

Information Security

CS 526

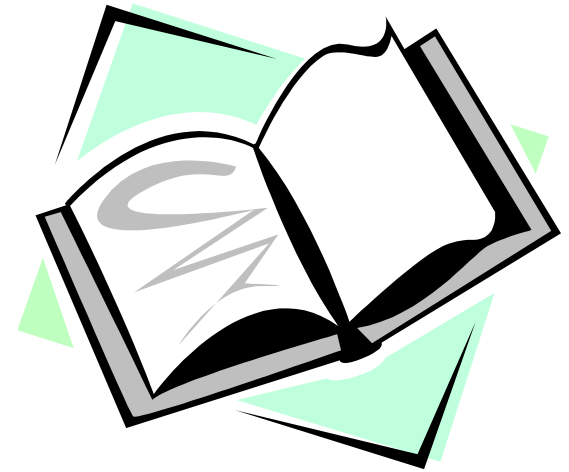
Topic 9



Software Vulnerabilities

Readings for This Lecture

- Wikipedia
 - **Privilege escalation**
 - **Directory traversal**
 - **Time-of-check-to-time-of-use**
 - **Buffer overflow**
 - **Stack buffer overflow**
 - **Buffer overflow protection**
 - **Format string attack**
 - **Integer overflow**
- [Smashing The Stack For Fun And Profit by Aleph One](#)



Why Software Vulnerabilities Matter?

- When a process reads input from attacker, the process may be exploited if it contains vulnerabilities.
- When an attacker successfully exploits a vulnerability, he can
 - Crash programs: Compromises availability
 - Execute arbitrary code: Compromises integrity
 - Obtain sensitive information: Compromises confidentiality
- Software vulnerability enables the attacker to run with privileges of other users, violating desired access control policy

Attacks Exploiting Software Vulnerabilities

- Drive-by download (drive-by installation)
 - malicious web contents exploit vulnerabilities in browsers (or plugins) to download/install malware on victim system.
- Email attachments in PDF, Word, etc.
- Network-facing daemon programs (such as http, ftp, mail servers, etc.) as entry points.
- Privilege escalation
 - Attacker on a system exploits vulnerability in a root process and gains root privilege

Common Software Vulnerabilities

- Input validation
- Race conditions
 - Time-of-check-to-time-of-use (TOCTTOU)
- Buffer overflows
- Format string problems
- Integer overflows

Sources of Input that Need Validation

- What are sources of input for local applications?
 - Command line arguments
 - Environment variables
 - Configuration files, other files
 - Inter-Process Communication call arguments
 - Network packets
- What are sources of input for web applications?
 - Web form input
 - Scripting languages with string input

Command line as a Source of Input: A Simple example

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

What can go wrong?

- Can easily add things to the command by adding ;, using e.g., "a; ls"
- User can set command line arguments to almost anything, e.g., by using execve system call to start a program, the invoker has complete control over all command line arguments.

Environment variables

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

Attack by Resetting PATH

- A setuid program has a system call: `system(ls);`
- The user sets his PATH to be `.` (current directory) 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"`)

Attack by Resetting IFS

- However, you must also reset the IFS variable
 - IFS is the characters that the system considers as white space
- If not, the user may add “s” to the IFS
 - `system(ls)` becomes `system(l)`
 - Place a function `l` in the directory
- Moral: things are intricately related and inputs can have unexpected consequences

Attack by Resetting LD_LIBRARY_PATH

- 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 RUID or the EGID is not the same as the RGID

A Web-Application Example: PHP passthru

- Idea
 - PHP `passthru(string)` executes command
 - Web-pages can construct *string* from user input and execute the commands to generate web content
 - Attackers can put “;” in input to run desired commands
- Example

```
echo 'Your usage log:<br />';
$username = $_GET['username'];
passthru("cat /logs/usage/$username");
```
- What if: “username=andrew;cat%20/etc/passwd”?

