

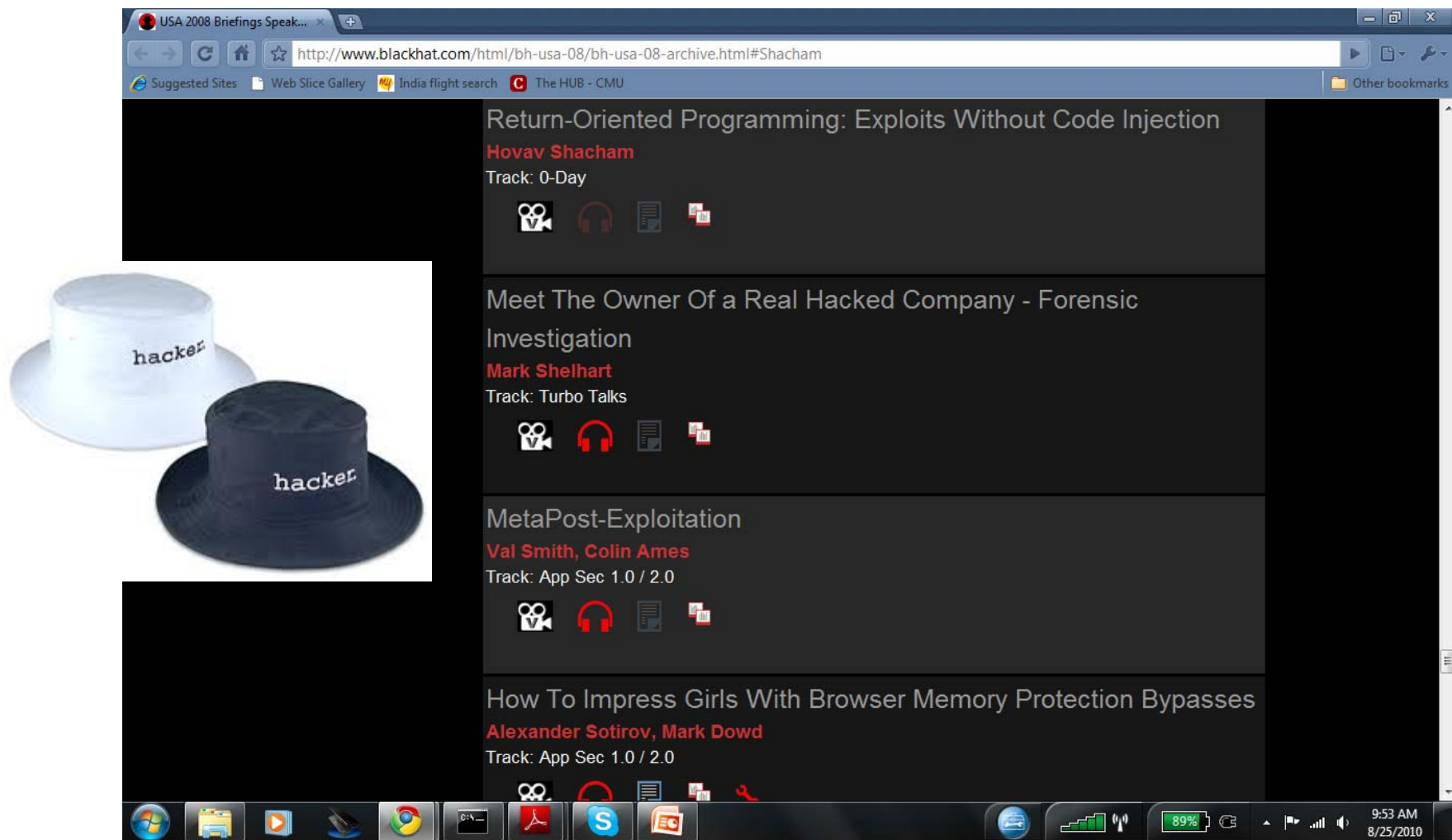
18732: Secure Software Systems

# Control Hijacking Attacks

Anupam Datta

CMU  
Fall 2010

# Blackhat and Whitehat



The screenshot shows a web browser window displaying the Blackhat USA 2008 archive page. The URL is <http://www.blackhat.com/html/bh-usa-08/bh-usa-08-archive.html#Shacham>. The page lists several talks:

- Return-Oriented Programming: Exploits Without Code Injection**  
Hovav Shacham  
Track: 0-Day
- Meet The Owner Of a Real Hacked Company - Forensic Investigation**  
Mark Shelhart  
Track: Turbo Talks
- MetaPost-Exploitation**  
Val Smith, Colin Ames  
Track: App Sec 1.0 / 2.0
- How To Impress Girls With Browser Memory Protection Bypasses**  
Alexander Sotirov, Mark Dowd  
Track: App Sec 1.0 / 2.0

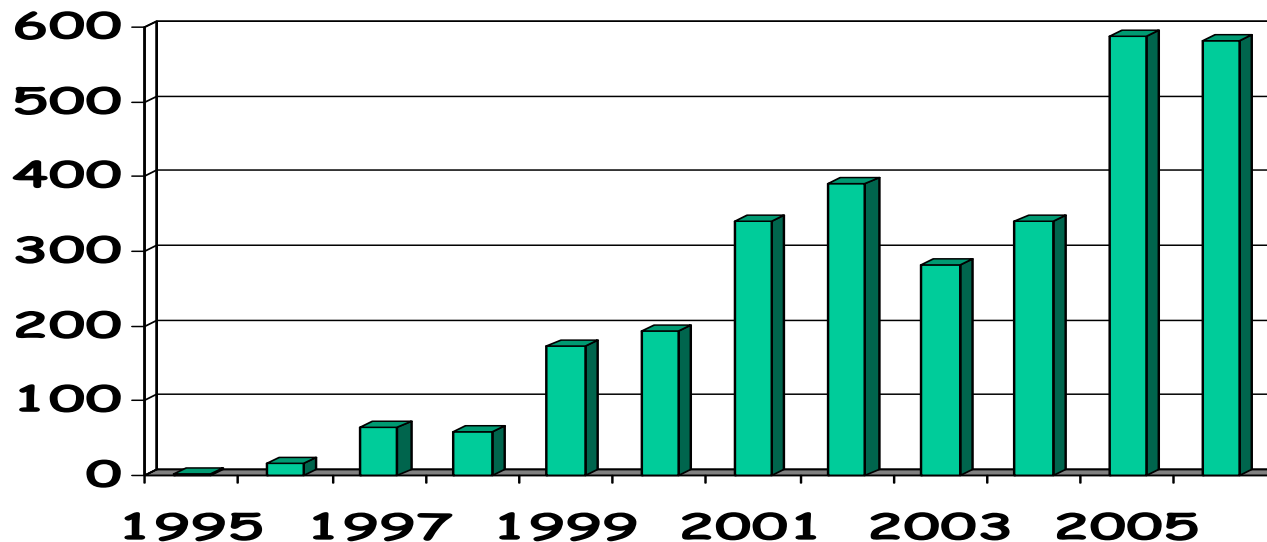
On the left side of the browser window, there is a small image of two bucket hats, one white and one black, both with the word "hacker" written on them.

