Introduction to Information Security

0368-3065, Spring 2015

Lecture 1:
Introduction,
Control Hijacking (1/2)

Eran Tromer

Slides credit:
Avishai Wool, Tel Aviv University

Administration

- Lecturer:
  Eran Tromer
  `tromer@cs.tau.ac.il` (include “infosec15” in email subject)
- Leading teaching assistants:
  Itamar Gilad, Nir Krakowski
- Teaching assistant / exercise checker:
  Michal Shagam
- Course web site:
  `http://course.cs.tau.ac.il/infosec15/`
  - Read the Instructions
  - Join the mailing list (automatic if you’re already registered)
  - Fill in the questionnaire

JOIN

FILL

Exercises

- Practical emphasis, challenging hands-on exercises.
- Final grade: 65% exam, 35% exercises
- Attend recitations for necessary context.

Further details in recitation.

Motivation

Information Security

Computer and Network Security

Cyber Security

A brief history of cyber

1948 … 2007 2010

Cybernet…

Cybernet…

The New Yo…

Data Breach at Sec…

27, 2011

Finance: …

National …

Physical (Stuxnet)

Cybersecurity ~
harming computers and things they control

Information / computer / network / control-system security, cryptography

Security goals and threats

GOAL	THREAT
• Data confidentiality	• Data exposure
• Data integrity	• Data modification
• System availability	• Denial of service
• User authentication	• Masquerading
• Privilege separation	• Privilege elevation

Prerequisite to many attacks: getting malicious code to run on victim's computer system

Basic Control Hijacking Attacks

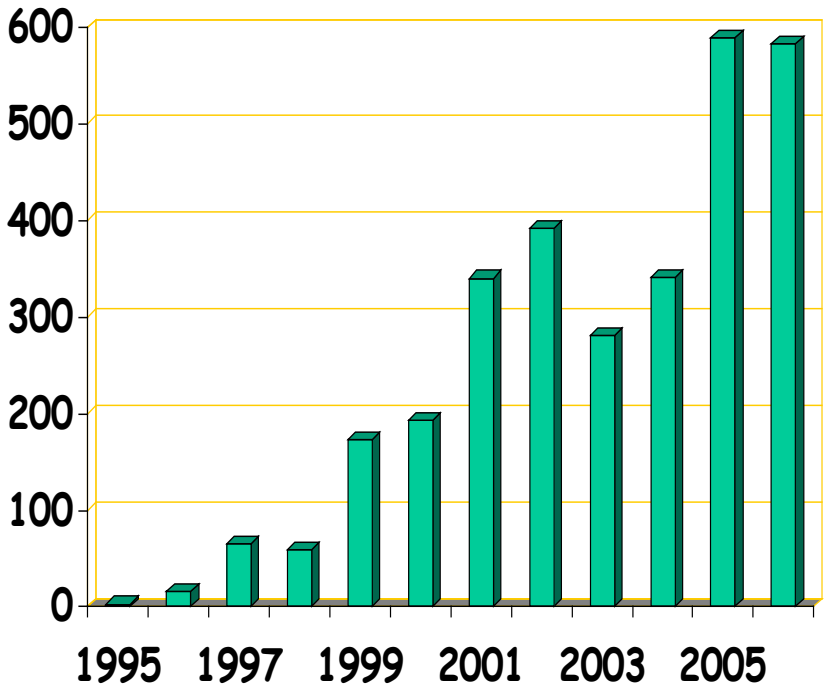
Control hijacking attacks

- Attacker's goal:
  - Take over target machine (e.g. web server)
    - Execute arbitrary code on target by hijacking application control flow

- Examples.
  - Buffer overflow attacks
  - Integer overflow attacks
  - Format string vulnerabilities

Example 1: buffer overflows

- Extremely common bug in C/C++ programs.
  - First major exploit: 1988 Internet Worm, `fingerd`.



≈20% of all vuln.

