

# Computer Security

## CS 426

### Lecture 4



## Software Vulnerabilities

# Readings for This Lecture

- **Counter Hack Reloaded**
  - **Chapter 7: Gaining Access Using Application and Operating System Attacks**
- **Smashing The Stack For Fun And Profit by Aleph One** (<http://doc.bughunter.net/buffer-overflow/smash-stack.html>)
- [http://en.wikipedia.org/wiki/Buffer\\_overflow](http://en.wikipedia.org/wiki/Buffer_overflow)

# Steps in a standard break-in (Getting in)

- Get your foot in the door
  - Steal a password file and run dictionary attack
  - Sniff passwords off the network, social engineering
  - Use input vulnerability in network-facing programs (e.g., web server, ftp server, mail server, browser, etc.)
- Use partial access to gain root (admin) access
  - Break some mechanism on the system
  - Often involve exploiting vulnerabilities in some local programs

# Steps in a standard break-in (Do )

- Set up some way to return
  - Install login program or web server with back door
- Cover your tracks
  - Disable intrusion detection, virus protection, system functions that show list of running programs, ...
- Perform desired attacks
  - break into other machines
  - taking over the machine
  - ...

# Common Software Vulnerabilities

- Input validation
- Buffer overflows
- Format string problems
- Integer overflows
- Failing to handle errors
- ...

# Input Validation

- Attackers have many ways to control input into a system
- What are some things that the attacker may try to achieve?
  - Crash programs
  - Execute arbitrary code
    - setuid or setgid programs
  - Obtain sensitive information

# Weak Input Validation

- Lots of programs have input
  - Command line arguments
  - Environment variables
  - Function calls from other modules
  - Configuration files
  - Network packets
- Many web site examples
  - Web form input
  - Scripting languages with string input

# Command line

- User can set command line arguments to almost anything
  - Using `execve` command
  - Do not trust name of the program (it can be sent to any value including NULL)
- Do not check for bad things (blacklisting)
- Check for things that are allowed (whitelisting)
- Check all possible inputs

# Simple example

```
void main(int argc, char ** argv) {  
    char buf[1024];  
    sprintf(buf, "cat %s", argv[1]);  
    system ("buf");  
}
```

Can easily add things to the command  
by adding ;

# Environment variables

- Users can set the environment variables to anything
  - Using `execve`
  - Has some interesting consequences
- Examples:
  - `LD_LIBRARY_PATH`
  - `PATH`
  - `IFS`

# An example attack

- Assume you have a setuid program that loads dynamic libraries
- UNIX searches the environment variable `LD_LIBRARY_PATH` for libraries
- A user can set `LD_LIBRARY_PATH` to `/tmp/attack` and places his own copy of the libraries here
- Most modern C runtime libraries have fixed this by not using the `LD_LIBRARY_PATH` variable when the `EUID` is not the same as the `UID` or the `EGID` is not the same as the `GID`

# More fun with environment variables

- A setuid program has a system call: `system(ls);`
- The user sets his PATH to be `.` and places a program `ls` in this directory
- The user can then execute arbitrary code as the setuid program
- Solution: Reset the PATH variable to be a standard form (i.e., `"/bin:/usr/bin"`)

# Even more fun

- You should not add `.` into the `PATH` variable unless it is necessary
  - If you must put it at the end
- However, you must also reset the `IFS` variable
  - `IFS` is the characters that the system considers as white space
- If not add “s” to the `IFS`
  - `system(ls)` becomes `system(l)`
  - Place a function `l` in the directory

# A Remote Example: PHP passthru

- Idea
  - PHP `passthru(string)` executes command
  - Pages can construct *string* from user input
  - Put “;” in user input to run your favorite command
    - Morris Internet worm did something similar using “|”
- Example
  - `passthru(“find . -print | xargs cat | grep $test”);`
- User input `; ls /`  
Runs `find . -print | xargs cat | grep ; ls /`

# Checking input can be tricky: Unicode vulnerabilities

- Some web servers check string input
  - Disallow sequences such as ../ or \
  - But may not check unicode %c0%af for '/'
- IIS Example, used by Nimda worm

```
http://victim.com/scripts/../../winnt/system32/cmd.exe?<some command>
```

- passes <some command> to cmd command
- scripts directory of IIS has execute permissions
- Input checking would prevent that, but not this

```
http://victim.com/scripts/..%c0%af..%c0%afwinnt/system32/...
```