# Control hijacking attacks

- Attacker's goal:
  - Take over target machine, e.g. web server
    - Execute arbitrary attack code on target by hijacking application control flow
- This lecture: three examples.
  - Buffer overflow attacks
  - Integer overflow attacks
  - Format string vulnerabilities

# 1. Buffer overflows

- Extremely common bug.
  - First major exploit: 1988 Internet Worm(fingerd)

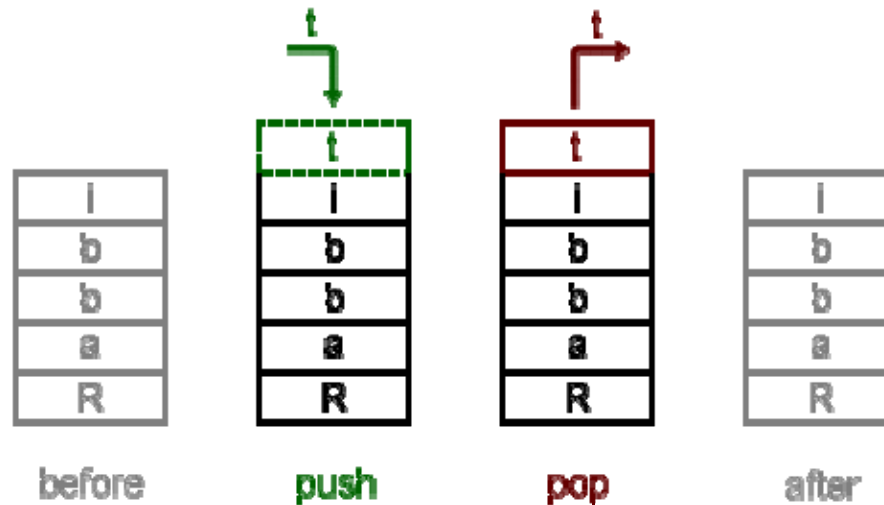


≈20% of all vuln.

2005-2007: ≈ 10%

Source: NVD/CVE

# Stack ADT



- A *stack* supports two operations
  - *push*: adds object to top of stack
  - *pop*: removes object from top of stack

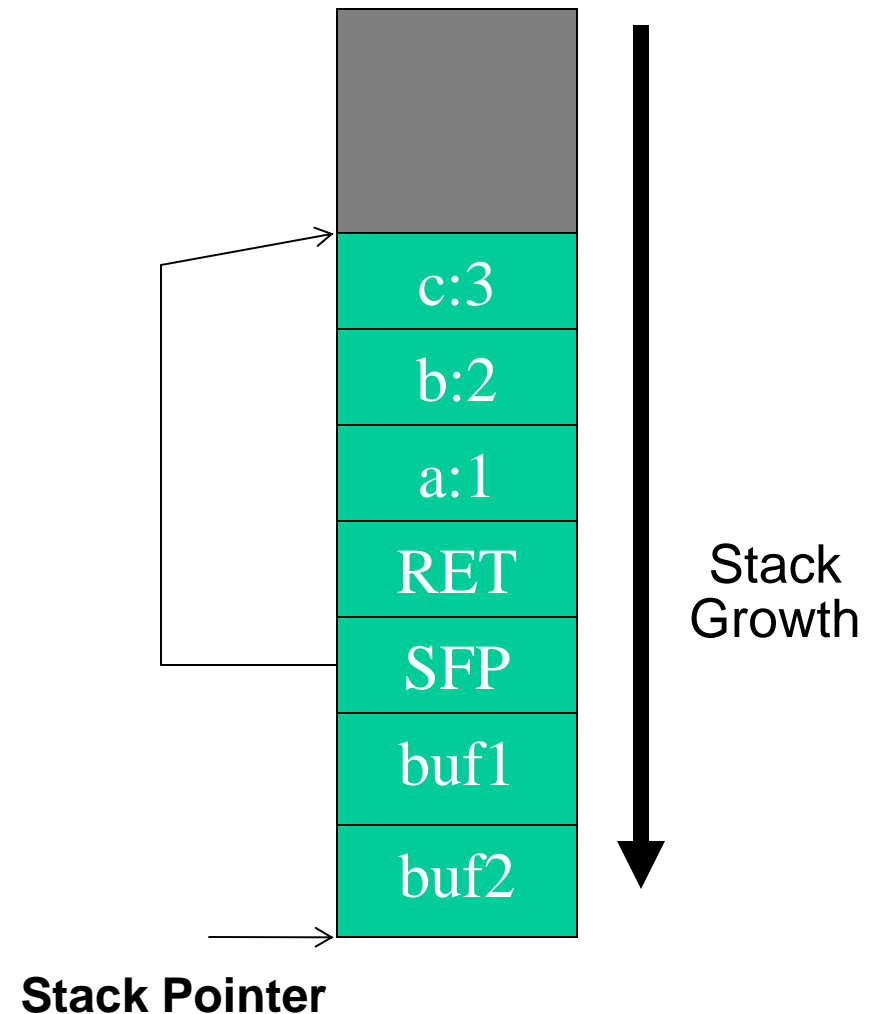
# Stack of Function Activ. Recs.

```
void main()  
{ function(1,2,3);  
}
```

```
void function(int a, int b, int c)  
{ char buffer1[5];  
  char buffer2[10];  
}
```

Function call: *push*

- Parameters, current IP as return address, saved frame pointer, local vars



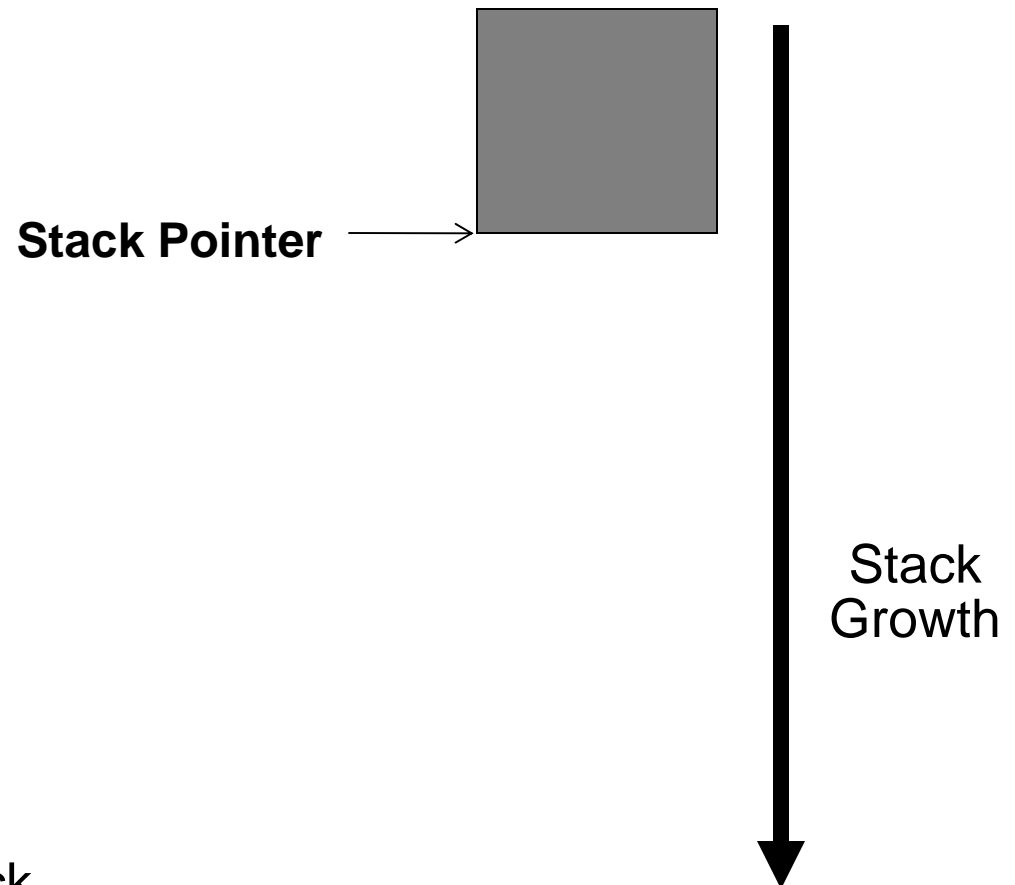
# Stack of Function Activ. Recs.

```
void main()  
{ function(1,2,3);  
}
```

