3) execute (arbitrary) injected code

Buffer Overflows

- Advantages
 - very effective
 - attack code runs with privileges of exploited process
 - can be exploited locally and remotely
 - interesting for network services
- Disadvantages
 - architecture dependent
 - directly inject assembler code
 - operating system dependent
 - use system call functions
 - some guess work involved (correct addressing)

Memory Management

- Many modern languages (Java, python, etc.) provide *automatic buffer size checks* when accessing memory

```
try {
    byte data[] = new byte[16];
    for (int i=0; true; i++) {
        byte b = readbyte();
        if (b == 0) break;
        data[i] = b;
    }
} catch (Exception e) {
    ...
}
```

Memory Management

- Many modern languages (Java, python, etc.) provide *automatic buffer size checks* when accessing memory

```
try {  
    byte data[] = new byte[16];  
    for (int i=0; true; i++) {  
        byte b = readbyte();  
        if (b == 0) break;  
        data[i] = b;  
    }  
} catch (Exception e) {  
    ...  
}
```

Statically sized buffer.
If user provides more
than 16 bytes of input,
the buffer will overflow!

- Here, the language will catch the overrun and throw an exception (that can be handled by the program)
 - if program ignores the exception, execution terminates (i.e., without causing any harm)

Memory Management

Int. Secure Systems Lab
Vienna University of Technology

- Some languages (mainly C/C++) do *not* provide such checks
 - program must make sure that only the allocated number of bytes are written to the buffer
 - if it fails to do so, adjacent buffers in memory will be overwritten
 - overwritten buffers may contain sensitive information (such as passwords, memory management information, or control flow data)

Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
 - program must make sure that only the allocated number of bytes are written to the buffer

```
– char password[16];  
– char data[16];  
–  
– snprintf(password,  
           sizeof(password)-1,  
           "secret");  
  
for (int i=0; 1; i++) {  
    char b = readbyte();  
    if (b == 0) break;  
    data[i] = b;  
}
```

compiler will “reserve” 16 bytes of memory on the *stack* for the password buffer

information, or control

Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
 - program must make sure that only the allocated number of bytes are written to the buffer

```
– char password[16];  
– char data[16];  
– snprintf(password,  
           sizeof(password)-1,  
           "secret");  
  
for (int i=0; 1; i++) {  
    char b = readbyte();  
    if (b == 0) break;  
    data[i] = b;  
}
```

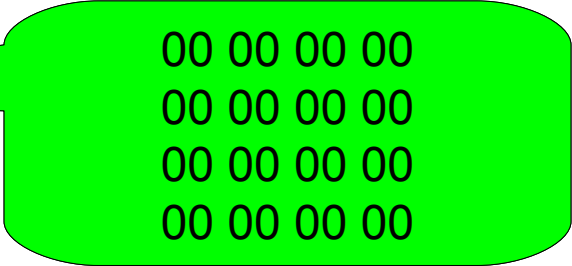
compiler will "reserve" 16 bytes of memory on the stack for the password buffer

same for the data buffer

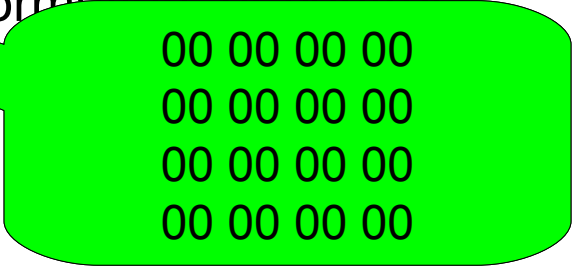
Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
 - program must make sure that only the allocated number of bytes are written to the buffer

```
– char password[16];  
– char data[16];  
– snprintf(password,  
           sizeof(password)-1,  
           "secret");  
  
for (int i=0; 1; i++) {  
    char b = readbyte();  
    if (b == 0) break;  
    data[i] = b;  
}
```



00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00



00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00

Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
 - program must make sure that only the allocated number of bytes are written to the buffer

```
– char password[16];  
– char data[16];  
– snprintf(password,  
           sizeof(password)-1,  
           "secret");  
  
for (int i=0; 1; i++) {  
    char b = readbyte();  
    if (b == 0) break;  
    data[i] = b;  
}
```

```
00 00 00 00  
00 00 00 00  
'e' 't' '\0' 00  
's' 'e' 'c' 'r'
```

```
00 00 00 00  
00 00 00 00  
00 00 00 00  
00 00 00 00
```

Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
 - program must make sure that only the allocated number of bytes are written to the buffer

```
– char password[16];  
  char data[16];  
  
– snprintf(password,  
           sizeof(password)-1,  
           "secret");  
  
for (int i=0; 1; i++) {  
  char b = readbyte();  
  if (b == 0) break;  
  data[i] = b;  
}
```

00 00 00 00
00 00 00 00
'e' 't' '\0' 00
's' 'e' 'c' 'r'

00 00 00 00
41 41 41 00
41 41 41 41
41 41 41 41

"AAAAAAAAAAAA\0"

Memory Management

- Some languages (mainly C/C++) do *not* provide such checks
 - program must make sure that only the allocated number of bytes are written to the buffer

```
– char password[16];  
– char data[16];  
– snprintf(password,  
           sizeof(password)-1,  
           "secret");  
  
for (int i=0; 1; i++) {  
    char b = readbyte();  
    if (b == 0) break;  
    data[i] = b;  
}
```

00 00 00 00
00 00 00 00
'e' 't' '\0' 00
41 41 41 41

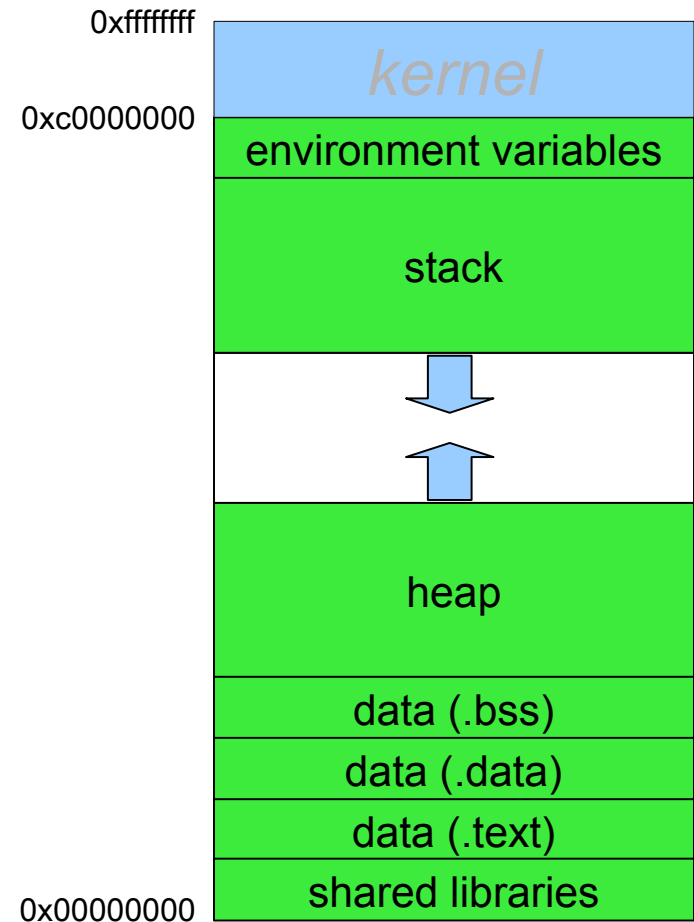
41 41 41 41
41 41 41 41
41 41 41 41
41 41 41 41

"AAAAAAAAAAAAAA
AAAAAAAA\0"

Memory Layout

Int. Secure Systems Lab
Vienna University of Technology

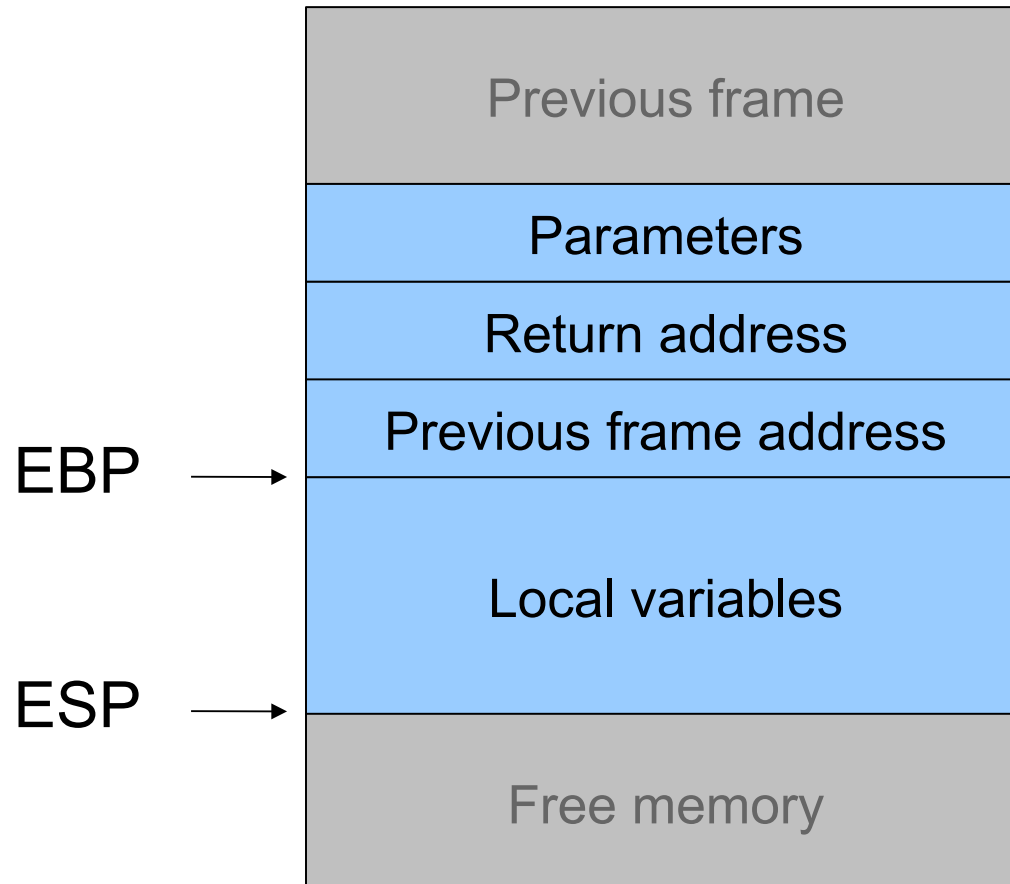
- Stack segment
 - local variables
 - procedure activation records
- Data segment
 - global uninitialized variables (.bss)
 - global initialized variables (.data)
 - dynamic variables (heap)
- Code (text) segment
 - program instructions
 - usually read-only
- In Linux, under the proc filesystem
\$ cat /proc/<pid>/maps