2005-2007: ≈ 10%

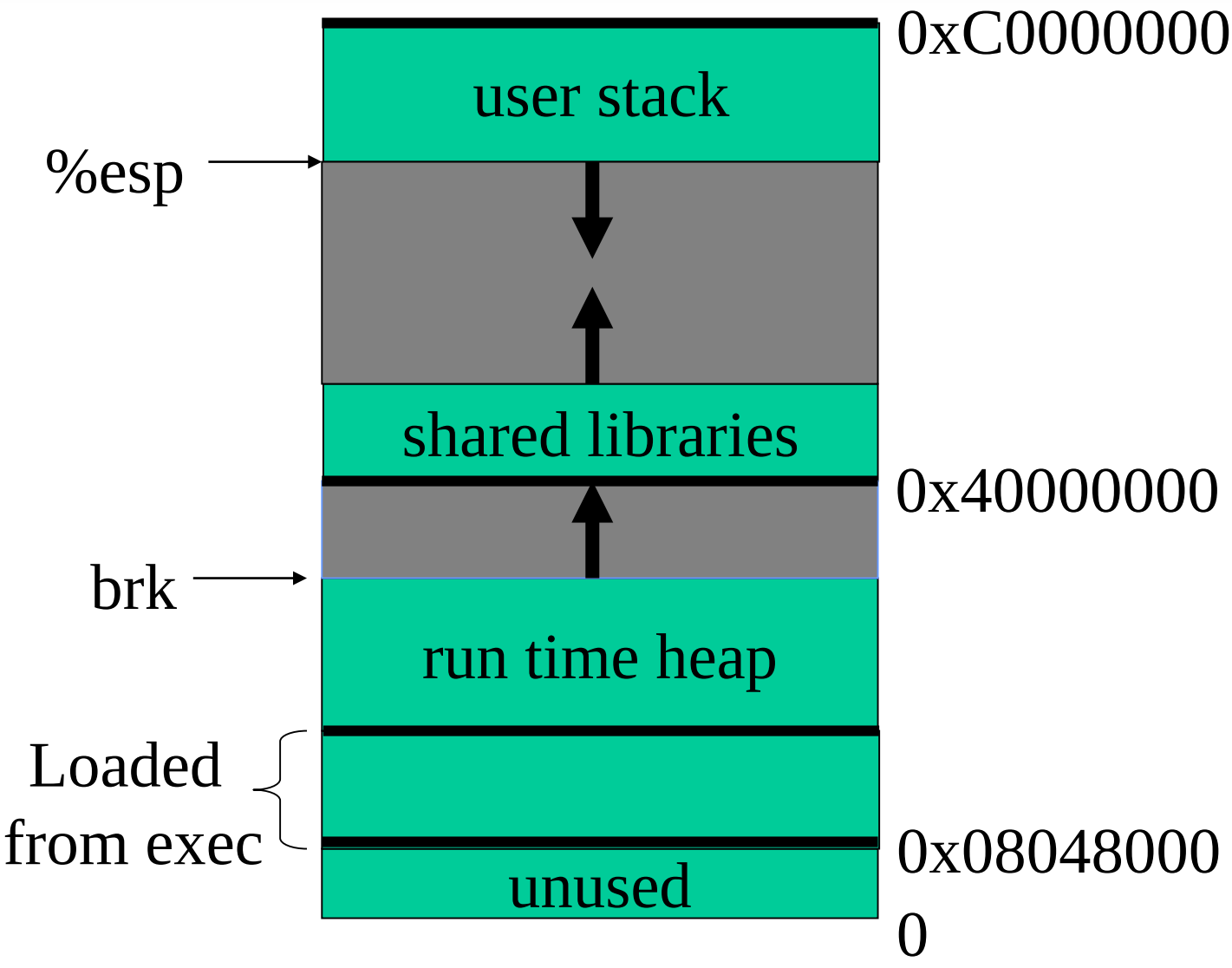
Source: NVD/CVE

What is needed

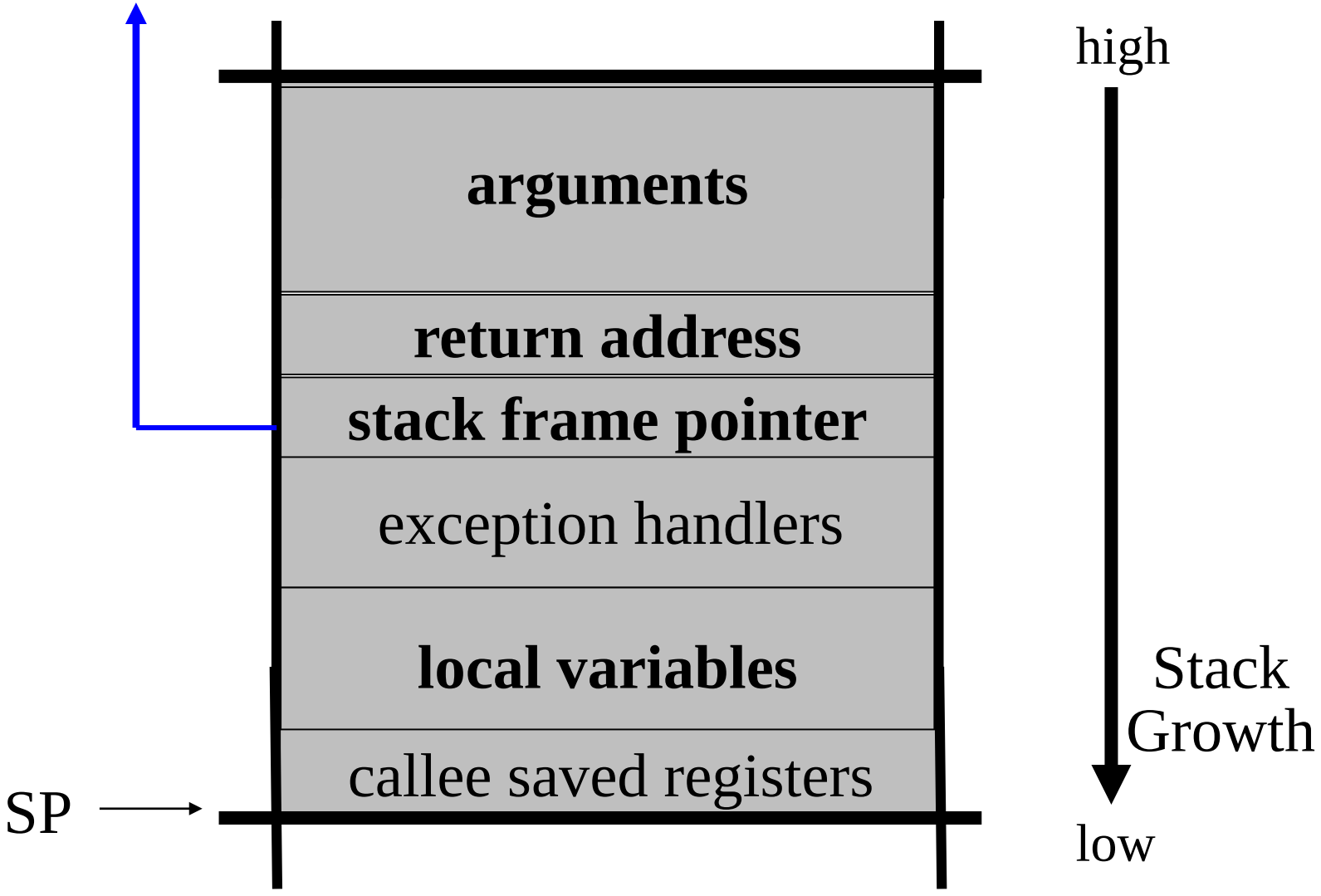
- Understanding C functions, the stack, and the heap.
- Know how system calls are made
- The exec() system call