- IIS first checks input, then expands unicode

see [www.sans.org/rr/threats/unicode.php](http://www.sans.org/rr/threats/unicode.php)

Fall 2007/Lecture 4

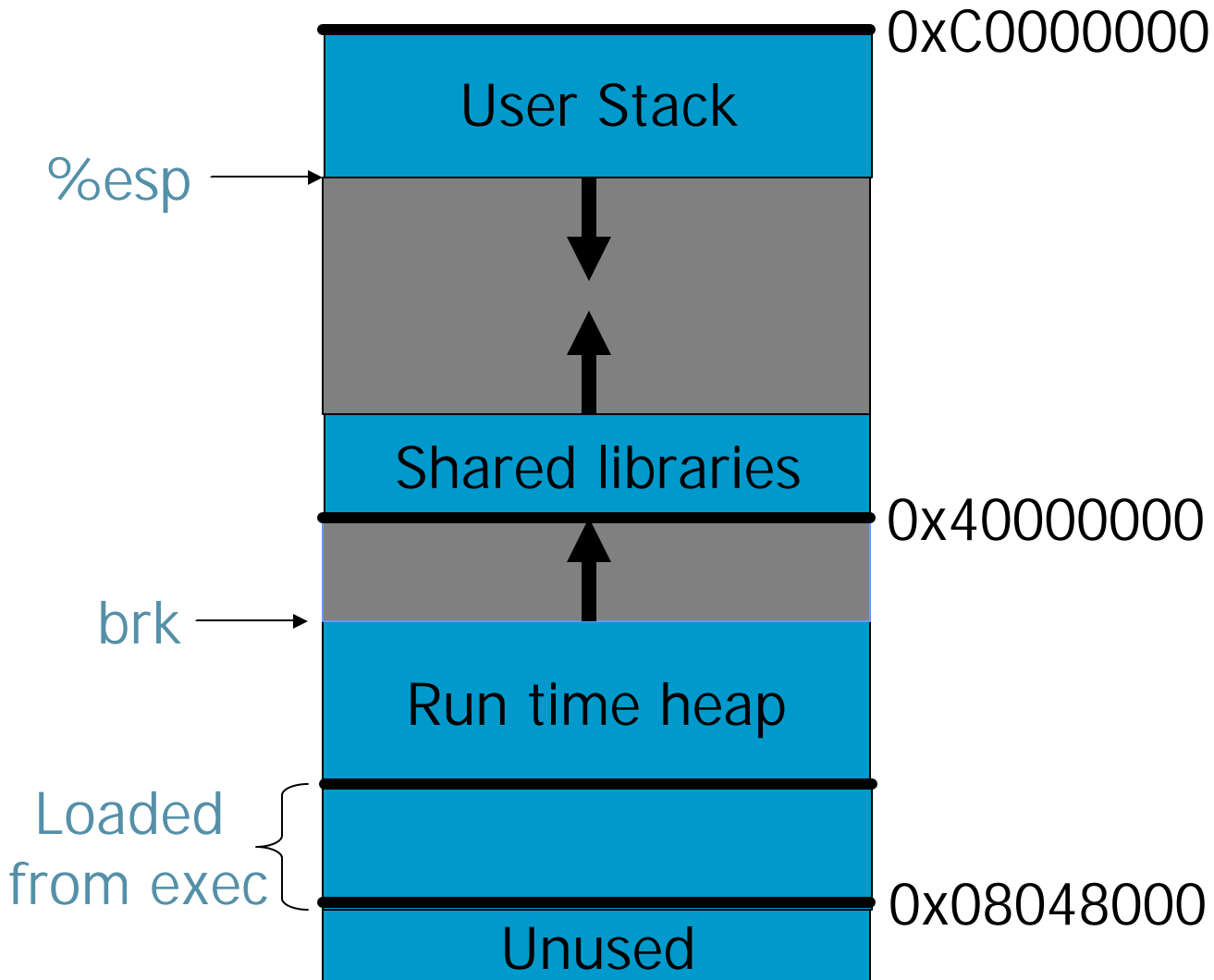
# Buffer overflows

- Extremely common bug.
  - First major exploit: 1988 Internet Worm. fingerd.
- 15 years later:  $\approx 50\%$  of all CERT advisories:
  - 1998: 9 out of 13
  - 2001: 14 out of 37
  - 2003: 13 out of 28
- Often leads to total compromise of host.
- Developing buffer overflow attacks:
  - Locate buffer overflow within an application.
  - Design an exploit.