Stack

- Usually grows towards smaller memory addresses
 - Intel, Motorola, SPARC, MIPS
- Special processor register points to top of stack
 - `stack pointer - SP`
 - points to *last stack element*
- Composed of *frames*
 - upon function *call*, a new frame is pushed on top of stack
 - used to conveniently reference *local variables*
 - upon function *return*, frame is discarded, last frame on stack is restored
 - address of current frame stored in processor register
 - `frame/base pointer - FP`

Frame



Stack – Function Call

```
int authenticate(char *name)
{
    // check for 'inetsecXXX'
    if (strncmp(name,"inetsec",7) || name[7]<'0' || name[7]>'9'...)
    {
        char error_msg[32];
        sprintf(error_msg, "Invalid user '%s'\n", name);
        fprintf(stderr, error_msg);
        return 1;
    }
    printf("authentication for %s succeeded\n", name);
    return 0;
}

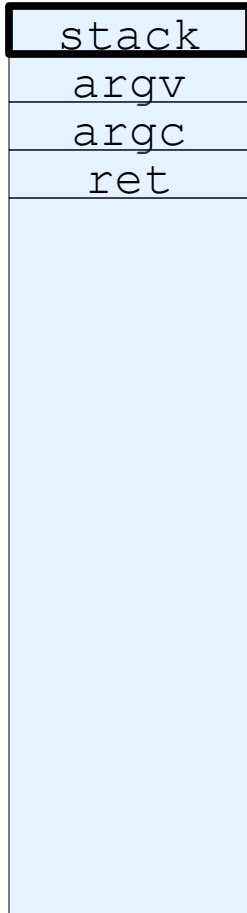
int main(int argc, char **argv)
{
    if (argc != 2) return 1;
    if (authenticate(argv[1])) return 2;
    ...
}
```

Stack – Function Call

```
int authenticate(char *name)
{
    // check for 'inetsecXXX'
    if (strncmp(name,"inetsec",7) || name[7]<'0' || name[7]>'9'...)
    {
        char error_msg[32];
        sprintf(error_msg, "Invalid user '%s'\n", name);
        fprintf(stderr, error_msg);
        return 1;
    }
    printf("authentication for %s succeeded\n", name);
    return 0;
}

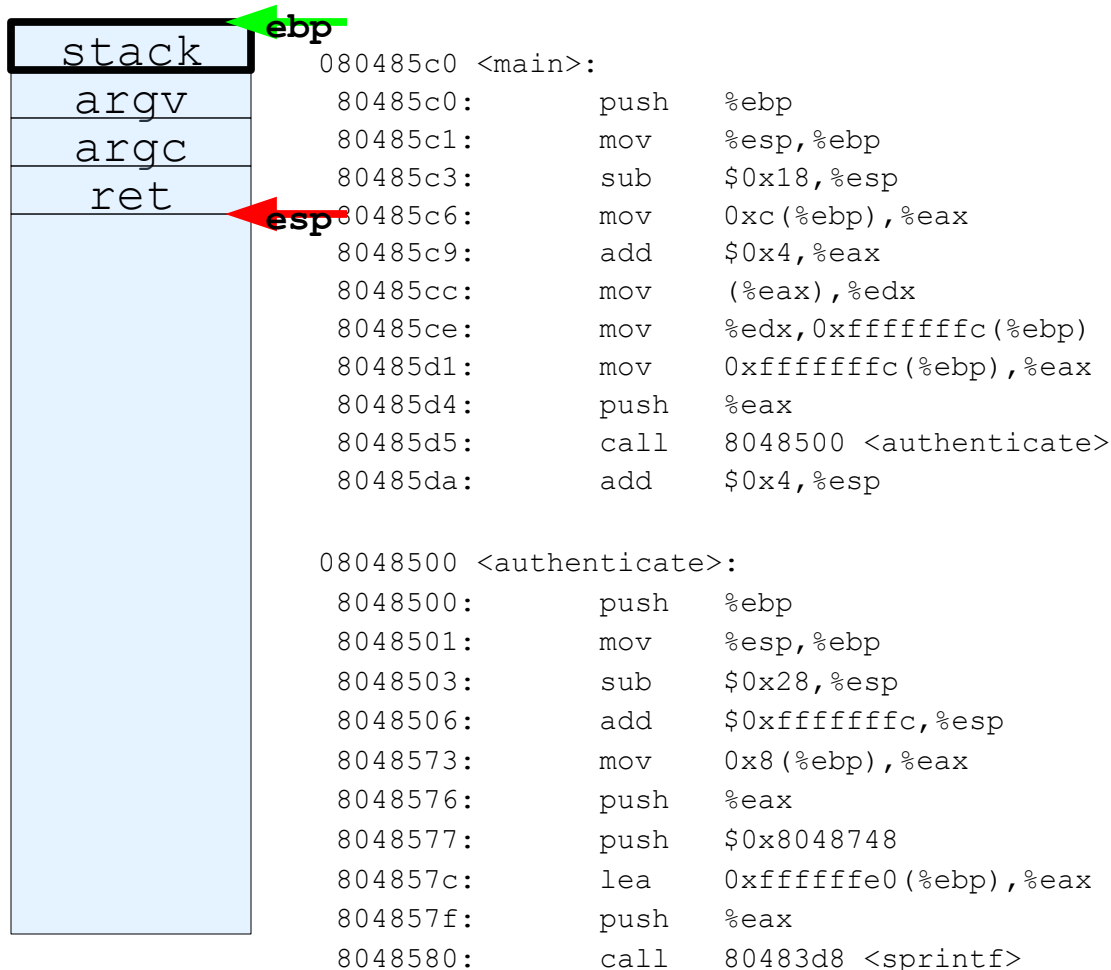
int main(int argc, char **argv)
{
    if (argc != 2) return 1;
    if (authenticate(argv[1])) return 2;
    ...
}
```

Stack – Function Call

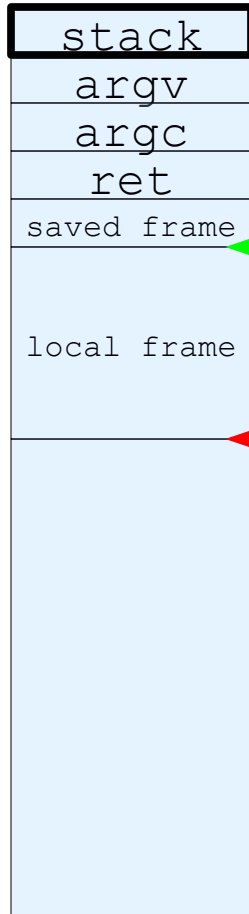


```
080485c0 <main>:  
80485c0:    push    %ebp  
80485c1:    mov     %esp,%ebp  
80485c3:    sub     $0x18,%esp  
80485c6:    mov     0xc(%ebp),%eax  
80485c9:    add     $0x4,%eax  
80485cc:    mov     (%eax),%edx  
80485ce:    mov     %edx,0xffffffff(%ebp)  
80485d1:    mov     0xffffffff(%ebp),%eax  
80485d4:    push   %eax  
80485d5:    call   8048500 <authenticate>  
80485da:    add     $0x4,%esp
```


Stack – Function Call



Stack – Function Call

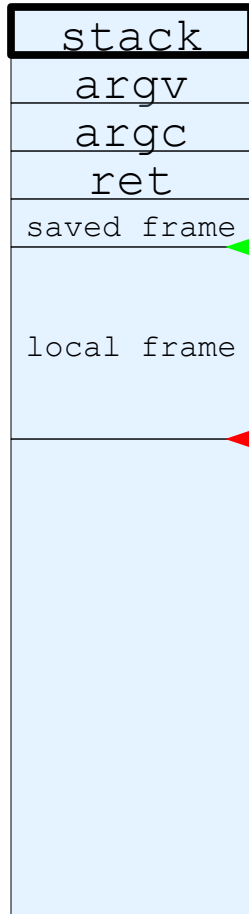


```
080485c0 <main>:
80485c0:   push   %ebp
80485c1:   mov    %esp,%ebp
80485c3:   sub    $0x18,%esp
80485c6:   mov    0xc(%ebp),%eax
80485c9:   add    $0x4,%eax
80485cc:   mov    (%eax),%edx
80485ce:   mov    %edx,0xffffffff(%ebp)
80485d1:   mov    0xffffffff(%ebp),%eax
80485d4:   push   %eax
80485d5:   call  8048500 <authenticate>
80485da:   add    $0x4,%esp

08048500 <authenticate>:
8048500:   push   %ebp
8048501:   mov    %esp,%ebp
8048503:   sub    $0x28,%esp
8048506:   add    $0xffffffff,%esp
8048573:   mov    0x8(%ebp),%eax
8048576:   push   %eax
8048577:   push   $0x8048748
804857c:   lea   0xffffffe0(%ebp),%eax
804857f:   push   %eax
8048580:   call  80483d8 <sprintf>
```

save frame and allocate new one

Stack – Function Call



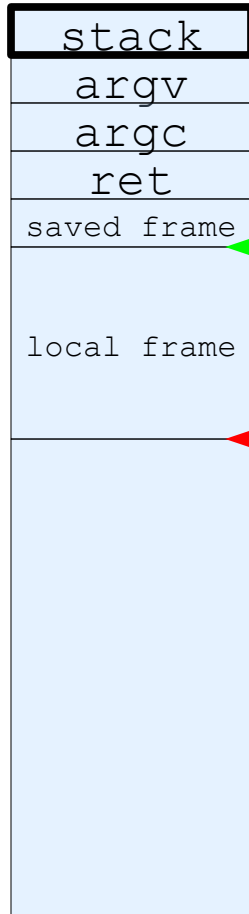
```
080485c0 <main>:
80485c0:  push   %ebp
80485c1:  mov    %esp,%ebp
80485c3:  sub   $0x18,%esp
80485c6:  mov   0xc(%ebp),%eax
80485c9:  add   $0x4,%eax
80485cc:  mov   (%eax),%edx
80485ce:  mov   %edx,0xffffffff(%ebp)
80485d1:  mov   0xffffffff(%ebp),%eax
80485d4:  push  %eax
80485d5:  call  8048500 <authenticate>
80485da:  add   $0x4,%esp

08048500 <authenticate>:
8048500:  push  %ebp
8048501:  mov   %esp,%ebp
8048503:  sub   $0x28,%esp
8048506:  add   $0xffffffff,%esp
8048573:  mov   0x8(%ebp),%eax
8048576:  push  %eax
8048577:  push  $0x8048748
804857c:  lea  0xffffffe0(%ebp),%eax
804857f:  push  %eax
8048580:  call  80483d8 <sprintf>
```

