Cristina Nita-Rotaru

CS526: Information security

Software security

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

Secure Programs

- Software quality
- Penetrate and patch approach

To understand program security one has to understand if the program behaves as its designer intended and as the user expects it

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
  - Obtain sensitive information: Compromises confidentiality
  - Execute arbitrary code: Compromises integrity
- Software vulnerability enables the attacker to run with privileges of other users, thus 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

1: Input validation

Sources of Input that Need Validation

- Sources of input for local applications
  - Command line arguments
  - Environment variables
  - Configuration files, other files
  - Inter-Process Communication call arguments
  - Network packets

- Sources of input for web applications
  - Web form input
  - Scripting languages with string input

Command line as a Source of Input

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

Intention: get a file name from input and then cat the file;

What can go wrong?

- Attacker can add to the command by 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

exec

```
#include <unistd.h>
int execl( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execle( const char *path, const char *arg  , ...,
char *const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv [],
char *const envp[] );
```

- Family of functions for replacing process's program with the one inside the exec() call.

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

Input Validation in Web Applications

- SQL injection
  - Caused by failure to validate/process inputs from web forms before using them to create SQL queries
- Cross Site Scripting
  - Caused by failure to validate/process inputs from web forms or URL before using them to create the web page
- Cross Site Request Forgery is not an input validation issue

A Remote 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

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
```

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/...
```

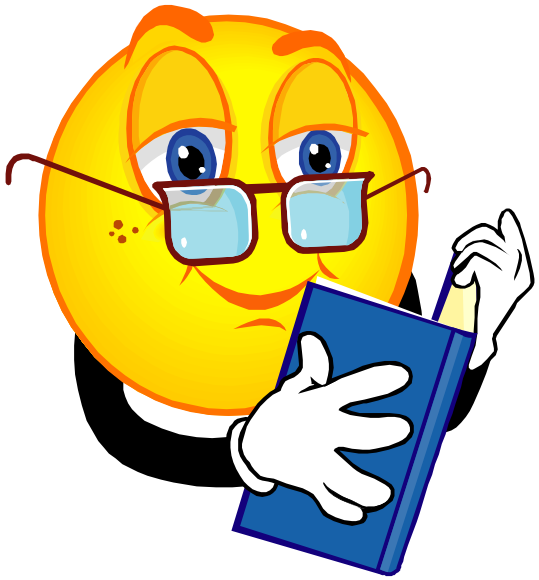
  - IIS first checks input, then expands unicode

Dealing 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

Take home lessons: Input Validation

- 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



2: Time-of-check-to-time-of-use

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

- TOCTTOU, pronounced "TOCK too“
- A class of software bugs 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 because P changed or X changed

Example

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

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

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

- 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)

Example: Attack

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

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

- Between the access verification and the open of the file, the attacker removed the file "file" and created a symbolic link pointing to the file "/etc/passwd"; this is done through some other program and requires coordination
- The result is that the program will open the paswwd file even if the user should not have had access to write to it

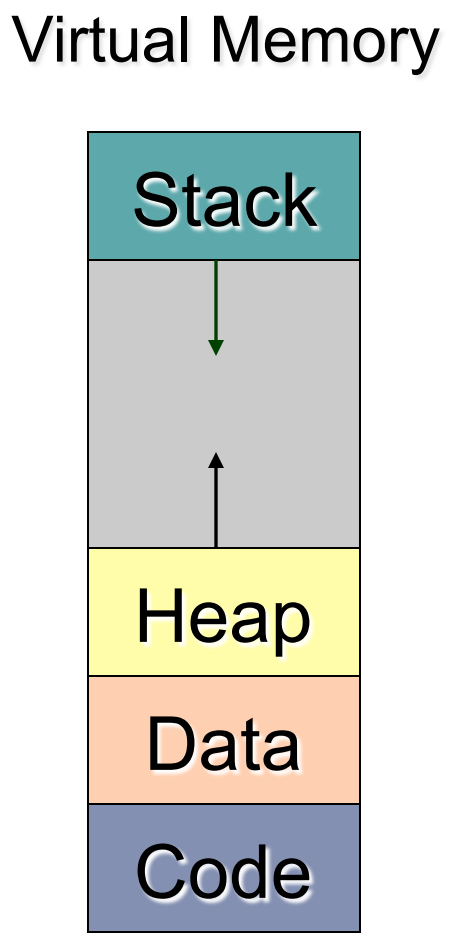
Take home lessons: TOCTTOU

- Exploiting a TOCTTOU vulnerability 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