Directory Traversal Vulnerabilities in Web Applications

- A typical example of vulnerable application in php code is:

```
<?php
$template = 'red.php';
if ( isset( $_COOKIE['TEMPLATE'] ) )
$template = $_COOKIE['TEMPLATE'];
include ( "/home/users/phpguru/templates/" . $template );
?>
```

- Attacker sends

GET /vulnerable.php HTTP/1.0

Cookie: TEMPLATE=../../../../../../../../etc/passwd

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

Input Validation Summary

- Lessons:
 - Malicious inputs can become code, or change the logic to do things that are not intended
 - Inputs interact with each other, sometimes in subtle ways
- Use systematic approaches to deal with input validation
 - Avoid checking for bad things (blacklisting) if possible
 - The logic for blacklisting may not be exhaustive
 - Code where input is used may have different logic
 - Instead, check for things that are allowed (whitelisting)
 - Or, use systematic rewriting

Time-of-check-to-time-of-use

- **TOCTTOU**, pronounced "*TOCK too*"
- A class of software bug caused by changes in a system between the checking of a condition (such as authorization) and use of the results of the check.
 - When a process *P* requests to access resource *X*, the system checks whether *P* has right to access *X*; the usage of *X* happens later
 - When the usage occurs, perhaps *P* should not have access to *X* anymore.
 - The change may be because *P* changes or *X* changes.

An Example TOCTTOU

- In Unix, the following C code, when used in a setuid program, is a TOCTTOU bug:

```
if (access("file", W_OK) != 0)
    { exit(1); }
```