save frame and allocate new one

func params are above frame border

Stack – Function Call



```
080485c0 <main>:
80485c0:  push  %ebp
80485c1:  mov   %esp,%ebp
80485c3:  sub   $0x18,%esp
80485c6:  mov   0xc(%ebp),%eax
80485c9:  add   $0x4,%eax
80485cc:  mov   (%eax),%edx
80485ce:  mov   %edx,0xffffffff(%ebp)
80485d1:  mov   0xffffffff(%ebp),%eax
80485d4:  push  %eax
80485d5:  call  8048500 <authenticate>
80485da:  add   $0x4,%esp

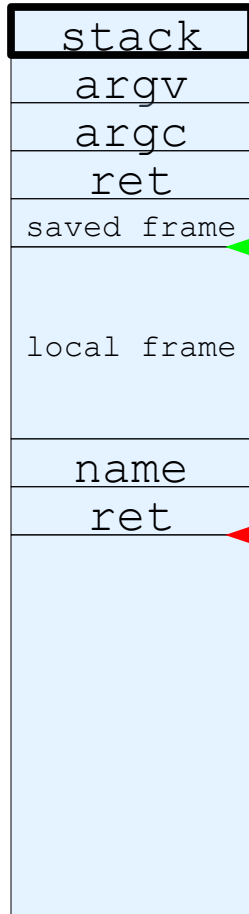
08048500 <authenticate>:
8048500:  push  %ebp
8048501:  mov   %esp,%ebp
8048503:  sub   $0x28,%esp
8048506:  add   $0xffffffff,%esp
8048573:  mov   0x8(%ebp),%eax
8048576:  push  %eax
8048577:  push  $0x8048748
804857c:  lea  0xffffffe0(%ebp),%eax
804857f:  push  %eax
8048580:  call  80483d8 <sprintf>
```

save frame and allocate new one

func params are above frame border

local vars are below frame border

Stack – Function Call



```

080485c0 <main>:
80485c0:    push    %ebp
80485c1:    mov     %esp,%ebp
80485c3:    sub    $0x18,%esp
80485c6:    mov    0xc(%ebp),%eax
80485c9:    add    $0x4,%eax
80485cc:    mov    (%eax),%edx
80485ce:    mov    %edx,0xffffffff(%ebp)
80485d1:    mov    0xffffffff(%ebp),%eax
80485d4:    push   %eax
80485d5:    call   8048500 <authenticate>
80485da:    add    $0x4,%esp

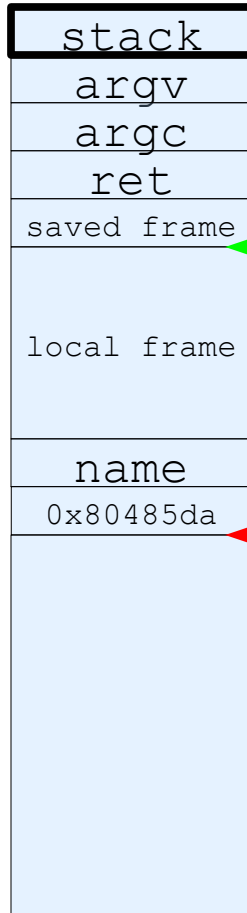
8048500 <authenticate>:
8048500:    push   %ebp
8048501:    mov    %esp,%ebp
8048503:    sub    $0x28,%esp
8048506:    add    $0xffffffff,%esp
8048573:    mov    0x8(%ebp),%eax
8048576:    push   %eax
8048577:    push   $0x8048748
804857c:    lea   0xffffffe0(%ebp),%eax
804857f:    push   %eax
8048580:    call   80483d8 <sprintf>
    
```

```

80485d4:    push   %eax
80485d5:    call   8048500 <authenticate>
80485da:    add    $0x4,%esp
    
```

push params on stack
call function
remove params from stack

Stack – Function Call



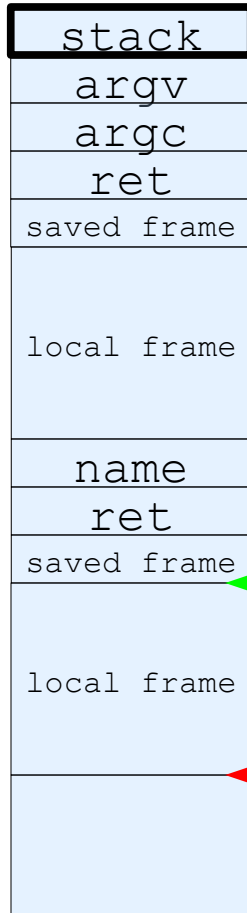
```
080485c0 <main>:
80485c0:   push   %ebp
80485c1:   mov    %esp,%ebp
80485c3:   sub    $0x18,%esp
80485c6:   mov    0xc(%ebp),%eax
80485c9:   add    $0x4,%eax
80485cc:   mov    (%eax),%edx
80485ce:   mov    %edx,0xffffffff(%ebp)
80485d1:   mov    0xffffffff(%ebp),%eax
80485d4:   push   %eax
80485d5:   call   8048500 <authenticate>
80485da:   add    $0x4,%esp

8048500 <authenticate>:
8048500:   push   %ebp
8048501:   mov    %esp,%ebp
8048503:   sub    $0x28,%esp
8048506:   add    $0xffffffff,%esp
8048573:   mov    0x8(%ebp),%eax
8048576:   push   %eax
8048577:   push   $0x8048748
804857c:   lea   0xffffffe0(%ebp),%eax
804857f:   push   %eax
8048580:   call   80483d8 <sprintf>
```

```
80485d4:   push   %eax
80485d5:   call   8048500 <authenticate>
80485da:   add    $0x4,%esp
```

push params on stack
call function
remove params from stack

Stack – Function Call

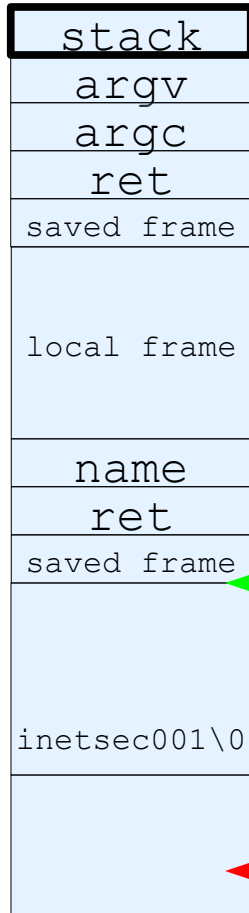


```
080485c0 <main>:
80485c0:    push    %ebp
80485c1:    mov     %esp,%ebp
80485c3:    sub     $0x18,%esp
80485c6:    mov     0xc(%ebp),%eax
80485c9:    add     $0x4,%eax
80485cc:    mov     (%eax),%edx
80485ce:    mov     %edx,0xffffffff(%ebp)
80485d1:    mov     0xffffffff(%ebp),%eax
80485d4:    push   %eax
80485d5:    call   8048500 <authenticate>
80485da:    add     $0x4,%esp

08048500 <authenticate>:
8048500:    push   %ebp
8048501:    mov     %esp,%ebp
8048503:    sub     $0x28,%esp
8048506:    add     $0xffffffff,%esp
8048573:    mov     0x8(%ebp),%eax
8048576:    push   %eax
8048577:    push   $0x8048748
804857c:    lea    0xffffffe0(%ebp),%eax
804857f:    push   %eax
8048580:    call   80483d8 <sprintf>
```

save frame and allocate new one

Stack – Function Call



```

080485c0 <main>:
80485c0:   push   %ebp
80485c1:   mov    %esp,%ebp
80485c3:   sub    $0x18,%esp
80485c6:   mov    0xc(%ebp),%eax
80485c9:   add    $0x4,%eax
80485cc:   mov    (%eax),%edx
80485ce:   mov    %edx,0xffffffff(%ebp)
80485d1:   mov    0xffffffff(%ebp),%eax
80485d4:   push   %eax
80485d5:   call  8048500 <authenticate>
80485da:   add    $0x4,%esp

08048500 <authenticate>:
8048500:   push   %ebp
8048501:   mov    %esp,%ebp
8048503:   sub    $0x28,%esp
8048506:   add    $0xffffffff,%esp
8048573:   mov    0x8(%ebp),%eax
8048576:   push   %eax
8048577:   push   $0x8048748
804857c:   lea   0xffffffe0(%ebp),%eax
804857f:   push   %eax
8048580:   call  80483d8 <sprintf>
    
```