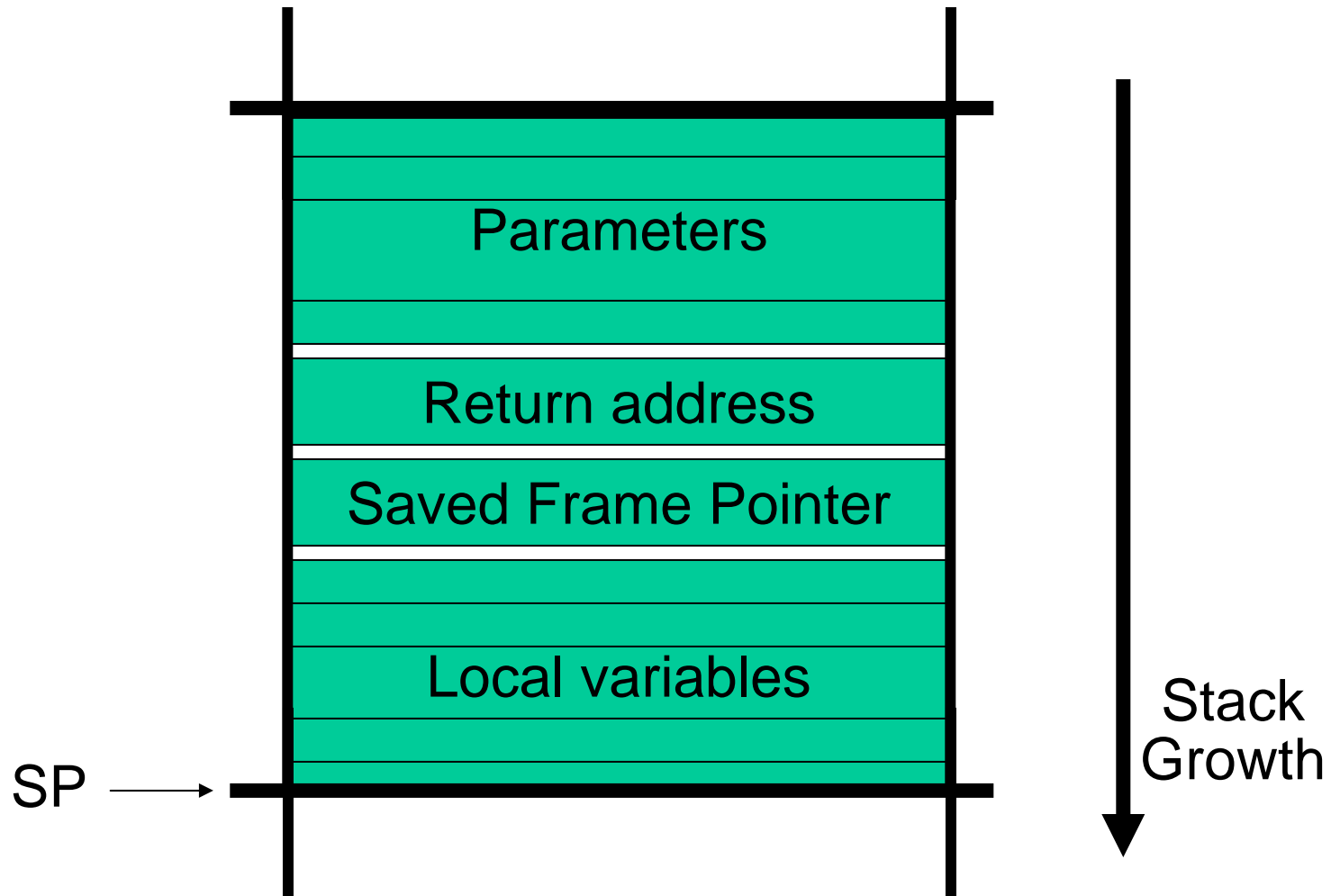
```
void function(int a, int b, int c)  
{ char buffer1[5];  
  char buffer2[10];  
}
```

Function return: *pop*

- Activation record, update stack pointer using SFP, update IP using RET



# Stack Frame



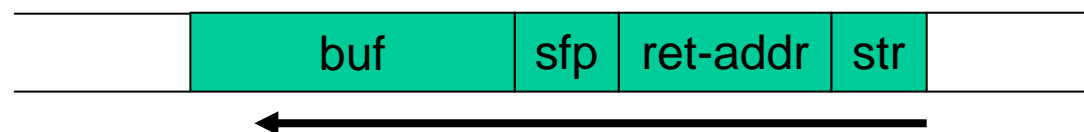
# What are buffer overflows?

- Suppose a web server contains a function:

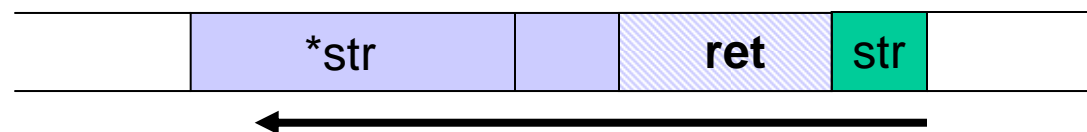
```
void func(char *str) {
    char buf[128];

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

- When the function is invoked the stack looks like:

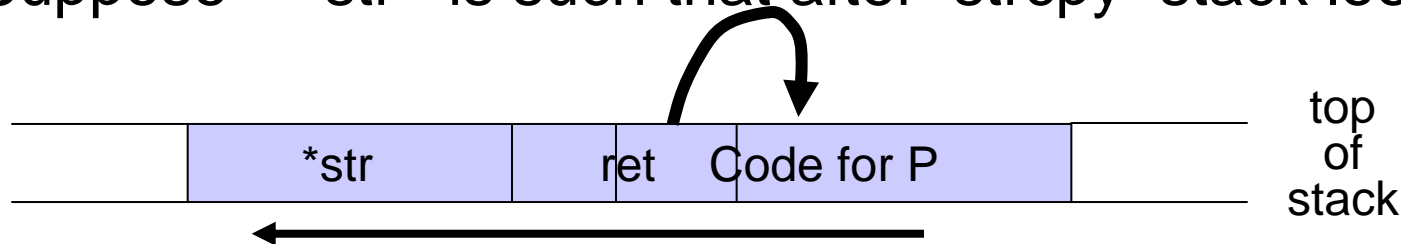


- What if `*str` is 136 bytes long? After `strcpy`:



# Basic stack exploit

- Problem: no range checking in `strcpy()`.
- Suppose `*str` is such that after `strcpy` stack looks like:



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

(exact shell code by Aleph One)

- When `func()` exits, the user will be given a shell !
- Note: attack code runs *in stack*.
- To determine `ret` guess position of stack when `func()` is called

# Many unsafe C lib functions

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

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

gets (char \*s)

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

⋮

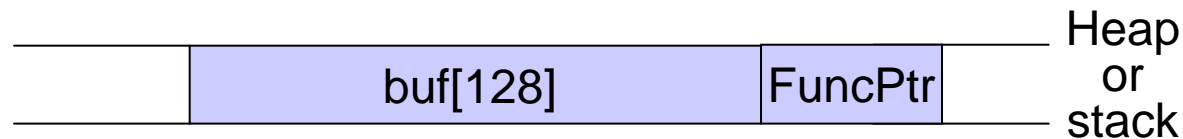
- “Safe” versions `strncpy()`, `strncat()` are misleading
  - `strncpy()` may leave buffer unterminated.
  - `strncpy()`, `strncat()` encourage off by 1 bugs (overwrite LSB of SFP).

# Exploiting buffer overflows

- Suppose web server calls `func()` with given URL.
  - Attacker sends a 200 byte URL. Gets shell on web server
- Some complications:
  - Program `P` should not contain the `'\0'` character.
  - Overflow should not crash program before `func()` exits.
- Sample remote buffer overflows of this type:
  - (2005) Overflow in MIME type field in MS Outlook.
  - (2005) Overflow in Symantec Virus Detection

# Control hijacking options

- Stack smashing attack:
  - Override return address in stack activation record by overflowing a local buffer variable.
- 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.

# Other types of overflow attacks

- Double free: double free space on heap.
  - Can cause mem mgr to write data to specific location
  - Examples: CVS server
- Integer overflows: (e.g. MS DirectX MIDI Lib) Phrack60
  - Integers are represented using a fixed number of bits
  - Wrap around (modulo max value + 1) if value greater than max

# Integer overflow (1)

```
int catvars(char *buf1, char *buf2, unsigned int len1, unsigned int len2)
{
    char mybuf[256];
    if((len1 + len2) > 256)
        { /* [3] */ return -1; }
    memcpy(mybuf, buf1, len1); /* [4] */
    memcpy(mybuf + len1, buf2, len2);
    do_some_stuff(mybuf); return 0;
}
```

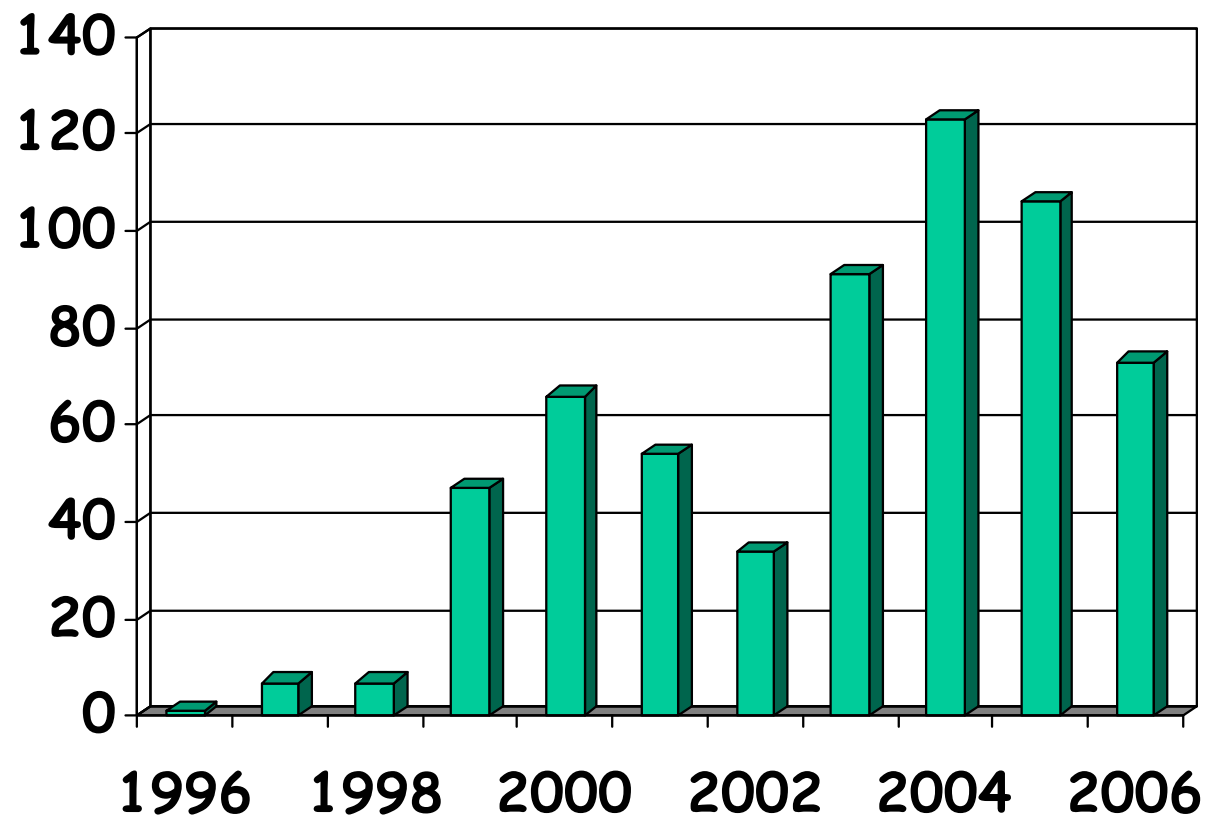
- **Problem:** If  $\text{len1} = 0x104$  and  $\text{len2} = 0xfffffc$ , then  
 $\text{len1} + \text{len2} = 0x100$  (decimal 256),  
which implies buffer overflow attack!

# Integer overflow (2)