Attacker tries to execute the following line in another process when this process reaches exactly this time:
Symlink("/etc/passwd", "file")

```
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- Here, *access* is intended to check whether the real user who executed the setuid program would normally be allowed to write the file (i.e., *access* checks the real userid rather than effective userid).

TOCTTOU

- Exploiting a TOCTTOU vulnerabilities requires precise timing of the victim process.
 - Can run the attack multiple times, hoping to get lucky
- Most general attack may require “single-stepping” the victim, i.e., can schedule the attacker process after each operation in the victim
 - Techniques exist to “single-step” victim
- Preventing TOCTTOU attacks is difficult

What is Buffer Overflow?

- A **buffer overflow**, or **buffer overrun**, is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer.
- The result is that the extra data overwrites adjacent memory locations. The overwritten data may include other buffers, variables and program flow data, and may result in erratic program behavior, a memory access exception, program termination (a crash), incorrect results or — especially if deliberately caused by a malicious user — a possible breach of system security.
- Most common with C/C++ programs

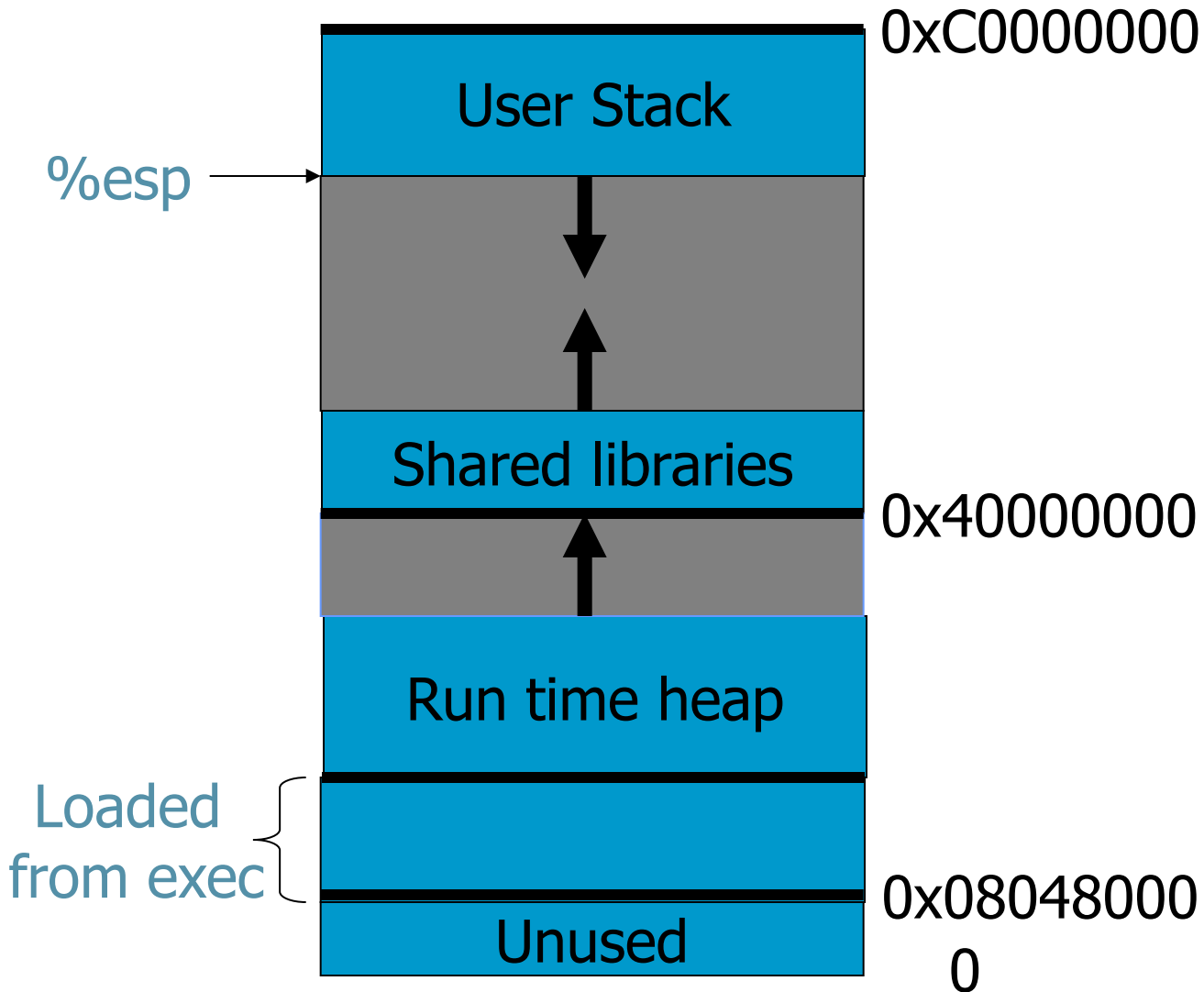
History

- Used in 1988's Morris Internet Worm
- Alphe One's "Smashing The Stack For Fun And Profit" in Phrack Issue 49 in 1996 popularizes stack buffer overflows
- Still extremely common today

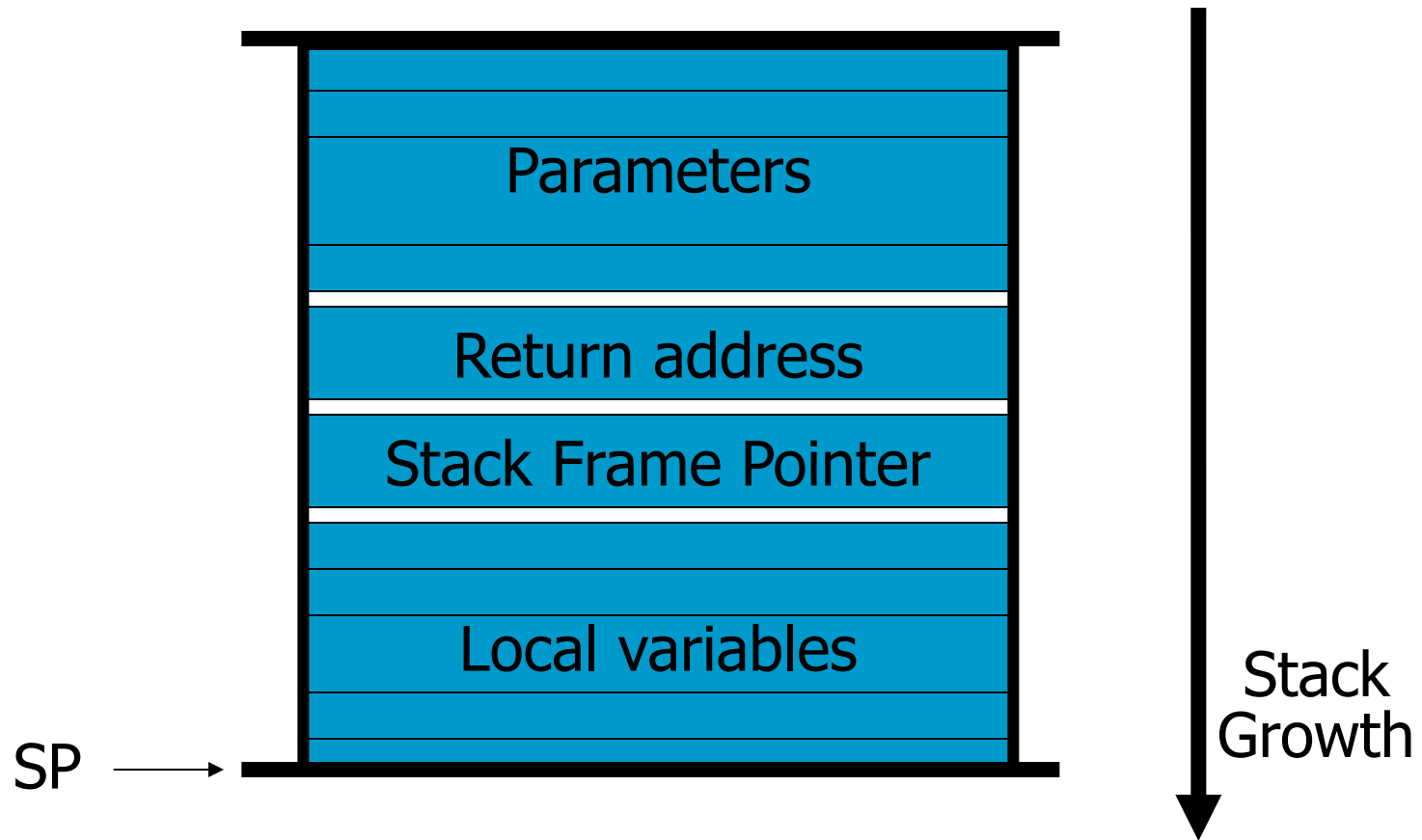
Types of Buffer Overflow Attacks

- Stack overflow
 - Shell code
 - Return-to-libc
 - Overflow sets ret-addr to address of libc function
 - Off-by-one
 - Overflow function pointers & longjmp buffers
- Heap overflow

Linux process memory layout



Stack Frame



What Happens in a Function Call?