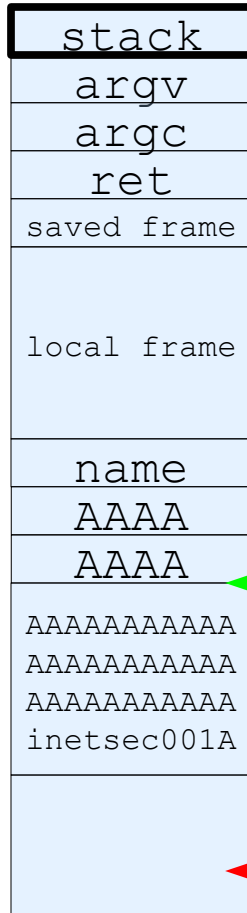
← ebp

← esp

mov 0x8(%ebp),%eax
push %eax
push \$0x8048748
lea 0xffffffe0(%ebp),%eax
push %eax
call 80483d8 <sprintf>

push params on stack
call function
later, remove params from stack

Stack – Function Call



```

080485c0 <main>:
80485c0:   push   %ebp
80485c1:   mov    %esp,%ebp
80485c3:   sub    $0x18,%esp
80485c6:   mov    0xc(%ebp),%eax
80485c9:   add    $0x4,%eax
80485cc:   mov    (%eax),%edx
80485ce:   mov    %edx,0xffffffff(%ebp)
80485d1:   mov    0xffffffff(%ebp),%eax
80485d4:   push   %eax
80485d5:   call   8048500 <authenticate>
80485da:   add    $0x4,%esp

08048500 <authenticate>:
8048500:   push   %ebp
8048501:   mov    %esp,%ebp
8048503:   sub    $0x28,%esp
8048506:   add    $0xffffffff,%esp
8048573:   mov    0x8(%ebp),%eax
8048576:   push   %eax
8048577:   push   $0x8048748
804857c:   lea   0xffffffe0(%ebp),%eax
804857f:   push   %eax
8048580:   call   80483d8 <sprintf>
    
```

push params on stack
call function
later, remove params from stack

Stack – Function Call

stack
argv
argc
ret
saved frame
local frame
name
41 41 41 41
41 41 41 41
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
inetsec001A

```

080485c0 <main>:
80485c0:    push    %ebp
80485c1:    mov     %esp,%ebp
80485c3:    sub     $0x18,%esp
80485c6:    mov     0xc(%ebp),%eax
80485c9:    add     $0x4,%eax
80485cc:    mov     (%eax),%edx
80485ce:    mov     %edx,0xffffffff(%ebp)
80485d1:    mov     0xffffffff(%ebp),%eax
80485d4:    push   %eax
80485d5:    call   8048500 <authenticate>
80485da:    add     $0x4,%esp

08048500 <authenticate>:
8048500:    push   %ebp
8048501:    mov     %esp,%ebp
8048503:    sub     $0x28,%esp
8048506:    add     $0xffffffff,%esp
8048573:    mov     0x8(%ebp),%eax
8048576:    push   %eax
8048577:    push   $0x8048748
804857c:    lea    0xffffffffe0(%ebp),%eax
804857f:    push   %eax
8048580:    call   80483d8 <sprintf>
    
```

← **ebp**

← **esp**

mov 0x8(%ebp),%eax
push %eax
push \$0x8048748
lea 0xffffffffe0(%ebp),%eax
push %eax
call 80483d8 <sprintf>

push params on stack
call function
later, remove params from stack

Stack – Function Call

stack
argv
argc
ret
saved frame
local frame
name
41 41 41 41
41 41 41 41
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
inetsec001A

```

080485c0 <main>:
80485c0:    push   %ebp
80485c1:
80485c3:
80485c6:
80485c9:
80485cc:
80485ce:
80485d1:
80485d4:
80485d5:
80485d8:    add    $0x4,%esp

08048500 <authenticate>:
8048500:    push   %ebp
8048501:    mov    %esp,%ebp
8048503:    sub    $0x28,%esp
8048506:    add    $0xfffffff0,%esp
8048573:    mov    0x8(%ebp),%eax
8048576:    push  %eax
8048577:    push  $0x8048748
804857c:    lea   0xffffffe0(%ebp),%eax
804857f:    push  %eax
8048580:    call  80483d8 <sprintf>
    
```

Failed to read a valid object file image from memory. Program received signal SIGSEGV, Segmentation fault. **0x41414141** in ?? ()

ebp

esp

push params on stack
call function
later, remove params from stack

Stack – Function Call

stack
argv
argc
ret
saved frame
local frame
name
41 41 41 41
41 41 41 41
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
inetsec001A

```

080485c0 <main>:
80485c0:   push   %ebp
80485c1:
80485c3:
80485c6:
80485c9:
80485cc:
80485ce:
80485d1:
80485d4:
80485d5:   jmp    08048500 <authenticate>
80485d8:   add    $0x4,%esp

08048500 <authenticate>:
8048500:   push   %ebp
8048501:   mov    %esp,%ebp
8048503:   sub    $0x28,%esp
8048506:   add    $0xfffffff0,%esp
8048573:   mov    0x8(%ebp),%eax
8048576:   push   %eax
8048577:   push   $0x8048748
804857c:   lea   0xffffffe0(%ebp),%eax
804857f:   push   %eax
8048580:   call  80483d8 <sprintf>
    
```

Instead of injecting AAAA (0x41414141) into the return address, the attacker could inject an arbitrary address and force the program to “return” to the given function instead of main!

ebp

esp

push params on stack
call function
later, remove params from stack

Stack – Function Call

stack
argv
argc
ret
saved frame
local frame
name
41 41 41 41
41 41 41 41
AAAAAAAAAAAA
AAAAAAAAAAAA
AAAAAAAAAAAA
inetsec001A

```

080485c0 <main>:
80485c0:    push   %ebp
80485c1:
80485c3:
80485c6:
80485c9:
80485cc:
80485ce:
80485d1:
80485d4:
80485d5:
80485d8:    jmp    08048500 <authenticate>
80485d9:    add    $0x4,%esp
08048500 <authenticate>:
8048500:
8048501:
8048502:
8048503:
8048504:
8048505:
8048506:
8048507:
8048508:
8048509:
804850a:
804850b:
804850c:
804850d:
804850e:
804850f:
8048510:
8048511:
8048512:
8048513:
8048514:
8048515:
8048516:
8048517:
8048518:
8048519:
804851a:
804851b:
804851c:
804851d:
804851e:
804851f:
8048520:
8048521:
8048522:
8048523:
8048524:
8048525:
8048526:
8048527:
8048528:
8048529:
804852a:
804852b:
804852c:
804852d:
804852e:
804852f:
8048530:
8048531:
8048532:
8048533:
8048534:
8048535:
8048536:
8048537:
8048538:
8048539:
804853a:
804853b:
804853c:
804853d:
804853e:
804853f:
8048540:
8048541:
8048542:
8048543:
8048544:
8048545:
8048546:
8048547:
8048548:
8048549:
804854a:
804854b:
804854c:
804854d:
804854e:
804854f:
8048550:
8048551:
8048552:
8048553:
8048554:
8048555:
8048556:
8048557:
8048558:
8048559:
804855a:
804855b:
804855c:
804855d:
804855e:
804855f:
8048560:
8048561:
8048562:
8048563:
8048564:
8048565:
8048566:
8048567:
8048568:
8048569:
804856a:
804856b:
804856c:
804856d:
804856e:
804856f:
8048570:
8048571:
8048572:
8048573:
8048574:
8048575:
8048576:    push   %eax
8048577:    push   $0x8048748
804857c:    lea   0xffffffe0(%ebp),%eax
804857f:    push   %eax
8048580:    call  80483d8 <sprintf>
    
```

Instead of injecting AAAA (0x41414141) into the return address, the attacker could inject an arbitrary address and force the program to “return” to the given function instead of main!

Instead of injecting “inetsec001AAAA” into the buffer, the attacker could inject arbitrary assembler statements. Thus, all she needs to do is jump to this memory area to be able to execute her malicious code!

push params on stack
call function
later, remove params from stack

Buffer Overflow

- Short recap
 - code gets injected into running process' memory space
 - program accepts more input than there is space allocated
- In particular, an array (or buffer) has not enough space
 - especially easy with C strings (character arrays)
 - plenty of vulnerable library functions
`strcpy`, `strcat`, `gets`, `fgets`, `sprintf` ..
- Input spills to adjacent regions and modifies
 - code pointer or application data
 - normally, this just crashes the program (e.g., `sigsegv`)

Buffer Overflow

- Simple buffer overflow
 - 1) inject some code into the process, and
 - 2) set code pointer to point to this content
- Code pointer
 - most often, the *return address* to the calling function
 - alternatives: function pointers or base-pointer modification
- Effect
 - causes a jump to code under our control
 - successfully modifies execution flow
 - have this code executed with privileges of running process

Shell Code

- Injected code (called *shell code*)
 - usually, a shell should be started
 - for remote exploits - input/output redirection via socket
 - use system call (`execve`) to spawn shell
- System calls
 - mechanism to ask operating system for services
 - transition from user mode to kernel mode
 - different implementations
- Linux system calls are invoked by
 - passing arguments in registers and
 - calling `0x80` interrupt

Shell Code

```
void main(int argc, char **argv) {  
    char *name[2];  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
  
    execve(name[0], &name[0], &name[1]);  
    exit(0);  
}
```

```
int execve(char *file, char *argv[], char *env[])
```

- `file` is name of program to be executed
"/bin/sh"
- `argv` is address of null-terminated argument array
{ "/bin/sh", NULL }
- `env` is address of null-terminated environment array
NULL

Shell Code

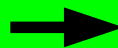
- `file` parameter
 - we need the null terminated string `/bin/sh` somewhere in memory
- `argv` parameter
 - we need the address of the string `/bin/sh` somewhere in memory,
 - followed by a NULL word
- `env` parameter
 - we need a NULL word somewhere in memory
 - we will reuse the null pointer at the end of `argv`

Shell Code

```
int execl(char *file, char *argv[], char *env[])
```

/bin/sh

addr



/bin/sh

0000

addr



0000

Shell Code

*Int. Secure Systems Lab
Vienna University of Technology*

```
int execve(char *file, char *argv[], char *env[])
```