- Attacker needs to know which CPU and OS used on the target machine:
  - Our examples are for x86 running Linux or Windows
  - Details vary slightly between CPUs and OSs:
    - Little endian vs. big endian (x86 vs. Motorola)
    - Stack Frame structure (Unix vs. Windows)

Linux process memory layout



Stack Frame

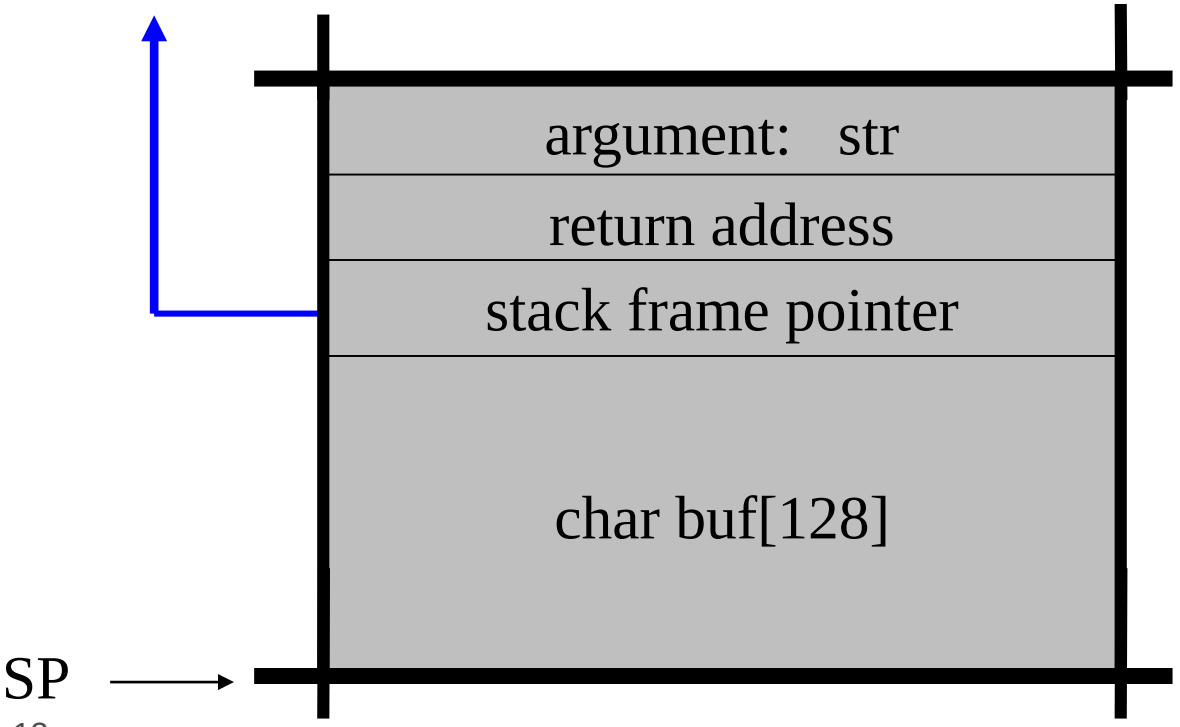

high
arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
SP
Stack
Growth
low

What are buffer overflows?

Suppose a web server contains a function:

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

When func() is called stack looks like:



What are buffer overflows?