```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    ...  
}  
  
int main() {  
    func("abc");  
}
```

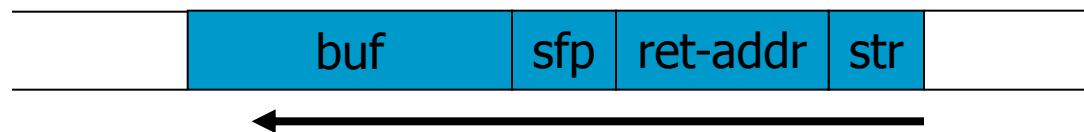
- Before main() calls func()
 - Push pointer to “abc” onto stack
 - Use “call func” assembly, which pushes current IP on stack
- Upon entering func()
 - Push stack frame pointer register (bp) on stack
 - Update sp to leave space for local variable.
- Upon leaving func()
 - Update sp to just below saved bp
 - Pop stack to bp, restore bp
 - Use “ret” assembly, which pop stack to IP

What are buffer overflows?

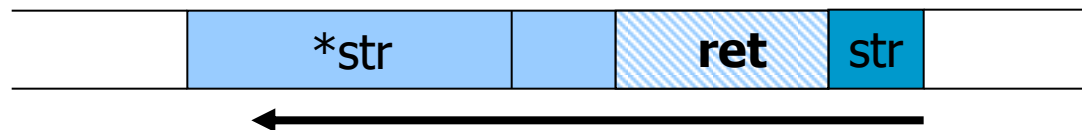
- Suppose a web server contains a function:

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

- When the function is invoked the stack looks like:

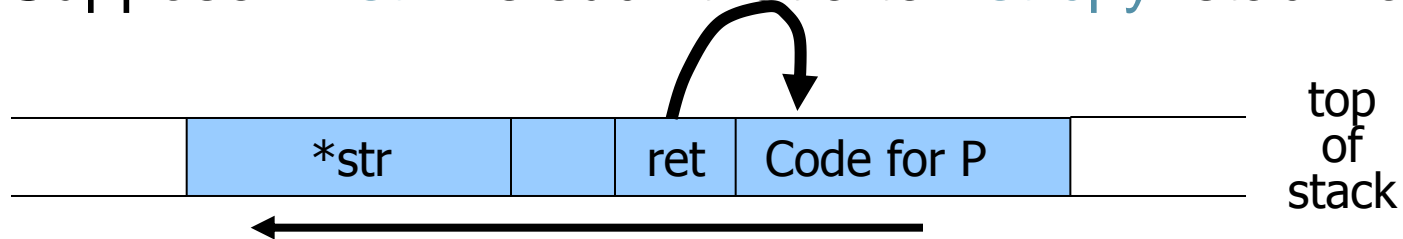


- What if `*str` is 136 bytes long? After `strcpy`:



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*.

Carrying out this attack requires

- Determine the location of injected code position on stack when `func()` is called.
 - So as to change stored return address on stack to point to it
 - Location of injected code is fixed relative to the location of the stack frame
- Program P should not contain the `'\0'` character.
 - Easy to achieve
- Overflow should not crash program before `func()` exits.

Some unsafe C lib functions

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

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

`gets (char *s)`

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

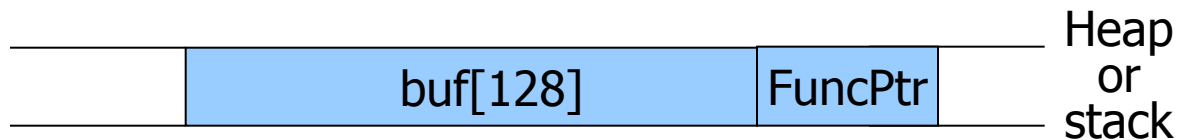
`sprintf (const char *format, ...)`