/bin/sh

addr



/bin/sh

0000

addr



0000

Shell Code

```
int execve(char *file, char *argv[], char *env[])
```

/bin/sh

addr →

/bin/sh

0000

addr →

0000

Shell Code

```
int execve(char *file, char *argv[], char *env[])
```

/bin/sh

addr



/bin/sh

0000

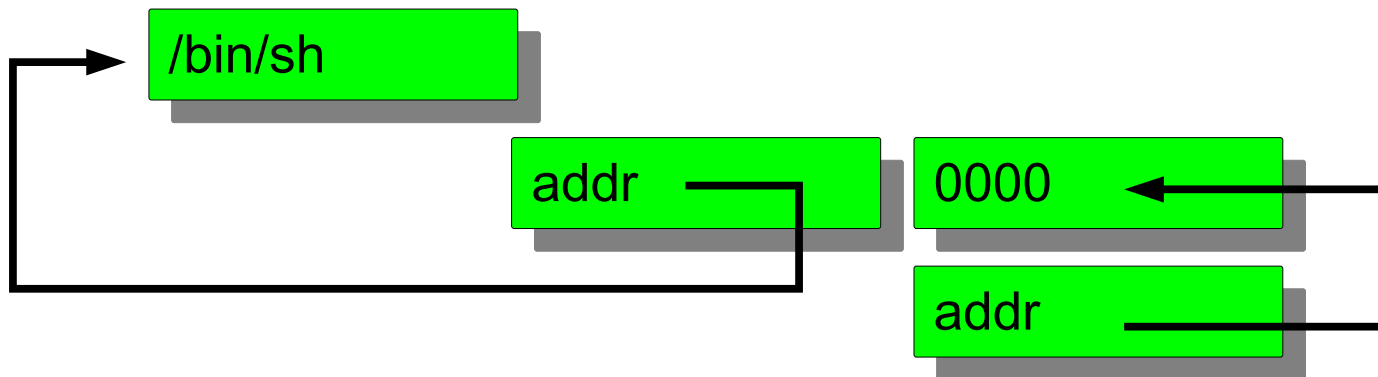
addr



0000

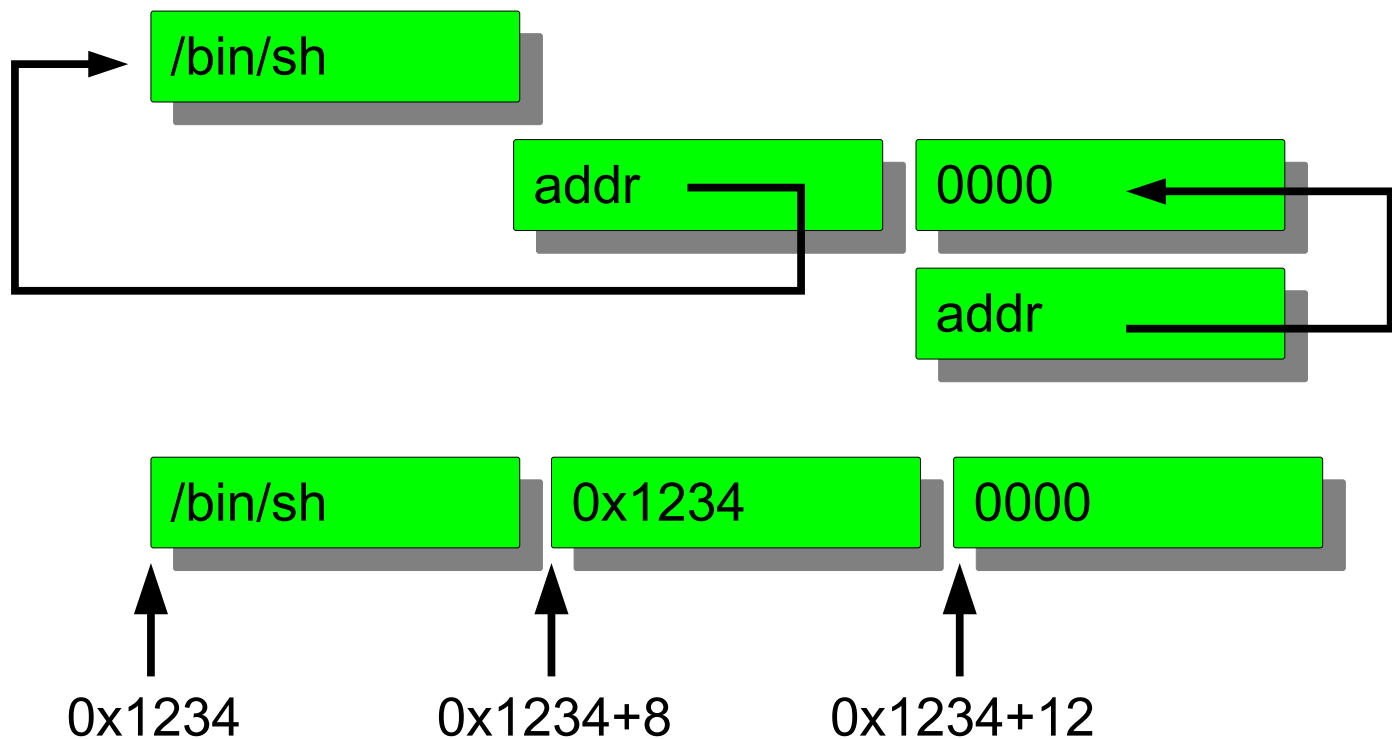
Shell Code

```
int exece(char *file, char *argv[], char *env[])
```



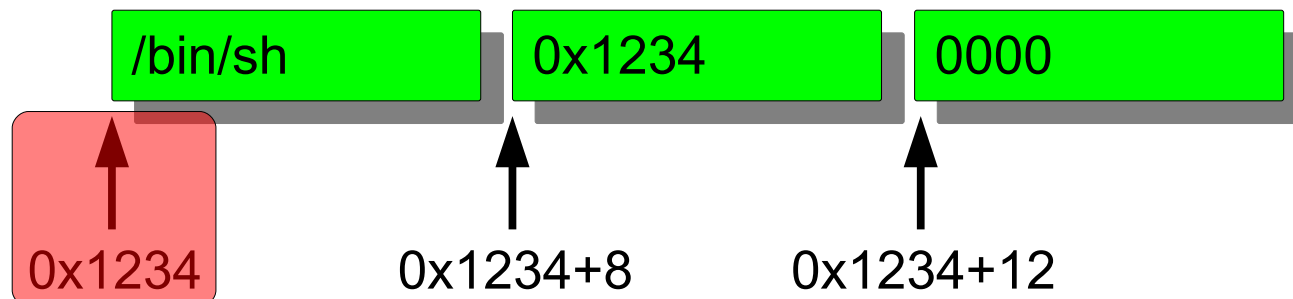
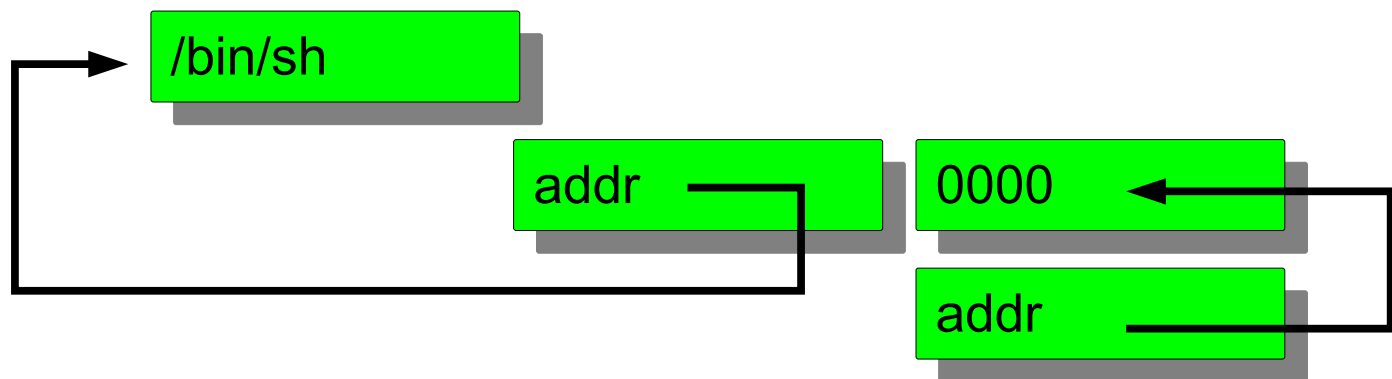
Shell Code

```
int execve(char *file, char *argv[], char *env[])
```



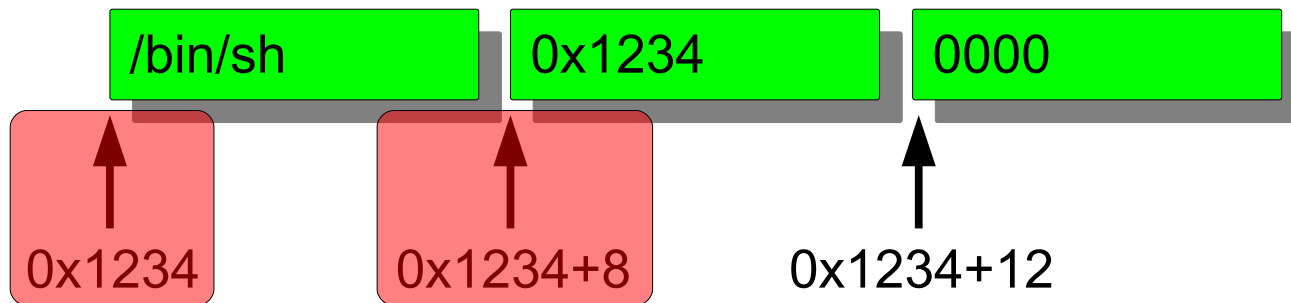
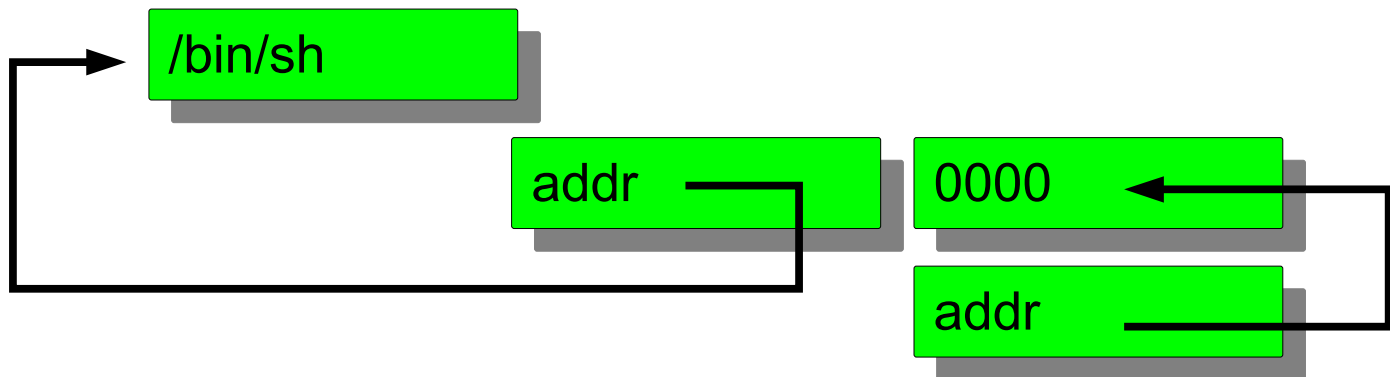
Shell Code

```
int exece(char *file, char *argv[], char *env[])
```