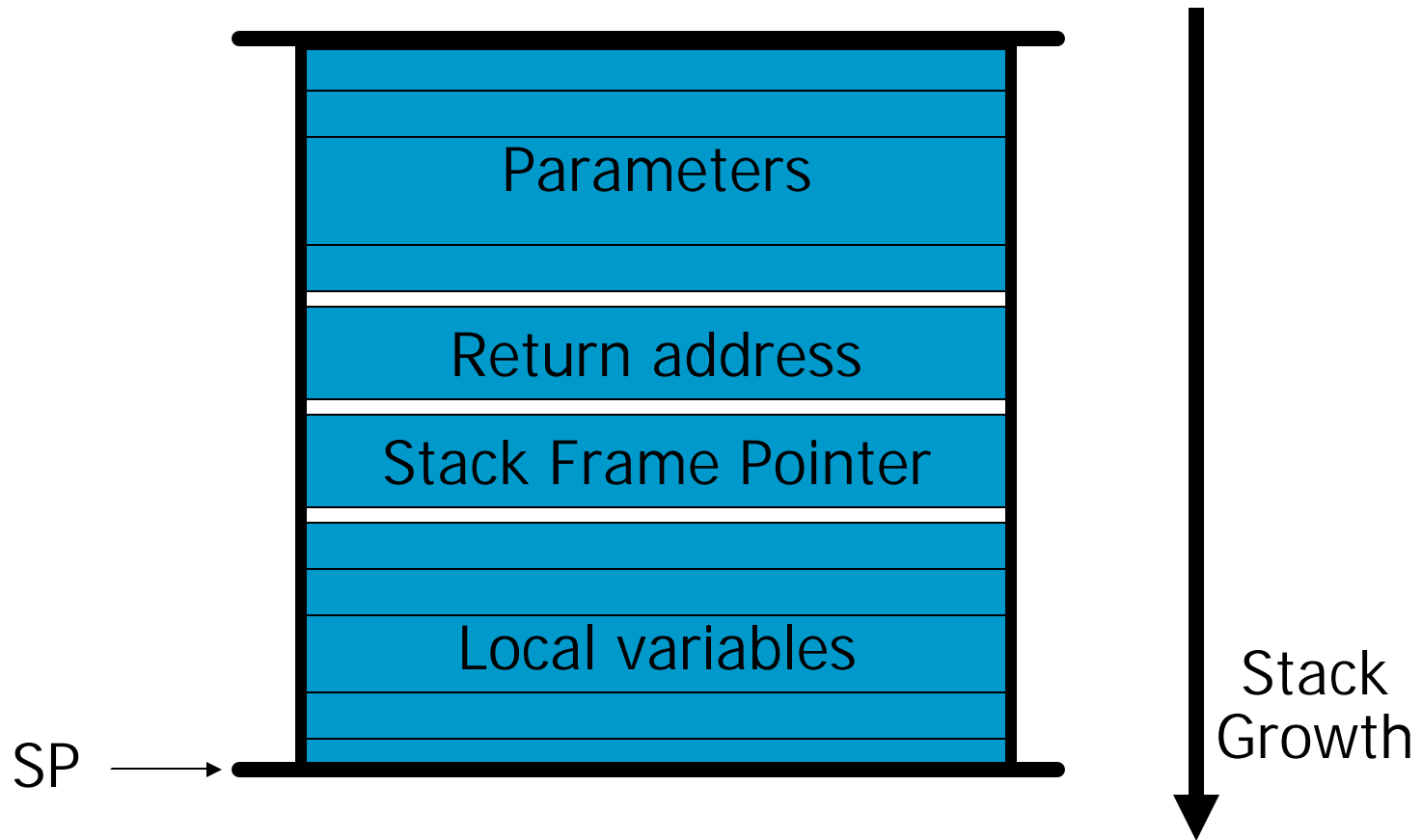
# What is needed

- Understanding C functions and the stack.
- Some familiarity with machine code.
- Know how systems calls are made.
- The `exec()` system call.
  
- 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 CPU's and OS:
    - Stack growth direction.
    - big endian vs. little endian.