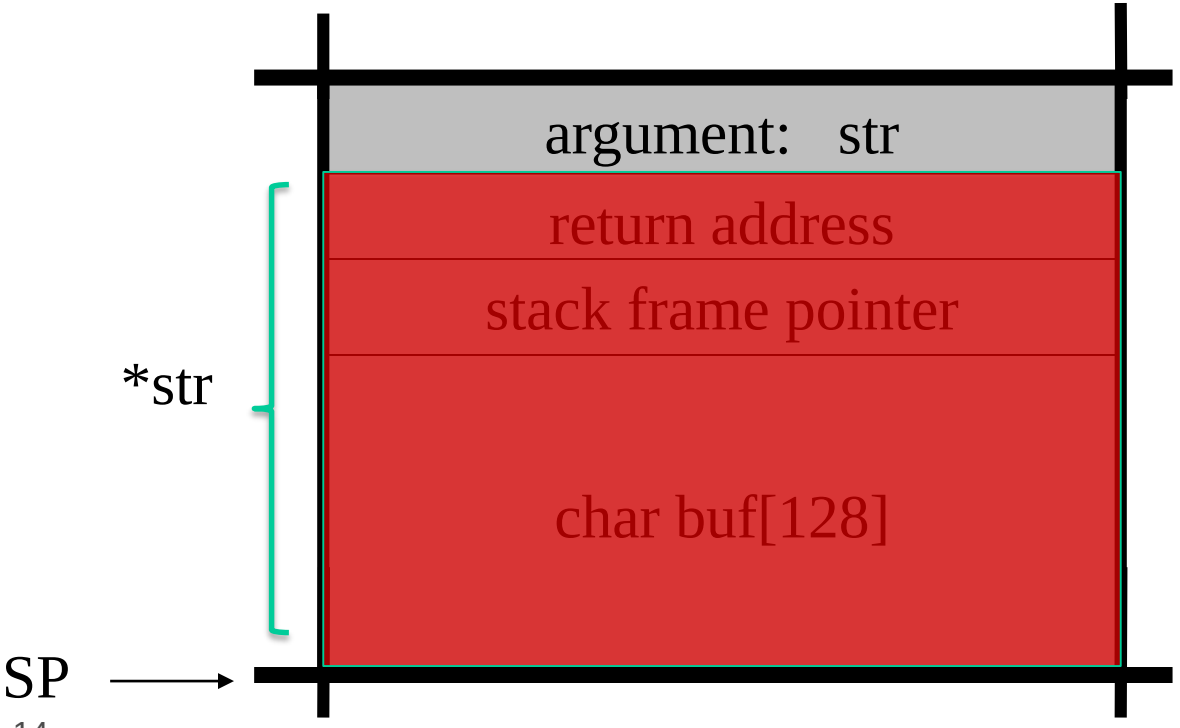
What if **`*str`** is 136 bytes long?

[this is a software bug]

After **`strcpy`**:

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```



Problem:

no length checking in strcpy()

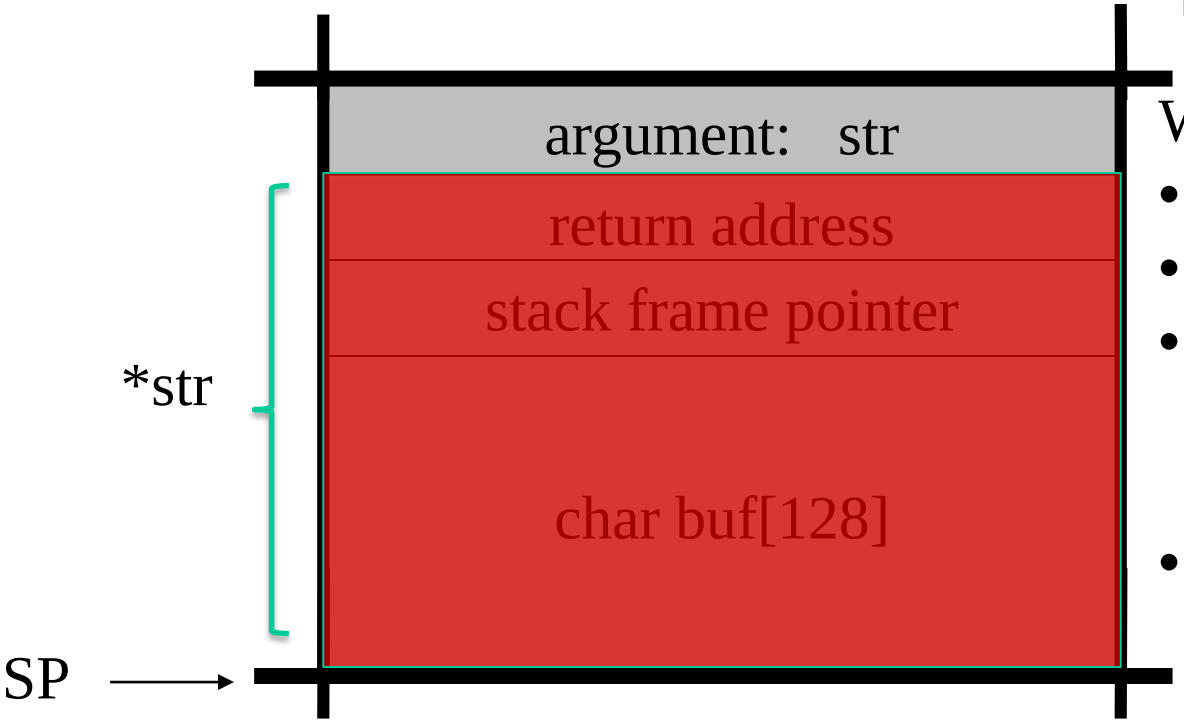
What are buffer overflows?

What if **`*str`** is 136 bytes long?