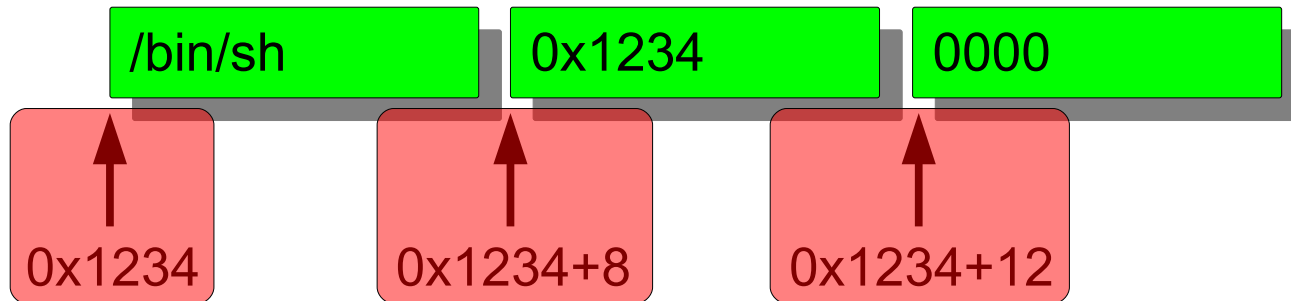
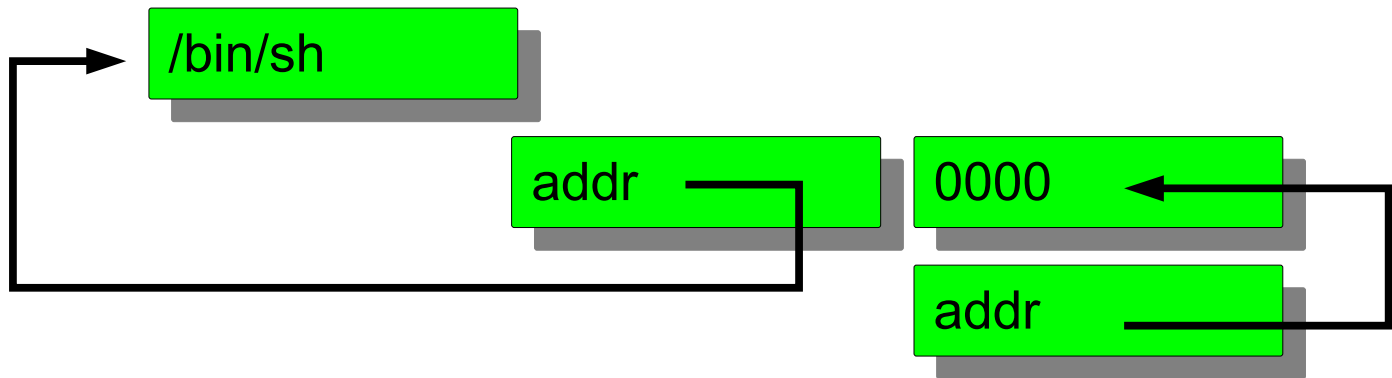
Shell Code

```
int exece(char *file, char *argv[], char *env[])
```



Shell Code

```
int exece(char *file, char *argv[], char *env[])
```



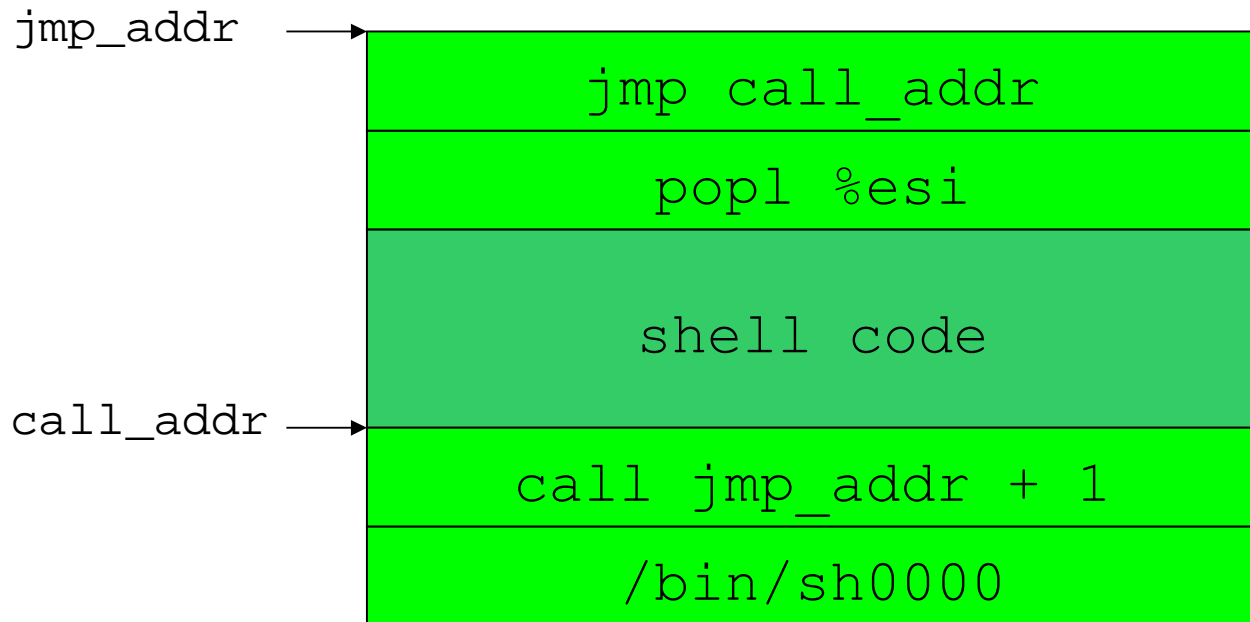
Shell Code

- Spawning the shell in assembly
 - 1) move system call number of `execve` (0x0b) into `%eax`
 - 2) move *address* of string `/bin/sh` into `%ebx`
 - 3) move *address of the address* of `/bin/sh` into `%ecx`
(using `lea`)
 - 4) move *address* of null word into `%edx`
 - 5) execute the interrupt 0x80 or `sysenter` instruction

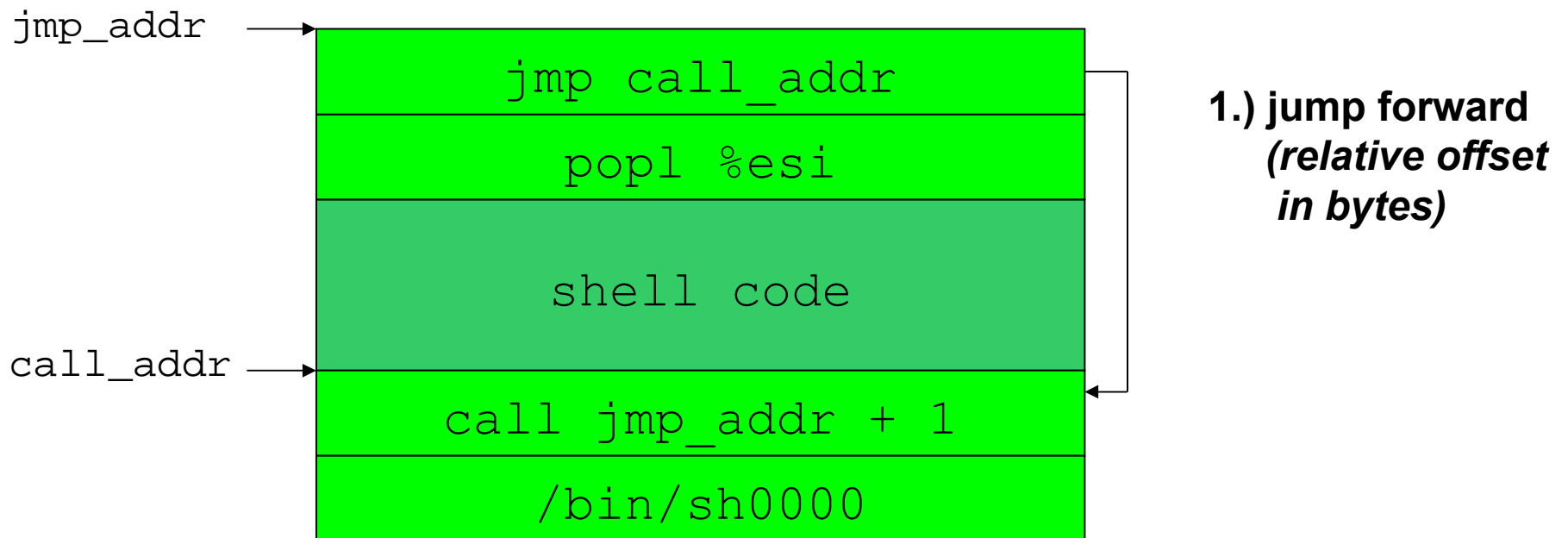
Shell Code

- Problem – position of shell code in memory is unknown
 - how do we determine the *address of string* ?
- Make use of instructions using *relative* addressing
 - `jmp` and `call` variants for relative and absolute targets
- `call` instruction saves IP of next instruction on the stack and then jumps
- Idea
 - `jmp` instruction at beginning of shell code to `call` instruction
 - `call` instruction right before `/bin/sh` string
 - `call` jumps back to first instruction after jump
 - now address of `/bin/sh` is on the stack