⋮

Other control hijacking opportunities

- In addition to overwrite return address on the stack, can also use overflow to overwrite the following:

- Function pointers: (used in attack on PHP 4.0.2)



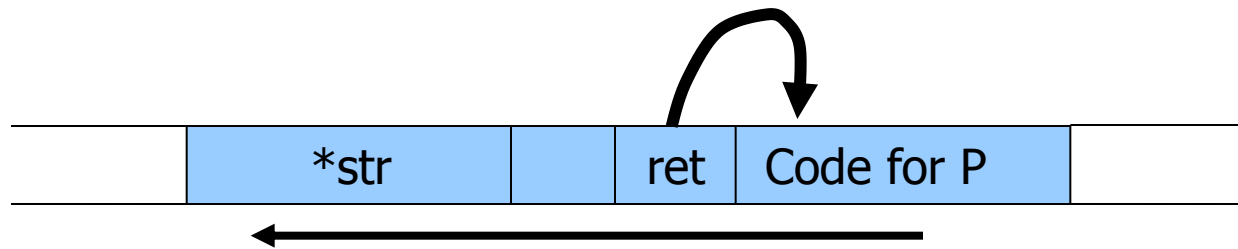
- Overflowing buf will override function pointer.
- Longjmp buffers: `longjmp(pos)` (used in attack on Perl 5.003)
 - Overflowing buf next to pos overrides value of pos.

return-to-libc attack

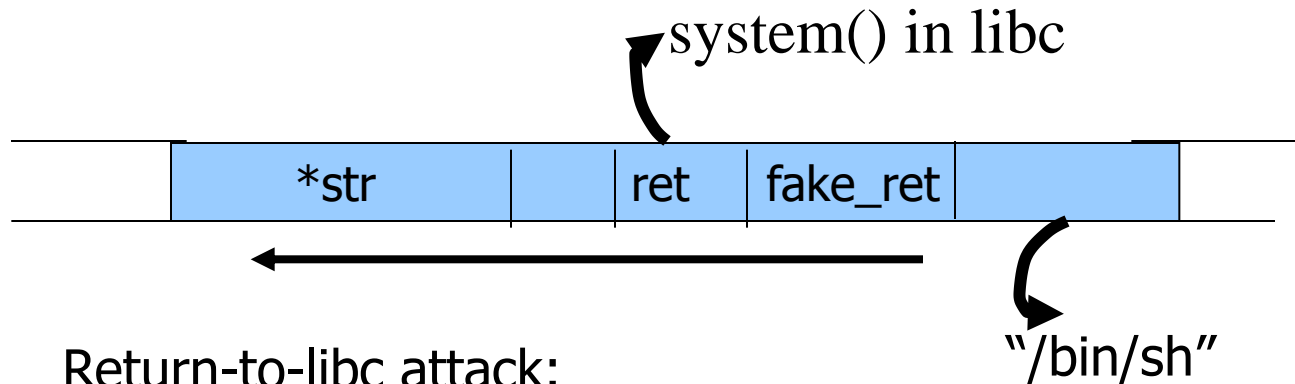
- “Bypassing non-executable-stack during exploitation using return-to-libc” by c0ntex
- Overflow ret address to point to injected shell code requires execution of injected code
 - Many defenses exist, e.g., data execution prevention
- Return-to-libc overwrites the return address to point to functions in libc (such as system())
 - Executing existing code
 - But set up the parameters so that the attacker gets a shell

return-to-libc attack

- Illustrating return-to-libc attack



Shell code attack: Program P: `exec("/bin/sh")`



Return-to-libc attack:

Return-oriented programming

- Goal: executing arbitrary code without injecting any code.
- Observations:
 - Almost all instructions already exist in the process's address space, but need to piece them together to do what the attacker wants
- Attack:
 - Find instructions that are just before “return”
 - Set up the stack to include a sequence of addresses so that executing one instruction is followed by returning to the next one in the sequence.
- Effectiveness: has been shown that arbitrary program can be created this way

Off by one buffer overflow

- Sample code