After **`strcpy`**:

```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```



When func returns, usually:

- Some junk in return address
- CPU tries to jump to junk address
- "invalid instruction at 0x…." or "segmentation fault"

- Result: Web server crash (OS kills the process)

Now let's think like an attacker

- Attacker discovers the software bug
- Attacker has control over the content & length of `str`
  - E.g., maybe `str` is sent in the HTTP request that the attacker can fill
- Result: attacker can place binary values in the end of `str` to select the address that the CPU will jump to upon return
- This is **Control Hijacking**

Where to jump to?

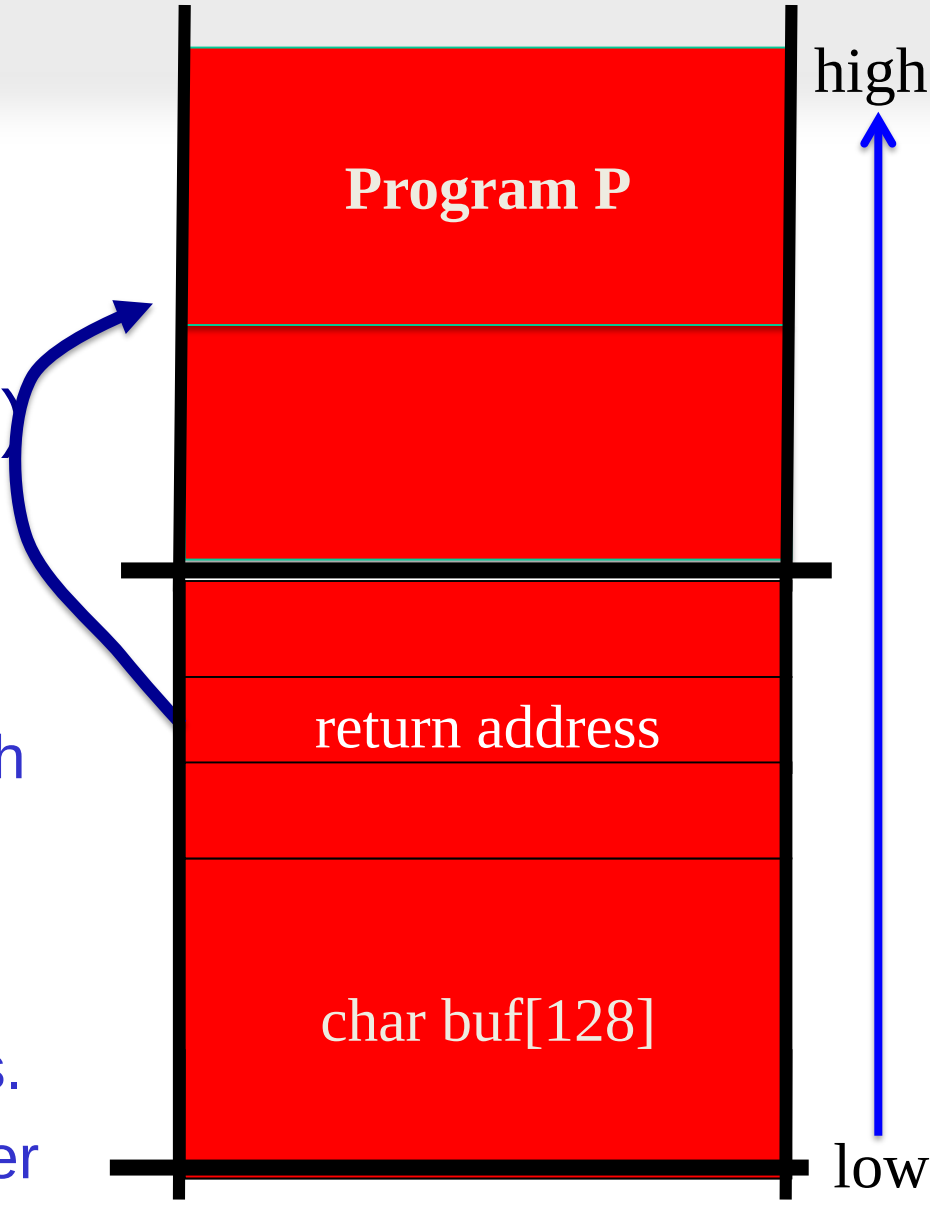
- Attacker goal: jump to an address that runs attacker-written code
  - But the process memory only has legitimate user & OS code