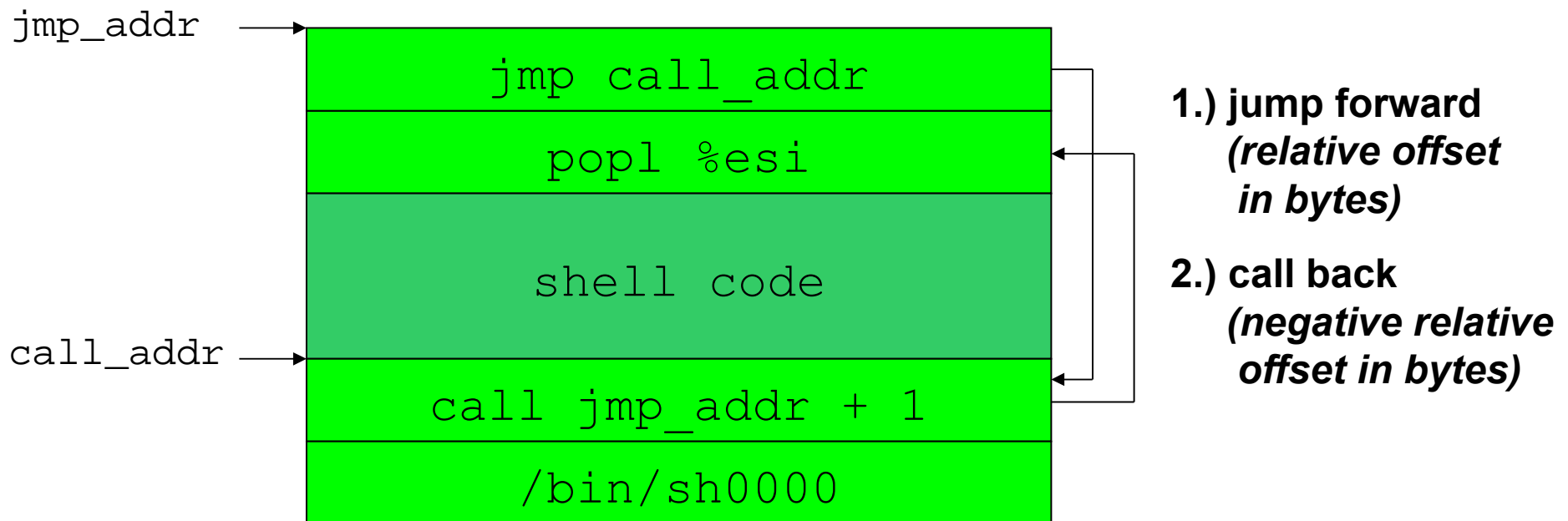
Shell Code



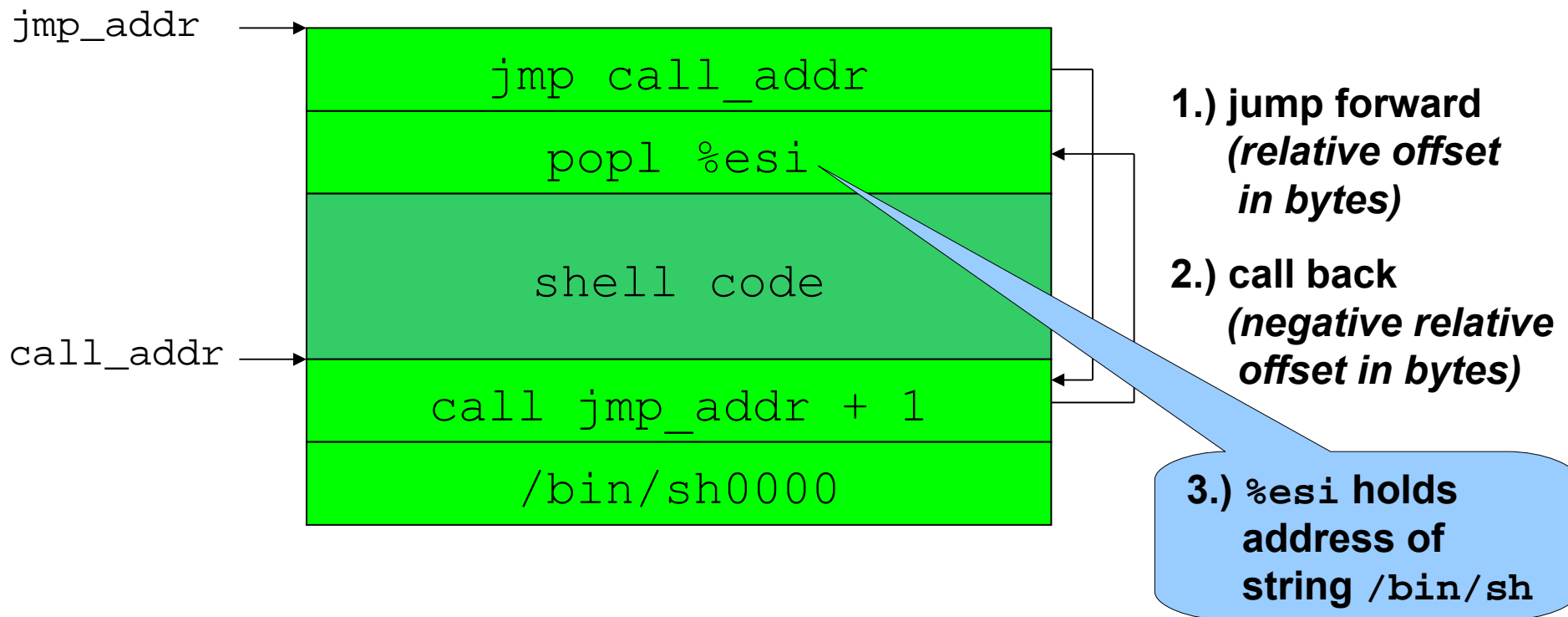
Shell Code



Shell Code



Shell Code



Shell Code

- Shell code is usually copied into a string buffer
- Problem
 - any null byte would stop copying
 - null bytes must be eliminated

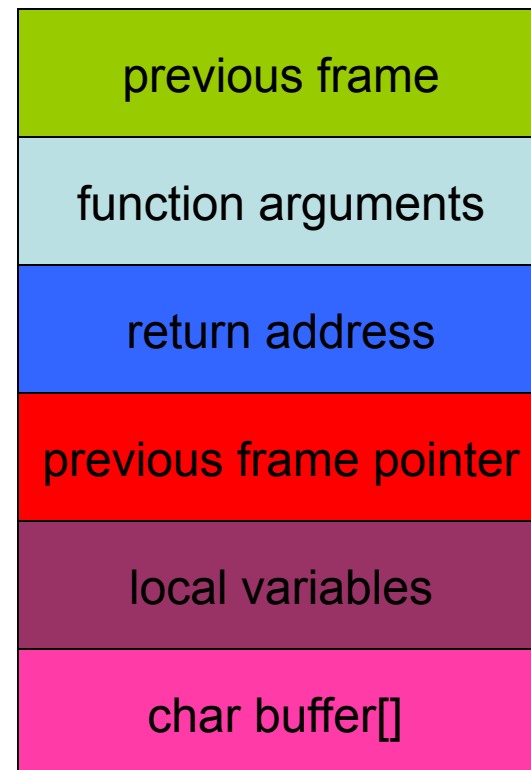
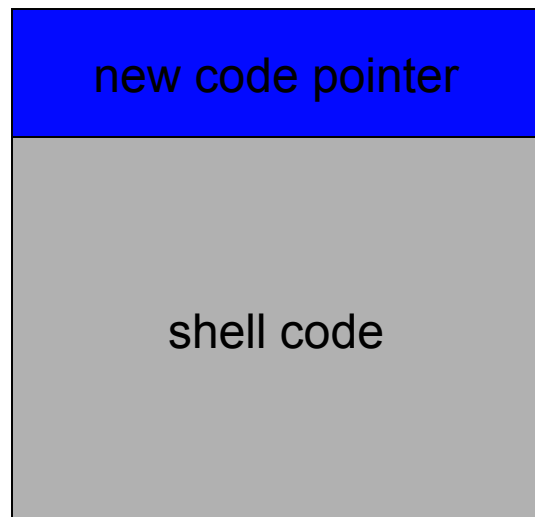
➤ Substitution

`mov 0x0, reg` → `xor reg, reg`

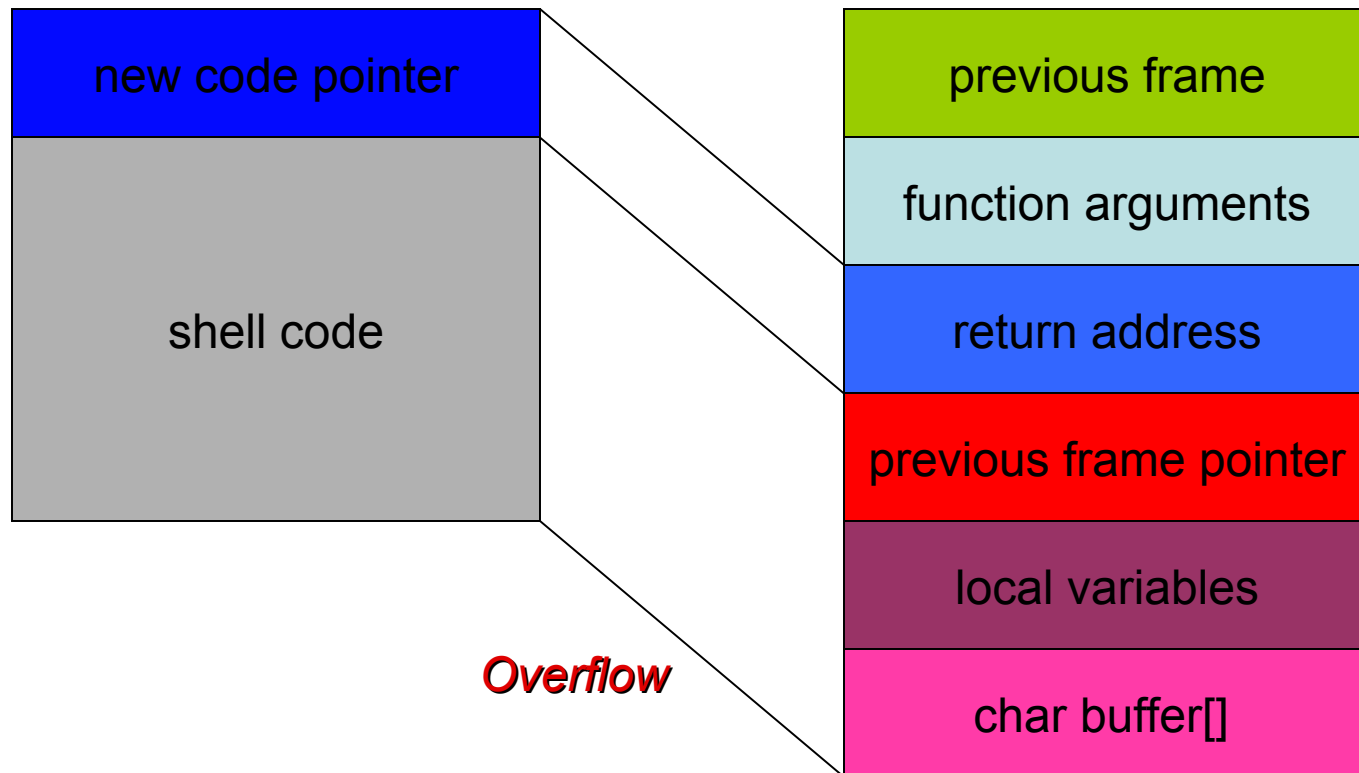
`mov 0x1, reg` → `xor reg, reg`
`inc reg`