3: Buffer overflow

Programs and Memory

- The operating system creates a process by assigning memory and other resources
- **Stack**: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions
- **Heap**: dynamic memory for variables that are created with malloc, calloc, realloc and disposed of with free
- **Data**: initialized variables including global and static variables, un-initialized variables
- **Code**: the program instructions to be executed



What is a Buffer Overflow?

- Buffer overflow occurs when a program or process tries to store more data in a buffer than the buffer can hold
- Very dangerous because the extra information may:
  - Affect user's data
  - Affect user's code
  - Affect system's data
  - Affect system's code

Why Does Buffer Overflow Happen?

- No check on boundaries
  - Programming languages give user too much control
  - Programming languages have unsafe functions
  - Users do not write safe code
- C and C++, are more vulnerable because they provide no built-in protection against accessing or overwriting data in any part of memory
  - Can't know the lengths of buffers from a pointer
  - No guarantees strings are null terminated

Why Buffer Overflow Matters

- Overwrites:
  - other buffers
  - variables
  - program flow data
- Results in:
  - erratic program behavior
  - a memory access exception
  - program termination
  - incorrect results
  - breach of system security

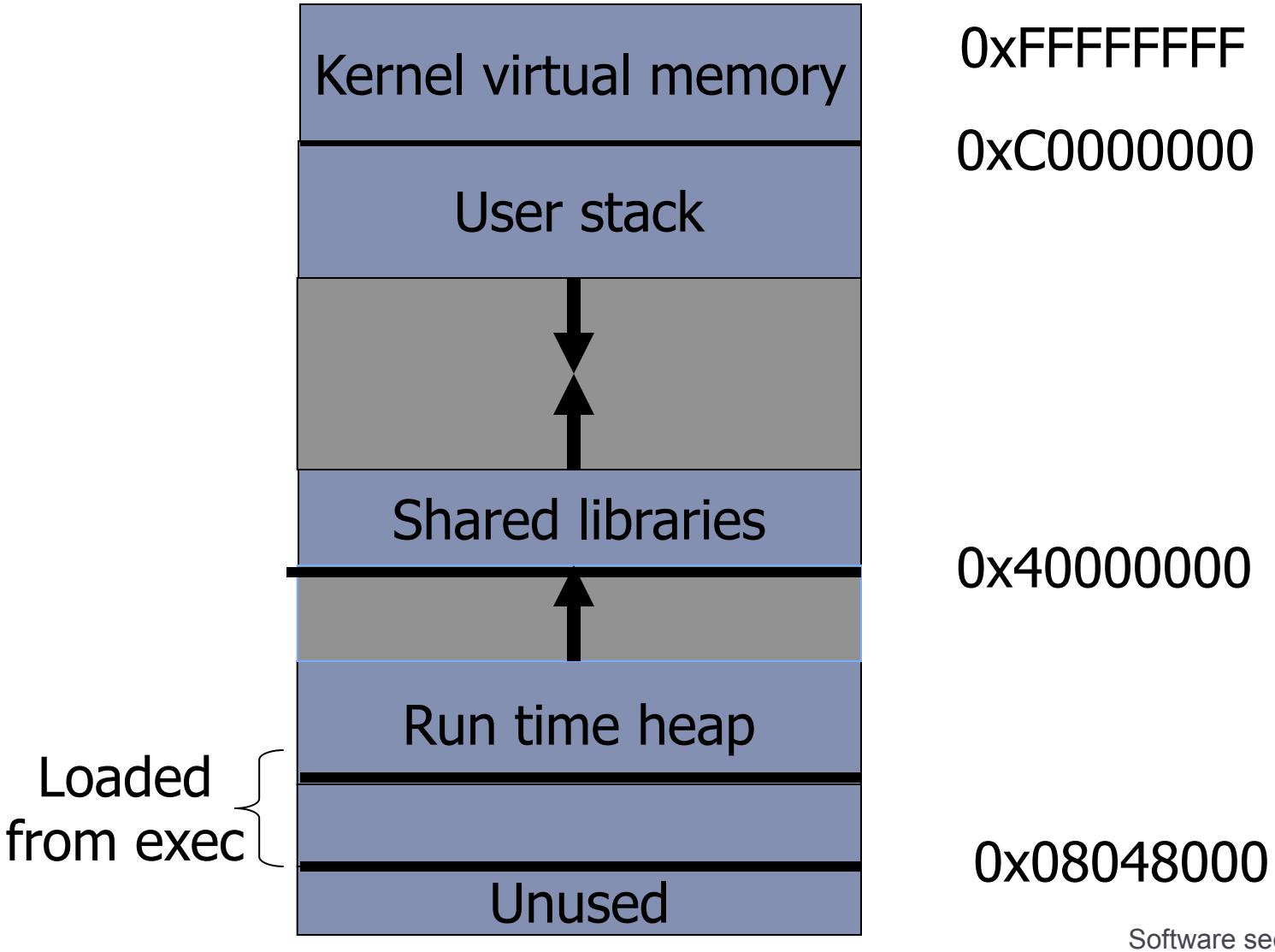
History

- 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

Example: Linux Process Memory Layout

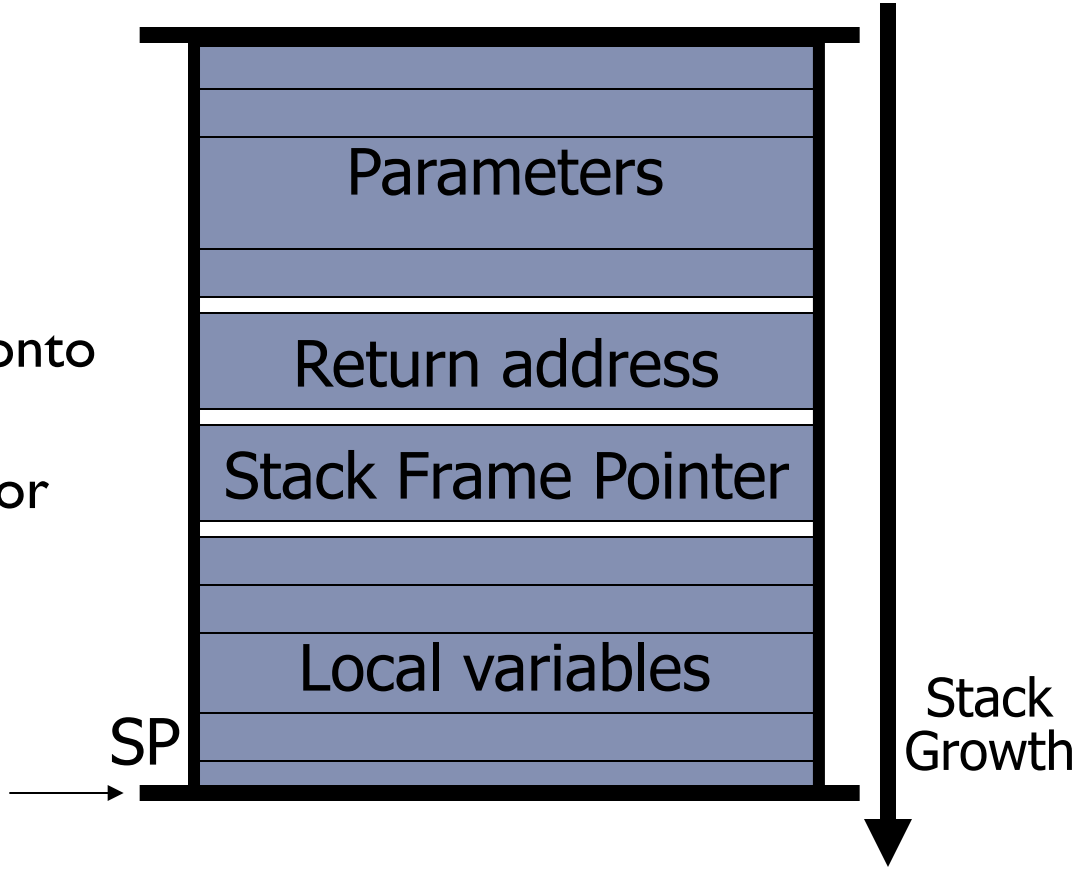


C Program Execution

- PC (program counter or instruction pointer) points to next machine instruction to be executed
- Procedure call:
  - Prepare parameters
  - Save state (SP (stack pointer) and PC) and allocate on stack local variables
  - Jumps to the beginning of procedure being called
- Procedure return:
  - Recover state (SP and PC (this is return address)) from stack and adjust stack
  - Execution continues from return address

Stack Frame

- Parameters for the procedure
- Save current PC onto stack (return address)
- Save current SP value onto stack
- Allocates stack space for local variables by decrementing SP by appropriate amount



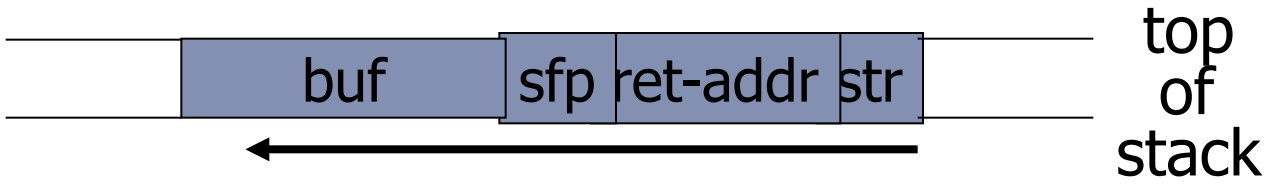
Example of a Stack-based Buffer Overflow

- Suppose a web server contains a function:

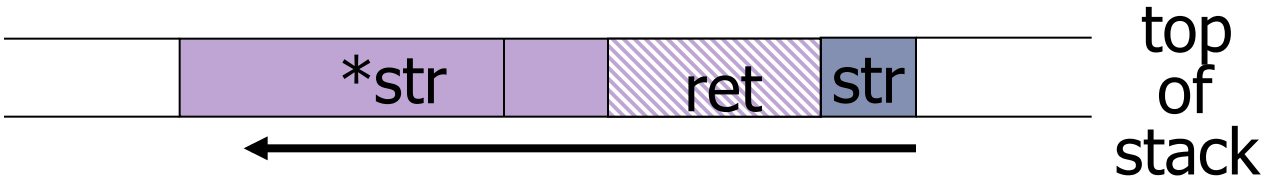
```
void my_func(char *str) {
  char buf[128];

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

- When the function is invoked the stack looks like:



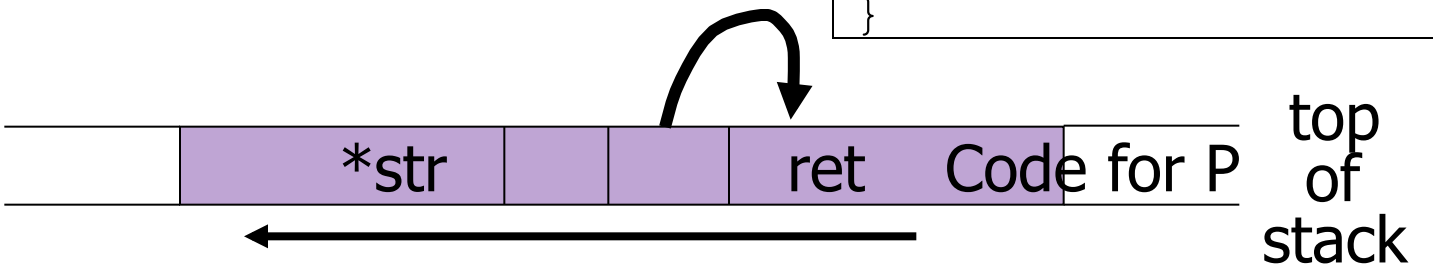
- What if **`*str`** is 136 bytes long? After **`strcpy:`**



Basic Stack Exploit

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

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



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

(exact shell code by Aleph One)

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

Carrying out this attack requires

- Determine the location of injected code position on stack when func() is called.
  - So as to change RET 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 (conts 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.

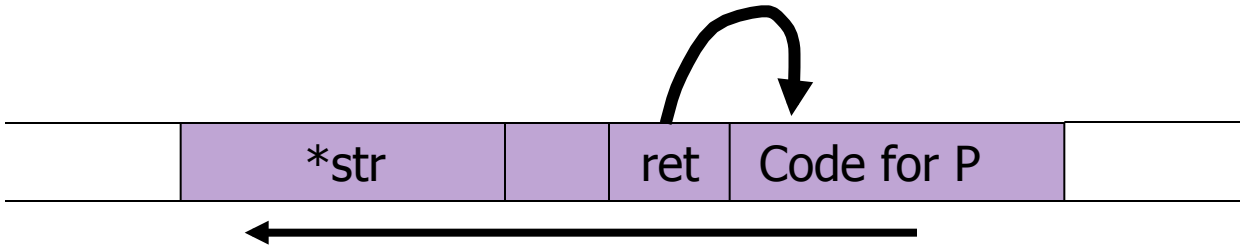
- Longjmp buffers: longjmp(pos) (used in attack on Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

return-to-libc attack

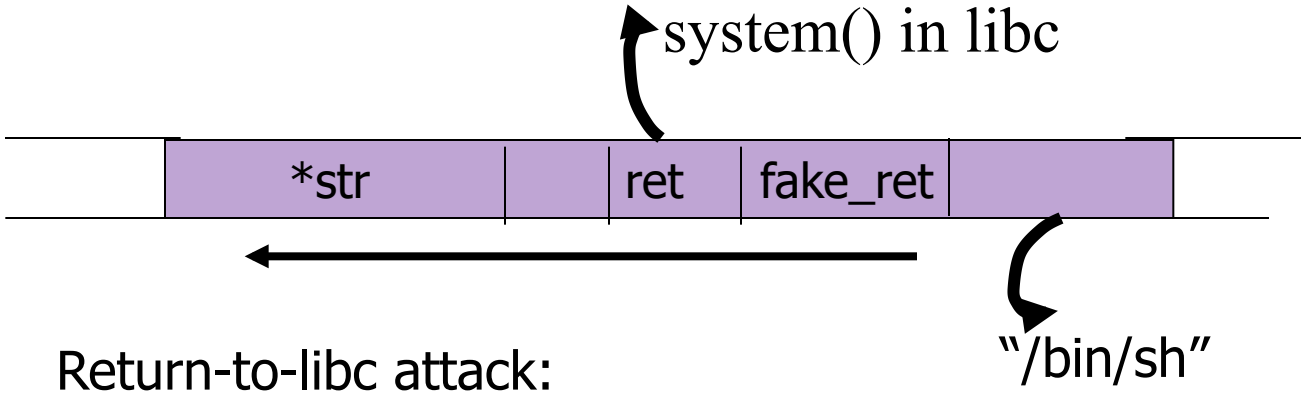
- "Bypassing non-executable-stack during exploitation using return-to-libs" by c0ntex
- Overflow ret address to point to injected shell code requires execution of injected code
  - Many defenses exist
- Return-to-libc overwrites the return address to point to functions in libc (such as system())
  -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