```
func f(char *input) {  
    char buf[LEN];  
    if (strlen(input) <= LEN) {  
        strcpy(buf, input)  
    }  
}
```

What could go wrong here?

Heap Overflow

- Heap overflow is a general term that refers to overflow in data sections other than the stack
 - buffers that are dynamically allocated, e.g., by malloc
 - statically initialized variables (data section)
 - uninitialized buffers (bss section)
- Heap overflow may overwrite other data allocated on heap
- By exploiting the behavior of memory management routines, may overwrite an arbitrary memory location with a small amount of data.
 - E.g., SimpleHeap_free() does
 - `hdr->next->next->prev := hdr->next->prev;`

Finding buffer overflows

- Hackers find buffer overflows as follows:
 - Run web server on local machine.
 - **Fuzzing**: 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.
- Then use disassemblers and debuggers (e..g IDA-Pro) to construct exploit.
- How to defend against buffer overflow attacks?

Preventing Buffer Overflow Attacks

- Use type safe languages (Java, ML).
- Use safe library functions
- Use better software development process, e.g., manual code review
- Static source code analysis.
- Non-executable stack
- Run time checking: StackGuard, Libsafe, SafeC, (Purify), and so on.
- Address space layout randomization.
- Detection deviation of program behavior
- Access control to control aftermath of attacks... (covered later in course)

Static source code analysis

- Statically check source code to detect buffer overflows.
- Main idea: automate the code review process.
- Several tools exist:
 - Example: Coverity (Engler et al.): Test trust inconsistency.
- Find lots of bugs, but not all.

Bugs to Detect in Source Code Analysis

- Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code

- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Data Execution Prevention

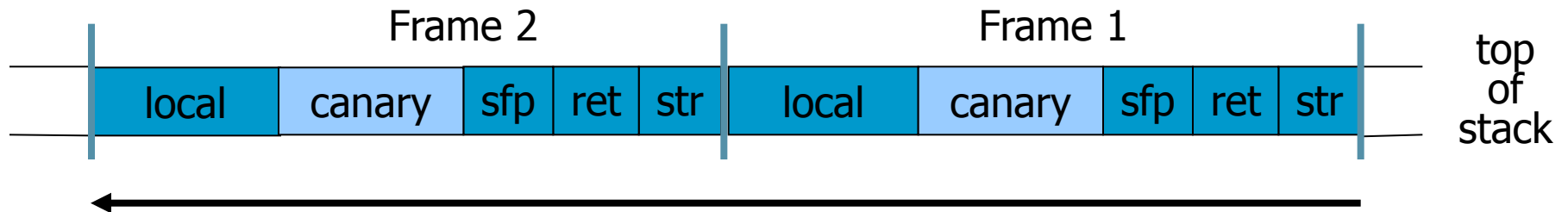
- Basic stack exploit can be prevented by marking stack segment as non-executable.
- More generally, prevent the execution of any memory page that can be dynamically changed
 - Supported in Windows since XP SP2. Code patches exist for Linux, Solaris.

Problems:

- Does not defend against ‘return-to-libc’ or “return-oriented programming”.
- Could potentially break JIT (just-in-time) compiling, or other legacy applications (e.g., LISP compiler)

Run time checking: StackGuard

- There are many run-time checking techniques ...
- StackGuard 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.
 - Hence, attacker cannot use string functions to corrupt stack.

Randomization: Motivations.

- Buffer overflow, **return-to-libc**, and **return-oriented programing** exploits need to know the (virtual) address to which pass control
 - Address of attack code in the buffer
 - Address of a standard kernel library routine
- Same address is used on many machines
 - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce **artificial diversity**
 - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Address Space Layout Randomization

- Arranging the positions of key data areas randomly in a process' address space.
 - e.g., the base of the executable and position of libraries (libc), heap, and stack,
 - Effects: for return to libc, needs to know address of the key functions.
 - Attacks:
 - Repetitively guess randomized address
 - Spraying injected attack code
- Vista has this enabled, software packages available for Linux and other UNIX variants

Format string problem

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

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

- Most likely program will crash: DoS.
- If not, program will print memory contents. Privacy?
- Full exploit using `user = "%n"`

Correct form:

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

Format string attacks (“%n”)

- `printf(“%n”, &x)` will change the value of the variable `x`
 - 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

History

- Danger 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

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?

Assuming short uses 16 bits.

What will they output?

Where Does Integer Overflow Matter?

- Allocating spaces using calculation.
- Calculating indexes into arrays
- Checking whether an overflow could occur
- Direct causes:
 - Truncation; Integer casting

C Data Types, Vary between Platforms

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

When casting occurs in C?

- When assigning to a different data type
- Type promotion; for binary operators `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`,
 - Integer types smaller than `int` are promoted when an operation is performed on them. If all values of the original type can be represented as an `int`, the value of the smaller type is converted to an `int`; otherwise, it is converted to an unsigned `int`.
 - There are other promotion rules when mixing signed with unsigned, and with integer types of different length
- For unary operators
 - `~` changes type, e.g., `~((unsigned short)0)` is `int`
 - `++` and `--` does not change type

Integer Overflow Vulnerabilities

Example

- **Example:**

```
const long MAX_LEN = 20K;  
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?

An Integer Overflow Vulnerability Example (from Phrack)

```
int main(int argc, char *argv[]) {  
    unsigned short s; int i; char buf[80];  
    if (argc < 3){ return -1; }  
    i = atoi(argv[1]);  
    s = i;  
    if(s >= 80) { printf("Input too long!\n"); return -1; }  
    printf("s = %d\n", s);  
    memcpy(buf, argv[2], i);  
    buf[i] = '\0'; printf("%s\n", buf); return 0;  
}
```

Another Integer Overflow

Vulnerability Example (from Phrack)

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

What could go wrong?

Yet Another Integer Overflow Vulnerability Example (from Phrack)

```
int table[800];  
int insert_in_table(int val, int pos){  
    if(pos > sizeof(table) / sizeof(int)) {  
        return -1;  
    }  
    table[pos] = val; return 0;  
}
```

What could go wrong?

Yet Another Example

// The function is supposed to return false when
// x+y overflows unsigned short.

// Does the function do it correctly?

```
bool IsValidAddition(unsigned short x,  
    unsigned short y) {  
    if (x+y < x)  
        return false;  
    return true;  
}
```


Coming Attractions ...

- Malwares