e.g. `movl 0x0, %eax` → `xor %eax, %eax`

Pulling It All Together



Pulling It All Together

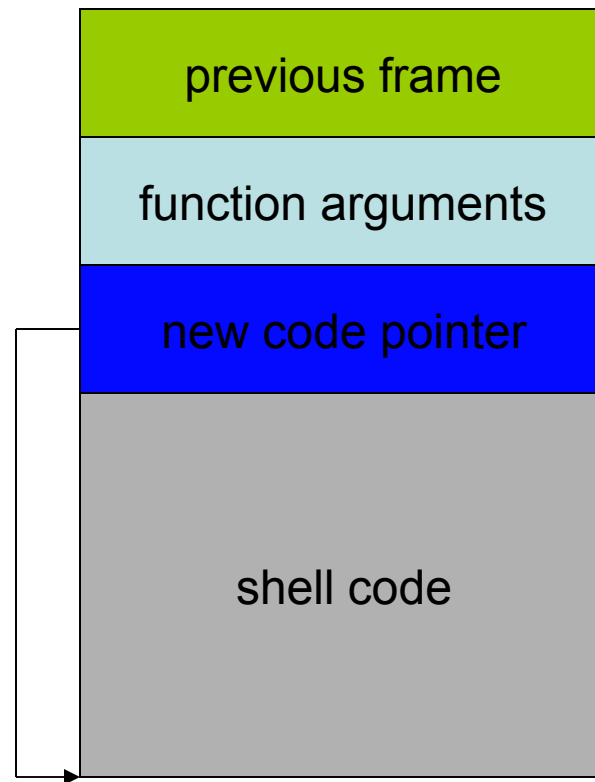


Pulling It All Together

Buffer overflow:

- injects code
- overwrites a code pointer (return address saved on the stack)

New code pointer needs to point to injected code



Code Pointer

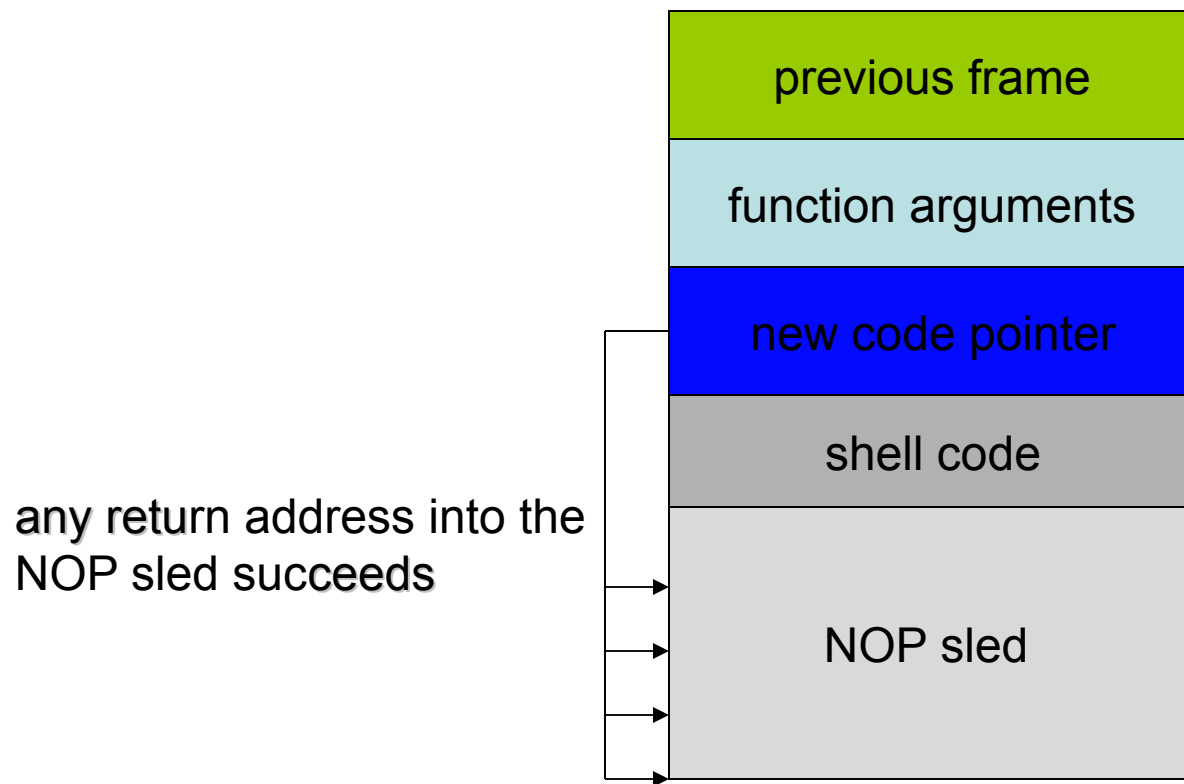
- Code pointer
 - e.g., return address in stack frame
 - must be overwritten with correct value
 - start of exploit code (`jmp`)
 - it has to be guessed (must be very precise)
- Hints
 - stack starts at same address for every program (depending on the security settings, OS might explicitly modify this)
 - can be obtained by function

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

Code Pointer

- NOP (no operation) sled
 - long series of NOP (`0x90`) (no operation) instructions at the beginning of exploit code
 - can be hundreds or even thousands of bytes long
 - return address must not be as precise anymore
 - it is enough to hit the NOP sled
 - doesn't have to be just NOPs: many other harmless instructions can be used (to make detection more difficult)
 - e.g., `ADMmutate`

Code Pointer



Small Buffers

- Buffer can be too small to hold exploit code
- Store exploit code in environmental variable
 - environment stored on stack
 - return address has to be redirected to environment variable
- Advantage
 - exploit code can be arbitrary long
- Disadvantage
 - access to environment needed (typically only for local exploits)

Articles

- Overflow memory region on the stack
 - overflow function return address
 - Phrack 49 -- Aleph One: Smashing the Stack for Fun and Profit
 - Phrack 58 -- Nergel: The advanced return-into-lib(c) exploits
 - overflow function frame (base) pointer
 - Phrack 55 -- klog: The Frame Pointer Overflow
 - overflow longjump buffer
- Overflow (dynamically allocated) memory region on the heap
 - Phrack 57 -- MaXX: Vudo malloc tricks
 - anonymous: Once upon a free() ...
- Overflow function pointers
 - stack, heap, BSS (e.g., PLT)

Defenses

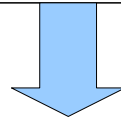
Defenses

- How to defend against buffer overflows?
 - can try to avoid exploitable bugs in programs...
 - ...but bugs still happen
- Compiler and linker can implement some defenses that make exploitation harder (or in some cases impossible)
 - non-executable stack
 - address space randomization
 - stack canaries (not discussed here)
 - ...
- These defenses have been disabled for our challenge environment
 - to make your life easier!

Avoiding Vulnerabilities

- Overflows often occur when using a number of vulnerable C standard library functions
 - strcpy, strcat, sprintf,...
- There are "safe" version of these functions with an extra length parameter limiting how much they write

```
char *strcpy(char *dest, const char *src);
```



```
char *strncpy(char *dest, const char *src, size_t n);
```

- In C++, you can mostly avoid managing buffers manually
 - use std::string instead of char*
 - use std::vector instead of other buffers
 - ...but overflows are still possible (no bounds checks in these classes)

Non-Executable Stack

- Modern CPUs and operating systems support marking memory pages as not executable
 - if a process attempts to jump into such a page, the program crashes
- Make stack non-executable!
 - standard buffer overflows do not work
 - ...but attacker may inject code elsewhere (heap)
- Stronger version "Write XOR execute":
 - no page can be writable and executable
 - cannot inject code anywhere where it is executable!
 - this is called DEP under windows (Data Execution Protection)
- Return-into-libc attacks still possible

Advanced Buffer Overflow

- Advanced buffer overflow
 - 1) set up function *parameters*, and
 - 2) set code pointer to point to *existing code*
- Effect
 - causes a jump to existing code with chosen arguments
 - also successfully modifies execution flow, but
 - cannot execute arbitrary code
- Alternative name: *return-into-libc* exploits
- More details in Advanced Inetsec ;-)

Address-Space Layout Randomization (ASLR)

*Int. Secure Systems Lab
Vienna University of Technology*

- Attacks rely on overwriting a code pointer
 - e.g.: return address on stack
- Need to overwrite it to point to some specific (injected) code that the attacker wants to run
 - need to know the address of that code in memory
 - even with NOP sled, need to know approximate address

Address-Space Layout Randomization (ASLR)

*Int. Secure Systems Lab
Vienna University of Technology*

- Attacks rely on overwriting a code pointer
 - e.g.: return address on stack
- Need to overwrite it to point to some specific (injected) code that the attacker wants to run
 - need to know the address of that code in memory
 - even with NOP sled, need to know approximate address



IDEA:

- randomize memory layout!
- different layout for each execution

Address-Space Layout Randomization (ASLR)

*Int. Secure Systems Lab
Vienna University of Technology*

- At each execution of a program, the memory layout is different
 - libc, and other dynamically linked libraries are linked in at different (random) addresses each time
 - the code segment is also relocated to a random address
- Makes it hard to guess addresses for exploitation
 - address of buffer to jump into (NOP sled no longer enough!)
 - address of libc functions to call

Address-Space Layout Randomization (ASLR)

*Int. Secure Systems Lab
Vienna University of Technology*

- Deployed on all modern systems (linux, Windows)
 - enabled by default
- Full ASLR requires relocatable binaries
 - if only libraries are relocated, defense is weak
 - not as widely deployed
- On 32-bit systems, defense can be broken with brute-force (guessing) attack
 1. Try an address (more or less) at random.
 2. Program jumps to the address.
 3. Program will (usually) just crash.
 4. Go back to step 1 and try again.

Conclusion

- Buffer overflows
 - implementation flaw
 - occur when an application receives more input than there is space allocated for this input
- Exploit steps
 - inject shell code or parameters
 - practical issues: locate shell code in memory, NULL bytes, NOP sled
 - change code pointer
- Code pointer
 - various possibilities to change
 - return address, frame pointer, jump buffer, function pointer
- Defenses:
 - Write XOR Execute, ASLR