# Linux process memory layout



# Stack Frame

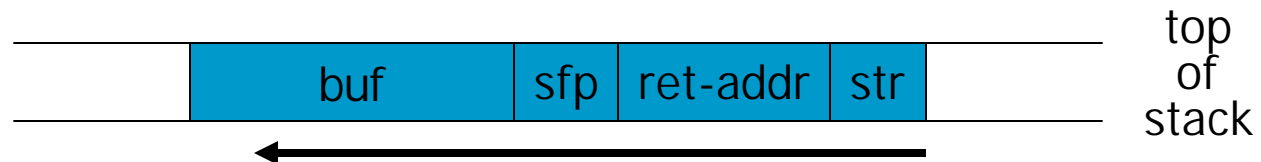


# What are buffer overflows?

- 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:

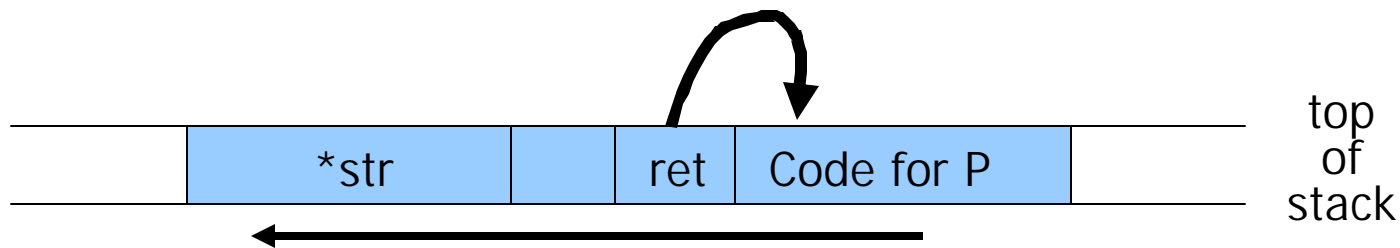


- What if `*str` is 136 bytes long? After `strcpy`:



# Basic stack exploit

- Main 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.

# Some unsafe C lib functions

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

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

gets (char \*s)

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

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

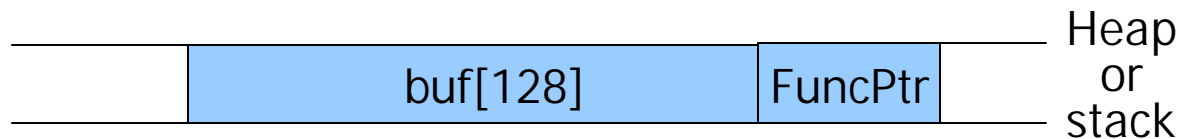
⋮

# Exploiting buffer overflows

- Suppose web server calls `func()` with given URL.
- Attacker can create a 200 byte URL to obtain shell on web server.
- Some complications:
  - Program `P` should not contain the `'\0'` character.
  - Overflow should not crash program before `func()` exits.

# Control hijacking opportunities

- Stack smashing attack:
  - Override return address in stack activation record by overflowing a local buffer variable.
- Function pointers: (used in attack on PHP 4.0.2)



- Overflowing buf will override function pointer.
- Longjmp buffers: `longjmp(pos)` (used in attack on Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

# Finding buffer overflows

- Hackers find buffer overflows as follows:
  - Run web server on local machine.
  - Issue requests with long tags.
    - All long tags end with “\$\$\$\$\$”.
  - If web server crashes,
    - search core dump for “\$\$\$\$\$” to find overflow location.
- Some automated tools exist. (eEye Retina, ISIC).
- Then use disassemblers and debuggers (e..g IDA-Pro) to construct exploit.

# Buffer overflow

- Imagine simple password-checking code

```
passwd() { ...  
  int funct(char *inp) {  
    char buf[10];  
    strcpy(buf,inp); }  
  ... }
```

- Function storage allocated on run-time stack
  - First return address (4 B)
  - Then locations for input parameter
  - Then space for buffer (10 chars)
- What if `strlen(inp) > 10` ?
  - Fill up buffer
  - Write over function parameter
  - Write over return address
  - “Return” will jump to location determined by input

Return addr
char *inp
buf[9]
buf[8]
...
buf[1]
buf[0]

# Some examples

- MSFT indexing service, an extension to IIS
  - telnet <site> 80
  - GET /somefile.idq?<long buffer>
  - Telnet to port 80 and send http GET with buffer over 240 bytes
  - Attacker can take over server
  - Form of attack used by Code Red to propagate
- TFTP server in Cisco IOS
  - Use overflow vulnerability to take over server (long filename)
- MS Xbox
  - James Bond 007 game has a save game option
  - Code to restore game has buffer overflow vulnerability
  - Can boot linux or run other code using game as "boot loader"

Many, many more examples

# Coming Attractions ...

- September 4:
  - Direct defense against buffer overflow attacks
  - Other software vulnerability problems