```
int copy_something(char *buf, int len)
{
    char kbuf[800];
    if(len > sizeof(kbuf))
    { /* [1] */ return -1; }
    return memcpy(kbuf, buf, len); /* [2] */
}
```

- **Problem:**
  - memcpy takes an unsigned int as the len parameter
  - Bounds check performed before the memcpy is done using signed integers
  - Pass negative value for len

# Integer overflow stats



Source: NVD/CVE

# Summary: designing buffer overflow exploits


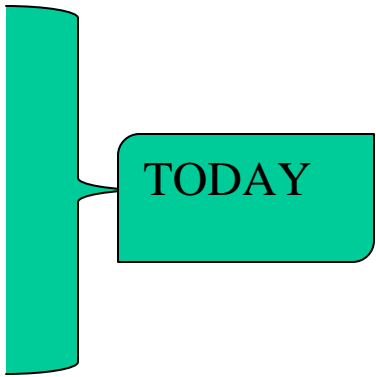
- Understanding C functions and the stack
- Some familiarity with machine code
- Know how system calls are made
- The `exec()` system call

• *Read Phrack paper by Aleph One*  
• *More in TA section, project 1*

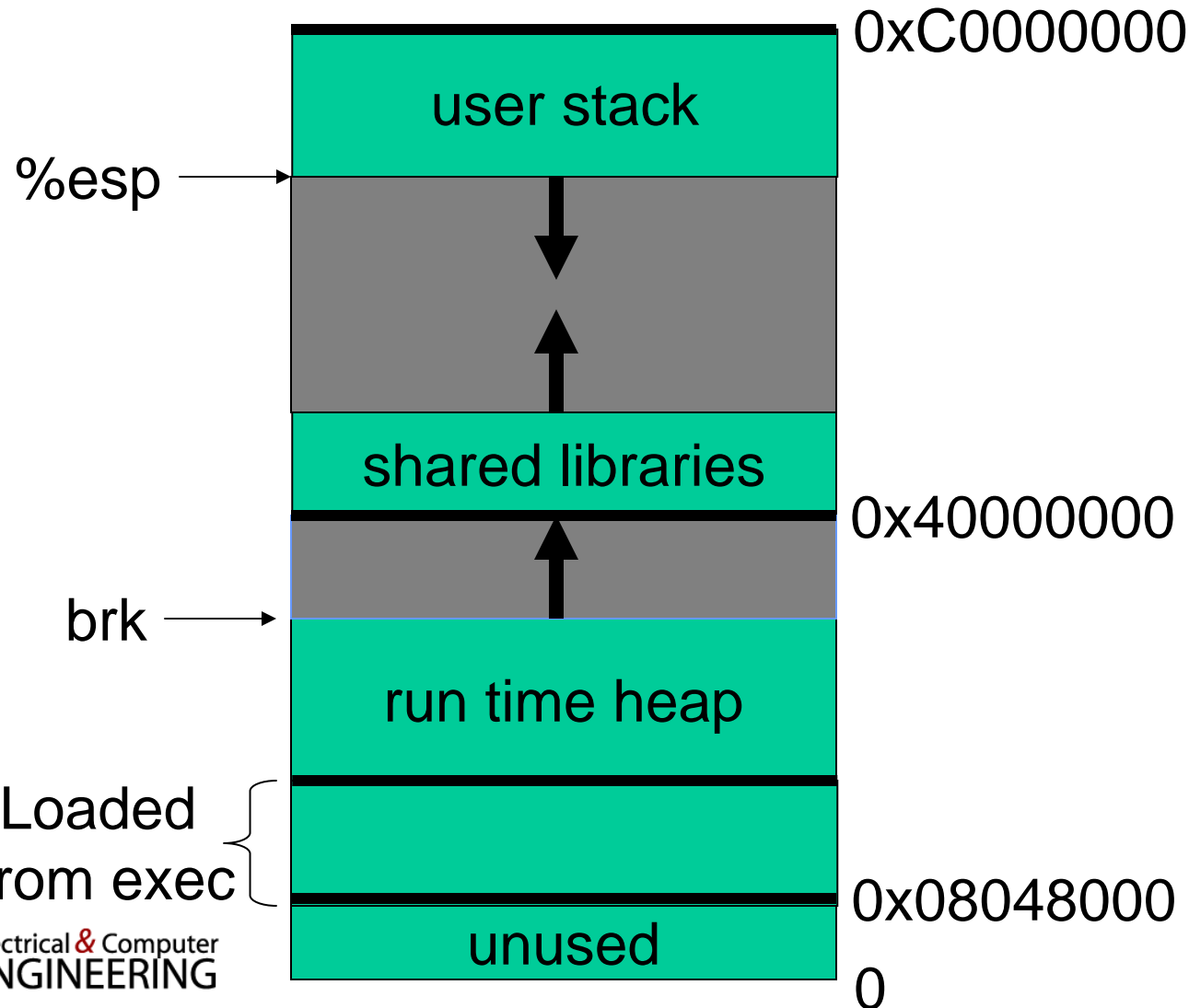
- 
- Attacker needs to know which CPU and OS are running on the target machine:
    - Our examples are for x86 running Linux
    - Details vary slightly between CPUs and OSs:
      - Little endian vs. big endian (**x86 vs. Motorola**)
      - Stack Frame structure (Linux vs. Windows)
      - Stack growth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

# Defenses

# Preventing hijacking attacks

1. Fix bugs: 
  - Audit software
    - Automated tools: Coverity, Prefast/Prefix.
  - Rewrite software in a type safe language (Java, ML)
    - C does not protect language abstractions
    - Rewriting difficult for existing (legacy) code ...
2. Concede overflow, but prevent code execution
3. Add runtime code to detect overflows exploits 
  - Halt process when overflow exploit detected
  - StackGuard, LibSafe, ...

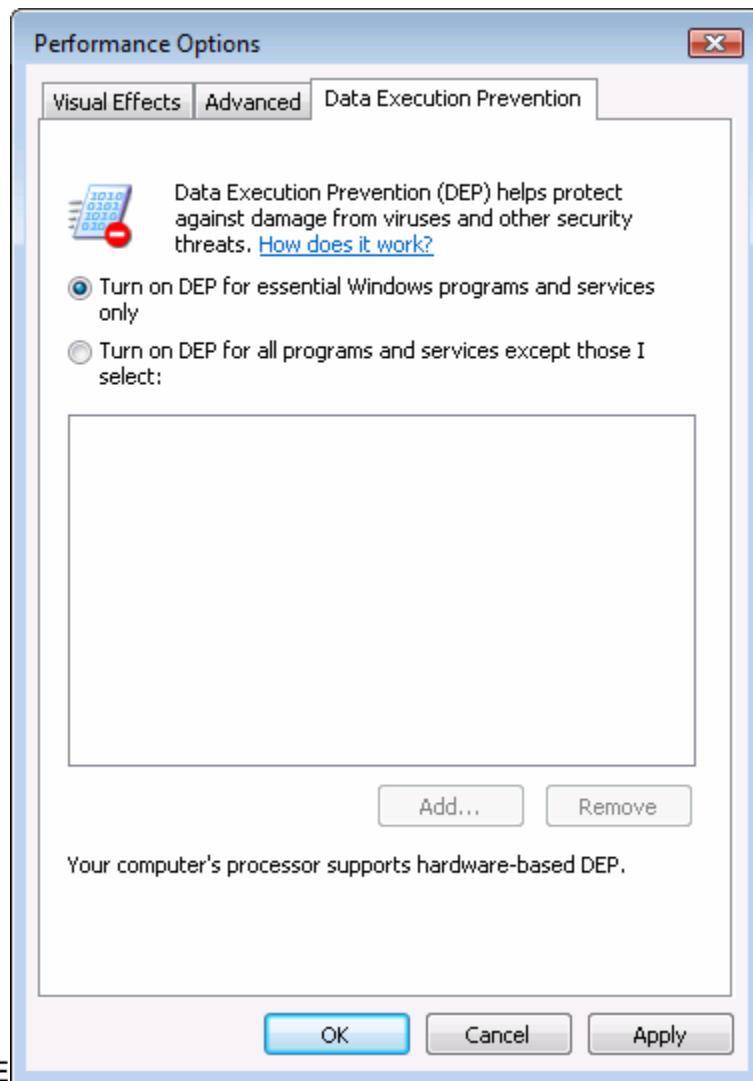
# Linux process memory layout



# Marking memory as non-execute (W^X)

- Prevent overflow code execution by marking stack and heap segments as **non-executable**
  - NX-bit on AMD Athlon 64, XD-bit on Intel P4 Prescott
    - NX bit in every Page Table Entry (PTE)
  - Deployment:
    - Linux (via PaX project); OpenBSD
    - Windows since XP SP2 (DEP)
- Limitations:
  - Some apps need executable heap (e.g. JITs).
  - Does not defend against **return-to-libc** exploit

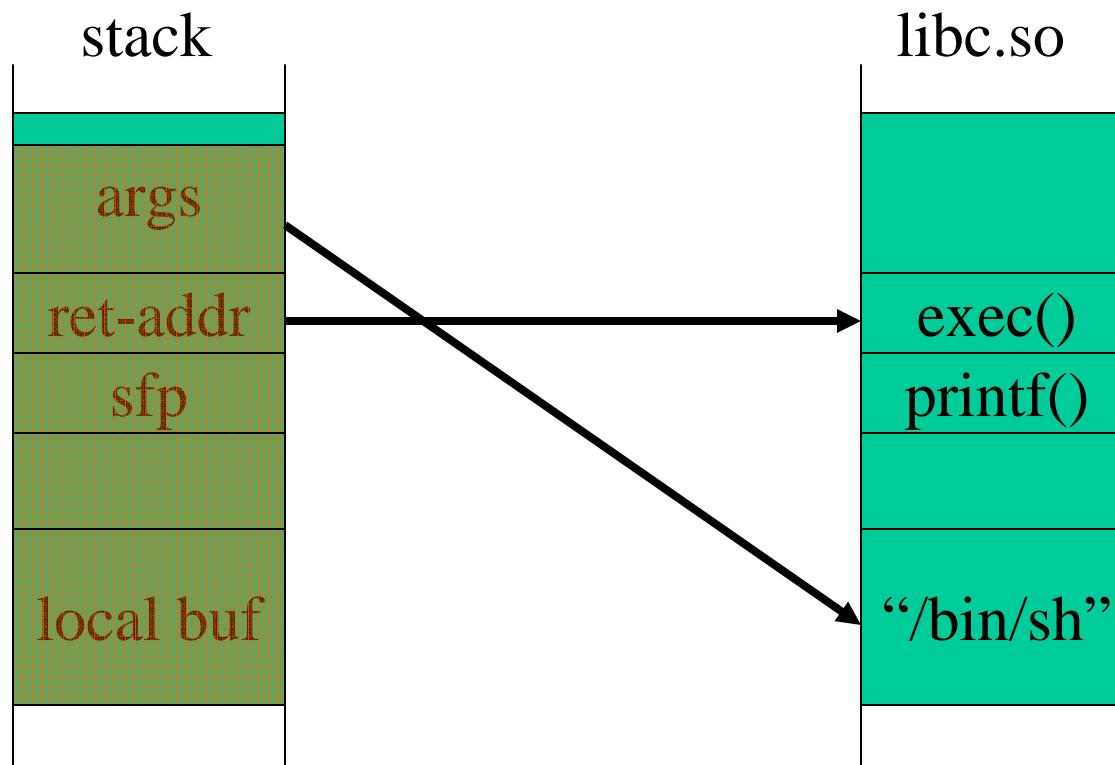
# Examples: DEP controls in Vista



DEP terminating a program

# Return to libc

- Control hijacking without code injection



# Response: randomization

- **ASLR**: (Address Space Layout Randomization)