- Need to inject his own code somehow

- Solution: make the buffer overflow even longer
- Place binary machine instructions at the end of `str`

Basic stack exploit

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

Program P: `exec("/bin/sh")`
(exact shell code by Aleph One)



- Attack code P runs *in stack*.
- When func() exits, the shell runs with its stdin and stdout (hopefully) redirected to the TCP connection.
- The attacker, on the other side of the TCP connection, gets full shell access.
- Attacker gains full privileges of the user (UID, GID) running the server.

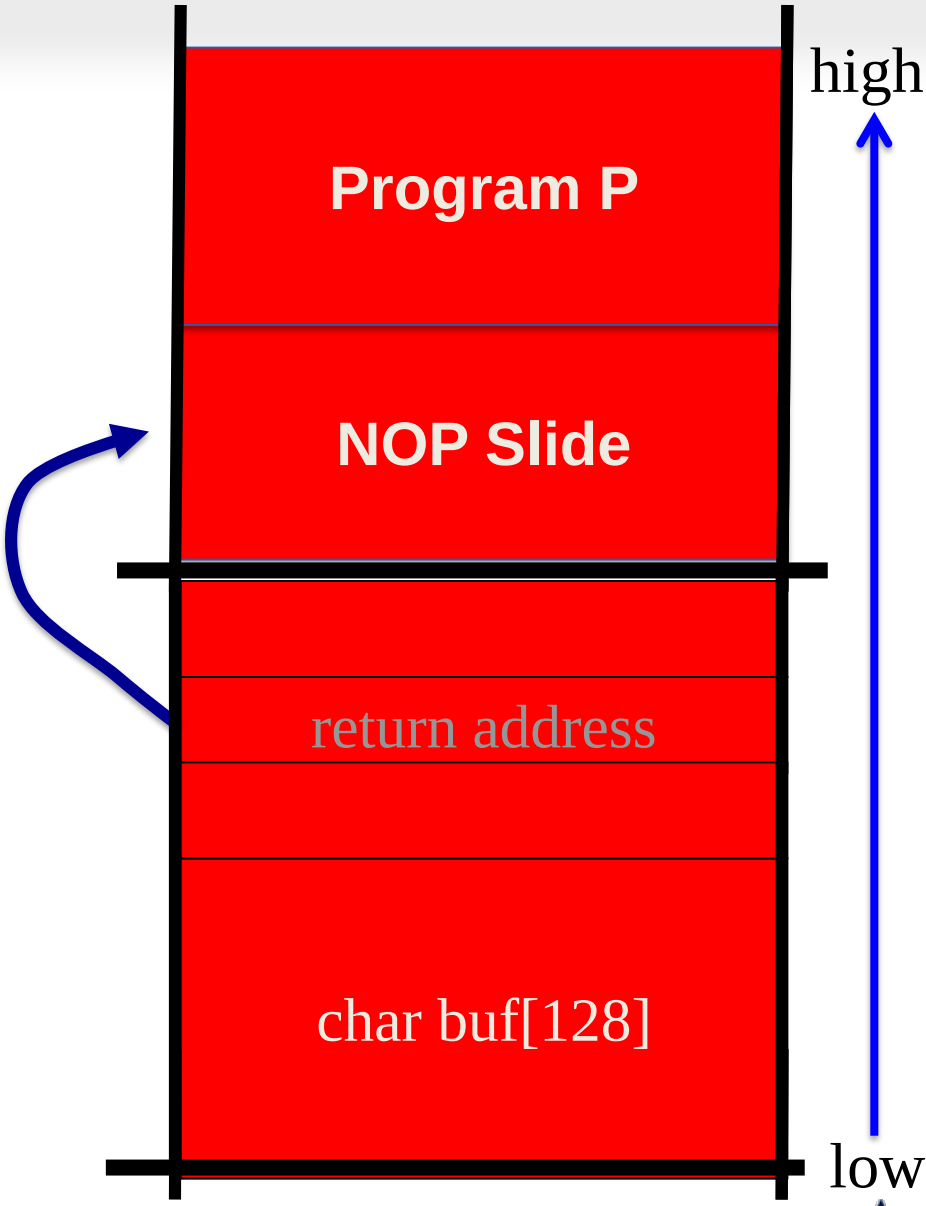
The NOP slide

Problem: how does attacker determine the return address?

Solution: NOP slide

- Guess approximate stack state when `func()` is called
- Insert many NOPs before program P:

```
nop ,
xor eax,eax ,
inc ax
...
```



Details and examples

- Some complications:
  - Program P should not contain the '\0' character.
  - Overflow should not crash program before func() exits.

- Sample remote stack smashing overflows:
  - (2007) Overflow in Windows animated cursors (ANI). LoadAniIcon()
  - (2005) Overflow in Symantec Virus Detection

```
test.GetPrivateProfileString "file", [long string]
```

Many unsafe libc functions

```
strcpy (char *dest,  const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf ( const char *format, … )
```

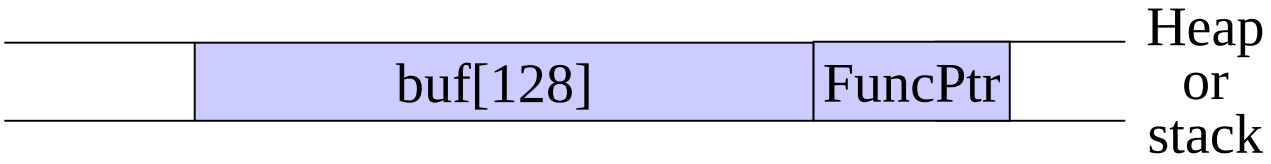
and many more.

- "Safe" libc versions `strncpy()`, `strncat()` are misleading
  - E.g.,. `strncpy()` may leave string unterminated.
- Windows C run time (CRT):
  - `strcpy_s (*dest, DestSize, *src)` ensures proper termination.

Buffer overflow opportunities

- Exception handlers: (Windows SEH attacks)
  - Overwrite the address of an exception handler in stack frame.

- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)



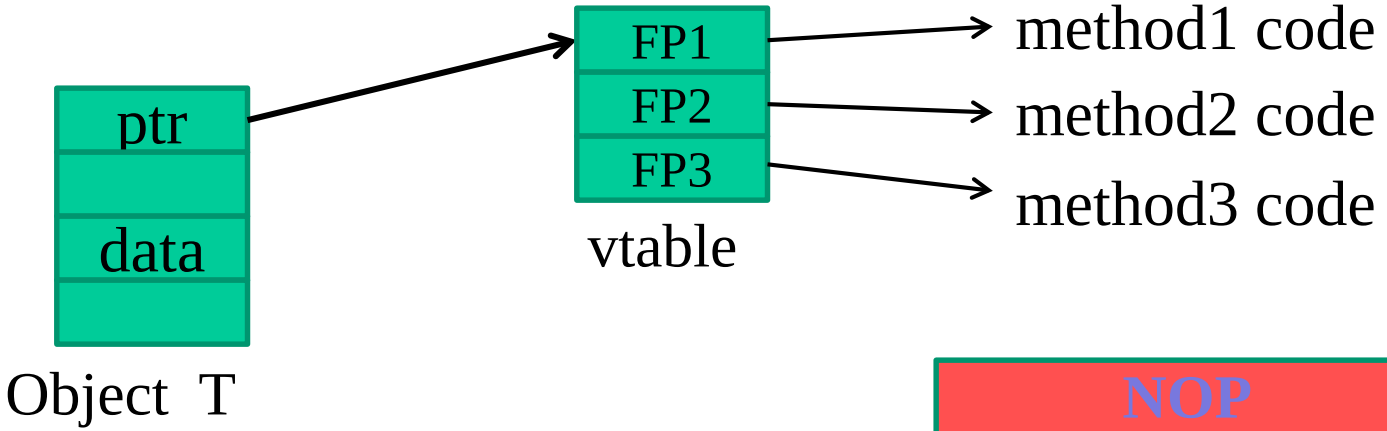
  - Overflowing `buf` will override function pointer.

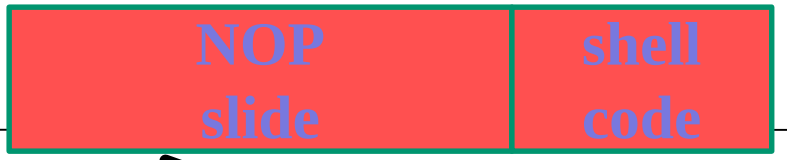
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - Overflowing `buf` next to `pos` overrides value of `pos`.

Corrupting method pointers

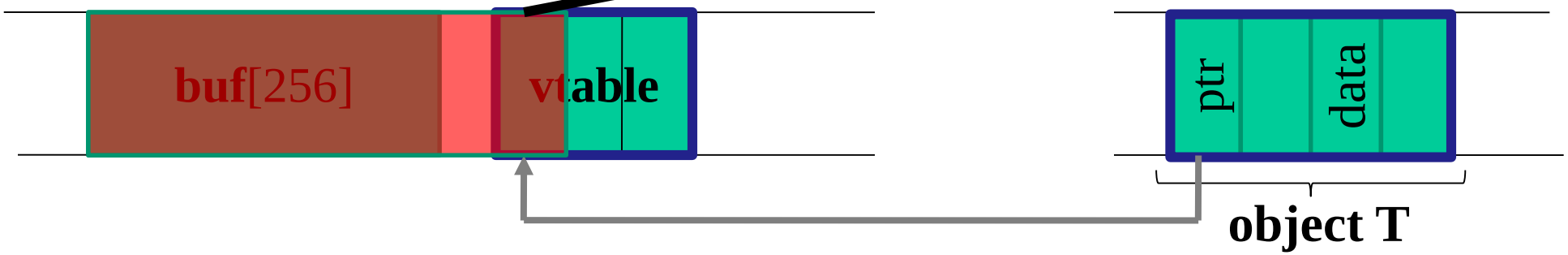
http://phrack.org/issues/56/8.html

- Compiler generated function pointers (e.g. virtual methods in C++ code)





- After overflow of **`buf`**:



Finding buffer overflows

- To find overflow:
  - Run web server on local machine
  - Issue malformed requests (ending with "$$$$$")
    - Many automated tools exist (called **fuzzers**)
  - If web server crashes, search core dump for "$$$$$" to find overflow location

- Construct exploit (not easy given latest defenses)

More Control Hijacking Attacks

More Hijacking Opportunities

- **Integer overflows**: (e.g. MS DirectX MIDI Lib)
- **Double free**: double free space on heap.
  - Can cause memory manager to write data to specific location
  - Examples: CVS server
- **Format string vulnerabilities**

Integer Overflow

- Integer overflow: an arithmetic operation attempts to create a numeric value that is larger than can be represented within the available storage space.
- Example:

Test 1:

```
short x = 30000;
short y = 30000;
printf("%d\n", x+y);
```

Test 2:

```
short x = 30000;
short y = 30000;
short z = x + y;
printf("%d\n", z)
```

Will two programs output the same?

C Data Types

• short int	6bits	[-32,768; 32,767]
• unsigned short int	16bits	[0; 65,535]
• unsigned int	32bits	[0; 4,294,967,295]
• Int	32bits	[-2,147,483,648; 2,147,483,647]
• long int	32bits	[-2,147,483,648; 2,147,483,647]
• char	8bits	[-128; 127]
• unsigned char	8bits	[0; 255]
• long long int		
• unsinged long long int		

When does casting occur in C?

- When assigning to a different data type
- For binary operators +, -, *, /, %, &, |, ^,
  - if either operand is an unsigned long, both are cast to an unsigned long
  - in all other cases where both operands are 32-bits or less, the arguments are both upcast to int, and the result is an int
- For unary operators
  - ~ changes type, e.g., ~((unsigned short)0) is int
  - ++ and -- does not change type

Where Does Integer Overflow Matter?

- Allocating spaces using calculation.
- Calculating indexes into arrays
- Checking whether an overflow could occur

- Direct causes:
  - Truncation; Integer casting

Integer Overflow: Example 1

```
const long MAX_LEN = 20000;
char   buf[MAX_LEN];
short len = strlen(input);
if (len < MAX_LEN)
    strcpy(buf, input);
```

Can a buffer overflow attack occur?

If so, how long does input needs to be?

Integer Overflows Example 2 (see Phrack 60)

Problem: what happens when int exceeds max value?