return-to-libc attack

- Illustrating return-to-libc attack



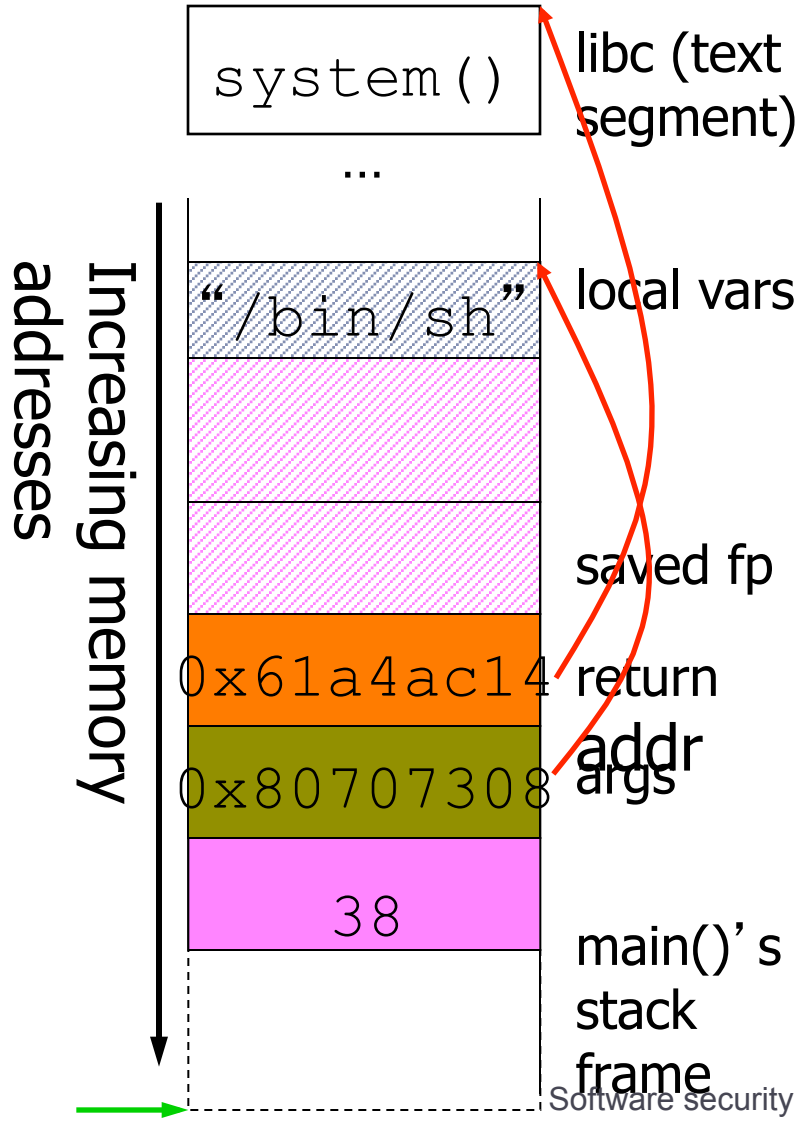
Shell code attack: Program P: `exec( "/bin/sh" )`



Return-to-libc attack:

Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function
  - e.g., `system("/bin/sh");`
- Return-to-libc attack



Slide thanks to Brad Karp, UCL.

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
- Static source code analysis
- Non-executable stack
- Run time checking: StackGuard, Libsafe, SafeC, (Purify)
- Address space layout randomization
- Instruction set randomization
- Detection deviation of program behavior
- Access control to control aftermath of attacks

Static Source Code Analysis

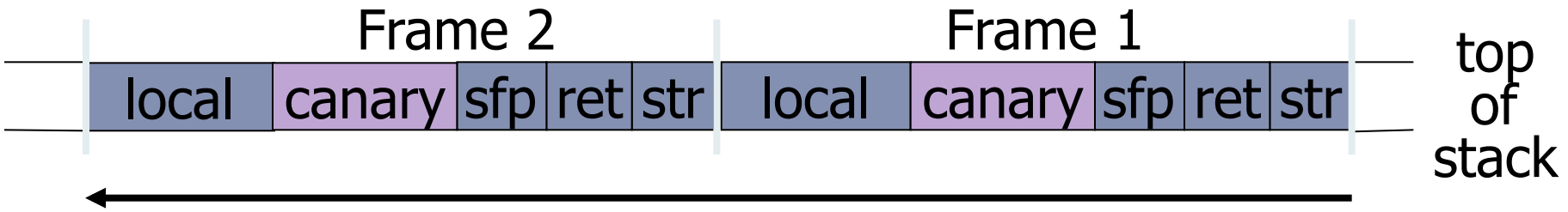
- Statically check source code to detect buffer overflows.
- Automate the code review process.
- Several tools exist:
  - Coverity (Engler et al.): Test trust inconsistency.
  - Microsoft program analysis group:
    - PREfix: looks for fixed set of bugs
    - PREfast: local analysis to find idioms for prog. errors.
  - Berkeley: Wagner, et al. Test constraint violations.
- 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

Run Time Checking: 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.
  - Hence, attacker cannot use string functions to corrupt stack.

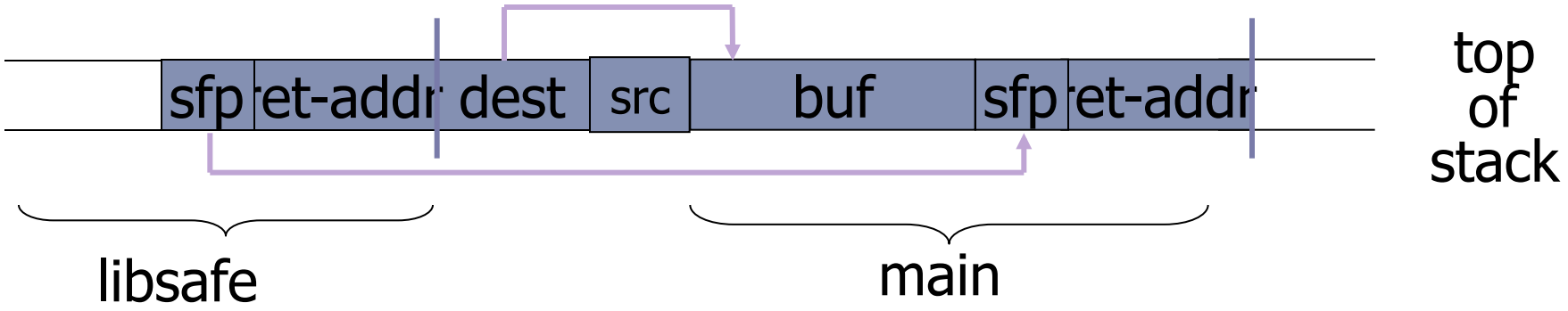
StackGuard: Implementation

- StackGuard implemented as a GCC patch