  - Map shared libraries to rand location in process memory
    - ⇒ Attacker cannot jump directly to exec function
  - Deployment:
    - Windows Vista: 8 bits of randomness for DLLs
    - Linux (via PaX): 16 bits of randomness for libraries
  - More effective on 64-bit architectures
- **Other randomization methods**:
  - Sys-call randomization: randomize sys-call id's
  - Instruction Set Randomization (ISR)

# ASLR Example

Booting Vista twice loads libraries into different locations:

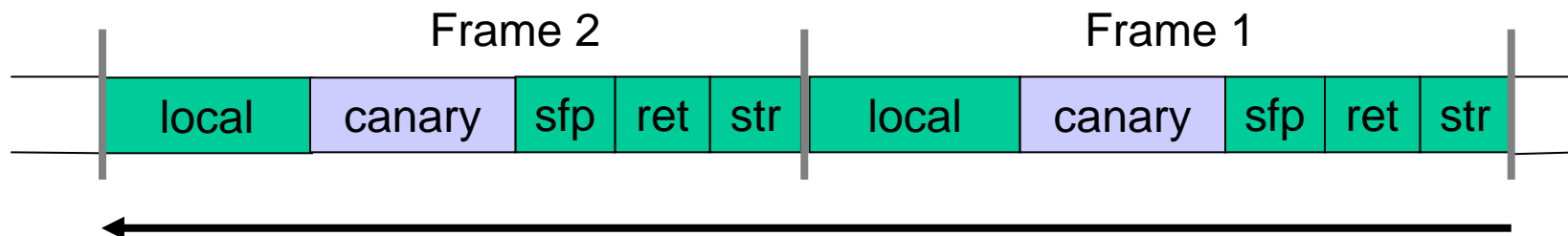
ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

# Run time checking

# Run time checking: StackGuard

- Many many run-time checking techniques ...
  - we only discuss methods relevant to overflow protection
- Solution 1: StackGuard
  - Run time tests for stack integrity.
  - Embed “canaries” in stack frames and verify their integrity prior to function return.



# Canary Types

- Random canary:
  - Choose random string at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
  - To corrupt random canary, attacker must learn current random string.
- Terminator canary:

Canary = 0, newline, linefeed, EOF

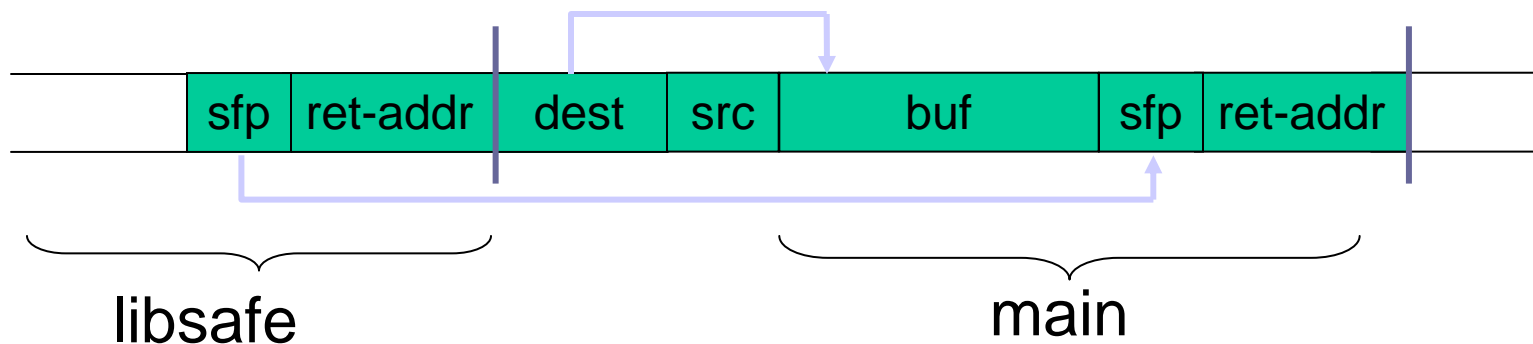
  - String functions will not copy beyond terminator.
  - Attacker cannot use string functions to corrupt stack.

# StackGuard (Cont.)

- StackGuard implemented as a GCC patch.
  - Program must be recompiled.
- Minimal performance effects: 8% for Apache.
- Heap protection: PointGuard.
  - Protects function pointers and setjmp buffers by encrypting them: XOR with random cookie
  - More noticeable performance effects

# Run time checking: Libsafe

- Solution 2: Libsafe (Avaya Labs)
  - Dynamically loaded library.
  - Intercepts calls to `strcpy (dest, src)`
    - Validates sufficient space in current stack frame
    - If so, does `strcpy`, otherwise, terminates application



# More methods ...

- StackShield
  - At function prologue, copy return address RET and SFP to “safe” location (beginning of data segment)
  - Upon return, check that RET and SFP is equal to copy.
  - Implemented as assembler file processor (GCC)

# Format string bugs

# 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

Let's try to understand the anatomy of this class of vulnerabilities

# Format function execution