```
int m;(32 bits)
short s;(16 bits)
char c; (8 bits)
```

```
c = 0x80 + 0x80 = 128 + 128 ⇒ c = 0
s = 0xff80 + 0x80           ⇒ s = 0
m = 0xffffff80 + 0x80       ⇒ m = 0
```

Can this be exploited?

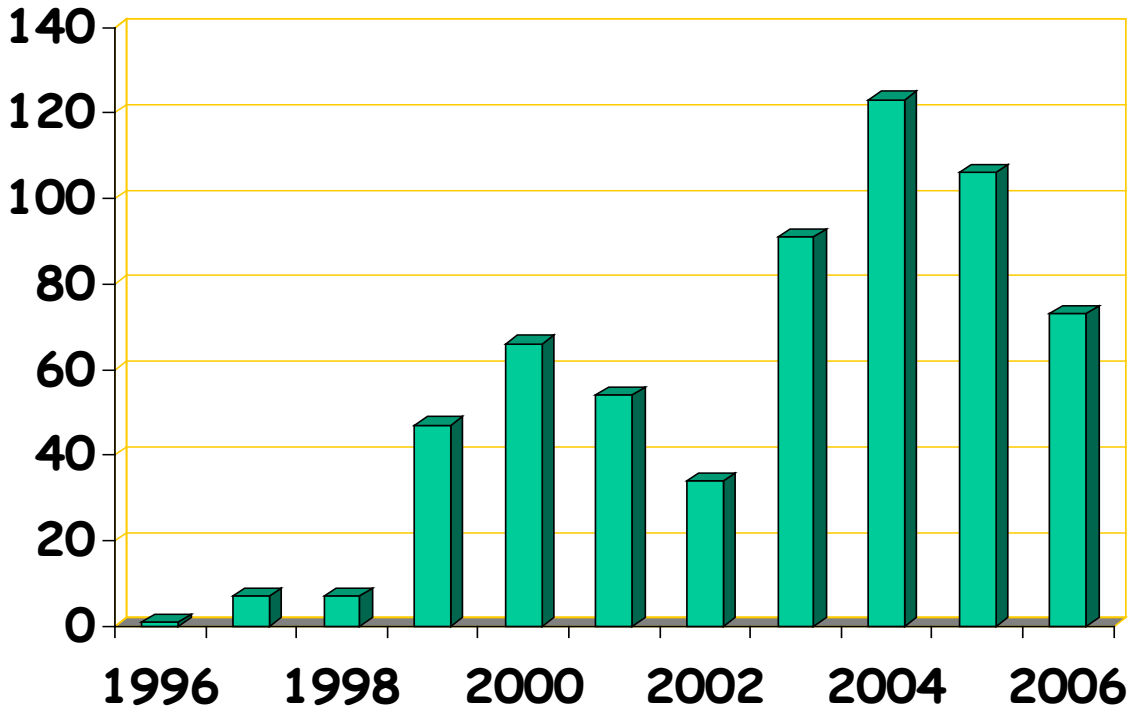
```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
    char temp[256];
    if  (len1 + len2 > 256)  {return -1}    // length check
    memcpy(temp, buf1, len1);              // concatenate buffers
    memcpy(temp+len1, buf2, len2);
    do-something(temp);                    // do stuff
}
```

What if **len1 = 0x80, len2 = 0xffffff80** ?

⇒ len1+len2 = 0

Second memcpy() will overflow stack (or heap) !!

Integer overflow exploit stats



Source: NVD/CVE

Format string bugs

Format string problem

- Want to print a message (without arguments) - via fprintf
- Correct form:

```
fprintf( stdout, "%s", input);
```

- Bug:

```
int func(char *input)  {
        fprintf( stderr, input);
      }
```

- "input" is treated as a format string – missing arguments
- If "input" includes % formats – access places on stack

Problem: what if `*input = "%s%s%s%s%s%s%s"` ?

  - Most likely program will crash (Denial of Service, violates Availability)
  - If not, program will print memory contents. Privacy?

Format string attacks (“%n”)

- “%n” format **writes** the number of bytes formatted so far into a variable!

- `printf(“abc %n”, &x)` will change the value of the variable `x` (to 4)
  - in other words, the parameter value on the stack is interpreted as a pointer to an integer value, and the place pointed by the pointer is overwritten

Exploit

- Dumping arbitrary memory:
  - Walk up stack until desired pointer is found.
  - `printf("%08x.%08x.%08x.%08x|%s|")`
- Writing to arbitrary memory:
  - `printf("hello %n", &temp)` writes “6” into temp.
  - `printf("%08x.%08x.%08x.%08x.%n")`

History

- First exploit discovered in June 2000.
- Examples:
  - wu-ftpd 2.* : remote root
  - Linux rpc.statd: remote root
  - IRIX telnetd: remote root
  - BSD chpass: local root

Vulnerable functions

Any function using a format string.

Printing:

```
printf, fprintf, sprintf, …
vprintf, vfprintf, vsprintf, …
```

Logging:

```
syslog,  err, warn
```