  - Program must be recompiled
  - Minimal performance effects: 8% for Apache
- Newer version: PointGuard
  - Protects function pointers and setjmp buffers by placing canaries next to them
  - More noticeable performance effects

- Note: Canaries do not offer full protection
  - Some stack attacks can leave canaries untouched

Run Time Checking: Libsafe

- Dynamically loaded library.
- Intercepts calls to strcpy (dest, src)
  - Validates sufficient space in current stack frame:
    |frame-pointer – dest| > strlen(src)
  - If so, does strcpy.
    Otherwise, terminates application.



Run Time Checking: 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)

Marking stack as non-execute

- Basic stack exploit can be prevented by marking stack segment as non-executable.
  - Support in Windows since XP SP2. Code patches exist for Linux, Solaris.
- Problems:
  - Does not defend against `return-to-libc' or "return-oriented programming".
  - Some apps need executable stack (e.g. LISP interpreters).
  - Does not block more general overflow exploits:
    - Overflow on heap, overflow func pointer.

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

Instruction Set Randomization

- Instruction Set Randomization (ISR)
  - Each program has a *different* and *secret* instruction set
  - Use translator to randomize instructions at load-time
  - Attacker cannot execute its own code.
- What constitutes instruction set depends on the environment.
  - For binary code, it is CPU instruction
  - For interpreted program, it depends on the interpreter

Instruction Set Randomization

- An implementation for x86 using the Bochs emulator
  - Network intensive applications don't have too much performance overhead
  - CPU intensive applications have one to two orders of slow-down
- Not yet used in practice

4: Format string

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

5: Integer overflow

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?

C Data Types

- 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
- 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

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 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?

Another Example

```
int ConcatBuffers(char *buf1, char *buf2,
                  size_t  len1, size_t len2) {
    char buf[0xFF];
    if ((len1 + len2) > 0xFF) return -1;
    memcpy(buf, buf1, len1);
    memcpy(buf+len1, buf2, len2);
    return 0;
}
```

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;
}
```

Take home lessons

- Software vulnerabilities are a huge problem
- Most common result from improper input validation and buffer overflow
- Avoid using functions that don't check boundaries
- Pay attention to integer overflow when checking sizes and copying buffers