```
printf ("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

From within the printf function the stack looks like this:

```
stack top  
...  
<&a>  
<a>  
<i>  
A  
...  
stack bottom
```

where:

A - address of the format string

i - value of the variable i

a - value of the variable a

&a address of the variable a

Format function parses the format string 'A', by reading a character at a time

- If not '%', the character is copied to the output
- If %, the character behind the '%' specifies the type of parameter that should be evaluated; the parameter is located on the stack

# Crashing a program: DoS attack

```
int func(char *user) {  
    fprintf( stdout, user);}
```

Problem: what if user = "%s%s%s%s%s%s%s" ??

- Most likely program will crash: Denial of Service
- Why?
  - %s will try to display memory from an address supplied on stack; very likely this is an illegal address, which is not mapped

Correct form:

```
int func(char *user) {  
    fprintf( stdout, "%s", user);}
```

# Viewing the stack

```
... printf(user) ...
```

Problem: what if `user = %08x.%08x.%08x.%08x\n`??

Retrieve 4 parameters from stack and display them  
as 8-digit padded hexadecimal numbers

# Buffer overflow using format string

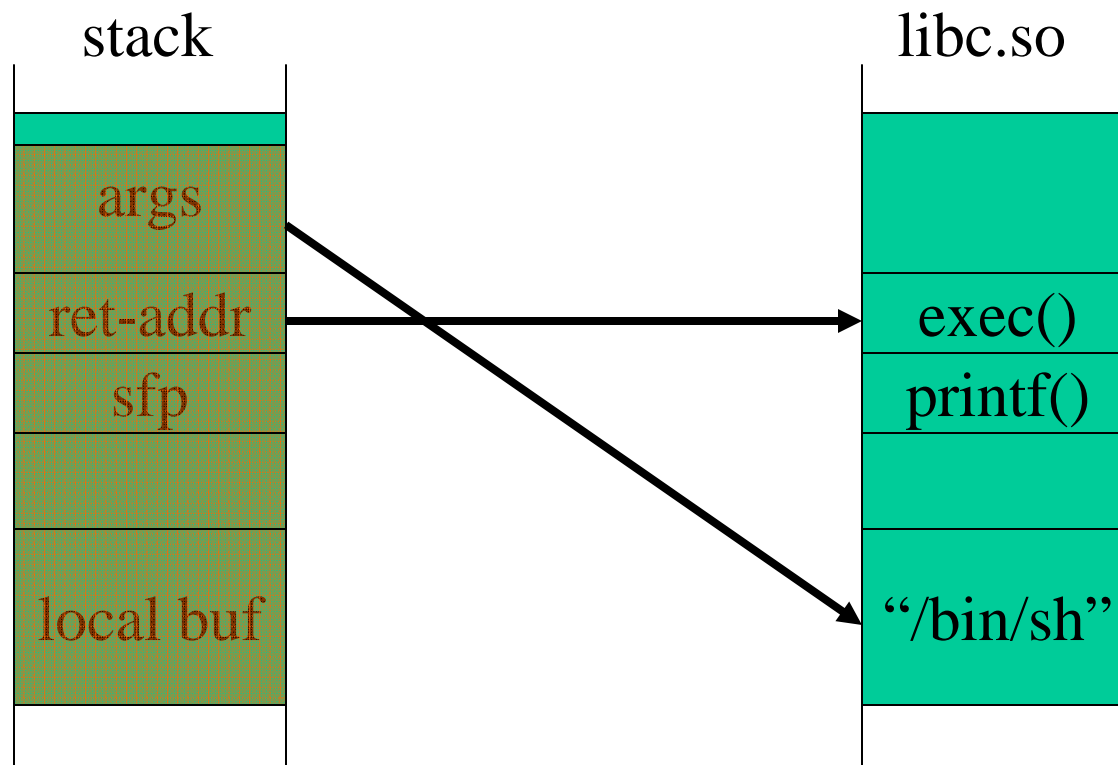
```
char errmsg[512], outbuf[512];  
sprintf (errmsg, "Illegal command: %400s", user);  
sprintf( outbuf, err...msg );
```

- What if user = "%497d \x3c\xd3\xff\xbf <nops> <shellcode>"
  - Bypass "%400s" limitation.
  - Will overflow outbuf; now launch regular stack smashing buffer overflow attack

# A new exploit: Return-oriented Programming

# Recall return to libc

- Control hijacking without code injection



Perception:

- Attacker cannot execute arbitrary code

- Attacker relies on contents of libc — remove `system()`?

# The Return-oriented programming thesis

any sufficiently large program codebase

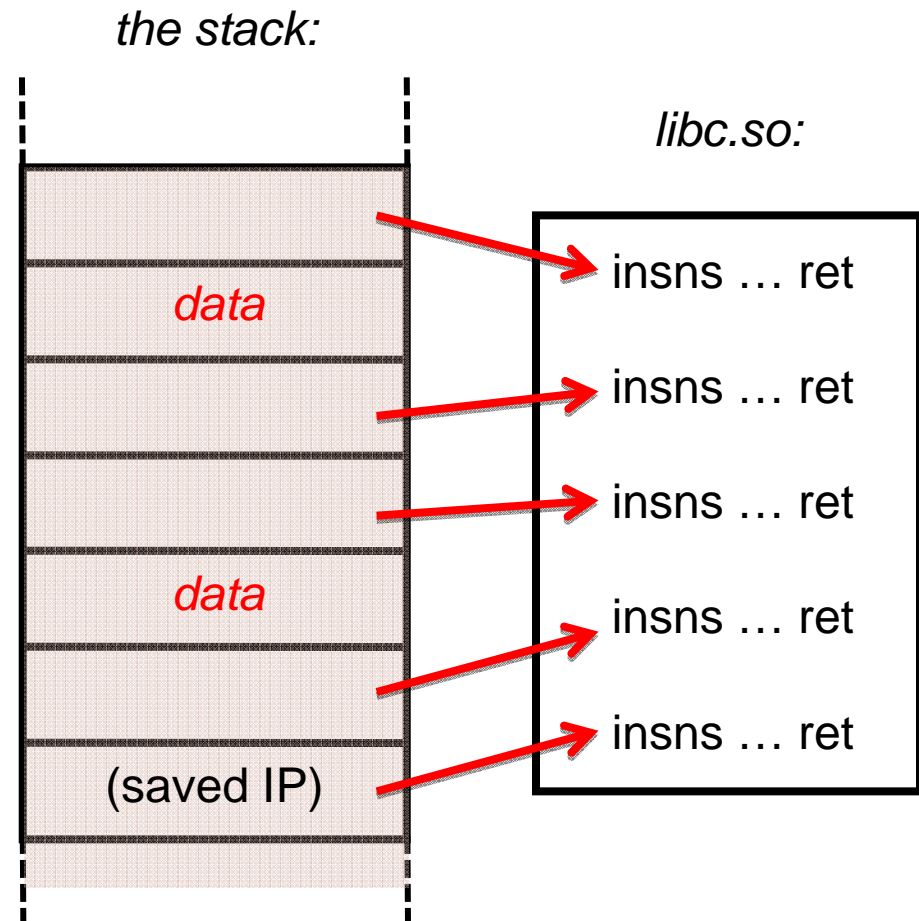


arbitrary attacker computation and behavior,  
*without* code injection

(Defense: control-flow integrity)

# Return-oriented programming

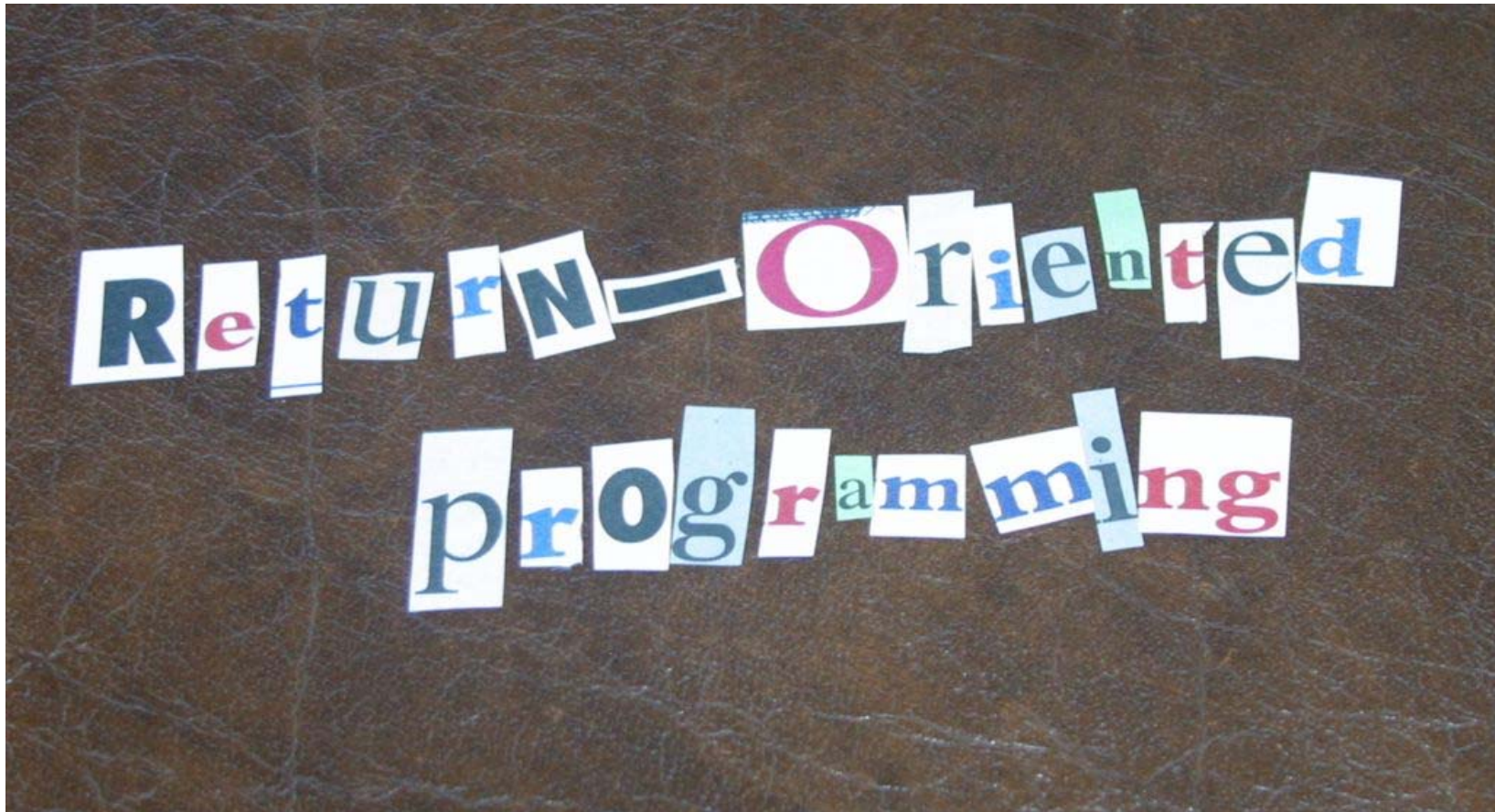
- Treat libc as a corpus of instruction sequences each ending in “return” insn
- Fill stack with pointers to sequences (and with data)
- Execution flows through sequences, induces desired behavior (Turing complete)
- Falsifies perception of return-into-libc as limited, easy to defeat



# Like rearranging magazine headlines ...



... to compose a ransom note.



# Questions?

# Sources

- Cowan et al., [Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade](#), Foundations of Intrusion Tolerant Systems 2003.
- blexim, [Basic Integer Overflows](#), Phrack 60, 2002
- team teso, [Exploiting Format String Vulnerabilities](#), 2001
  
- Many of the slides are from D. Boneh and J. Mitchell's Stanford CS 155 lecture on this topic
- Return-Oriented Programming slides based on slides from H. Shacham

