CSC429 – Computer Security

LECTURE 5
SOFTWARE SECURITY

Mohammed H. Almeshekah, PhD
meshekah@ksu.edu.sa

What is a Secure Software

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

Common Software Vulnerability

- Buffer overflows
- Input validation
- Format string problems
- Integer overflows

Software Security

Buffer Overflow

Buffer Overflow

- Buffer overflow occurs when a program or process tries to store more data in a buffer than the buffer can hold

- Why does an overflow happen?
  - No check on boundaries.

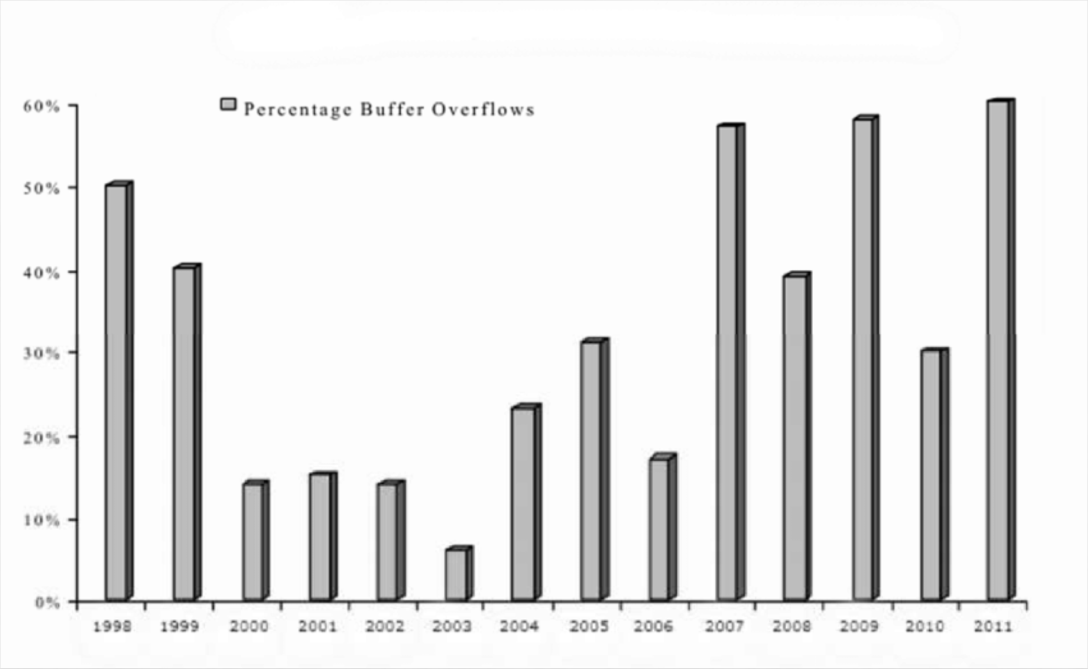
- C and C++, are more vulnerable
  - They provide no built-in protection against accessing or overwriting data in any part of memory
  - Can't know the lengths of buffers from a pointer.
  - No guarantees strings are null terminated.

Why Do We Care?

- An overflow overwrites:
  - other buffers
  - variables
  - program flow data
- Could results in:
  - Unexpected program behavior
  - A memory access exception
  - Program termination
  - Incorrect results
  - Breach of system security

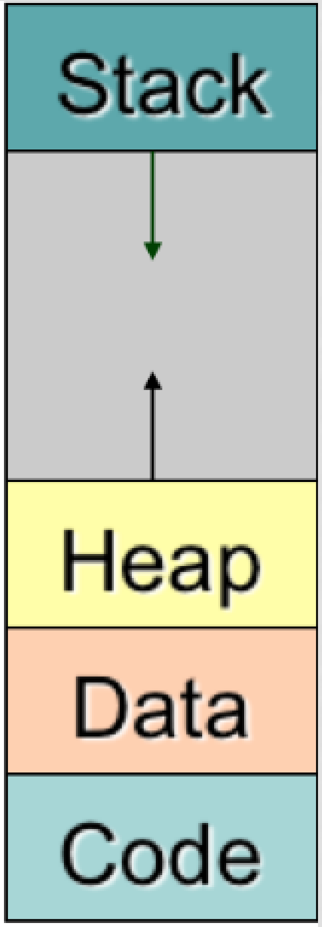
History of Buffer Overflows

- Used in 1988's Morris Internet Worm.
- Alphe One's "Smashing The Stack For Fun And Profit" in 1996 popularizes stack buffer overflows



Programs and System's Memory

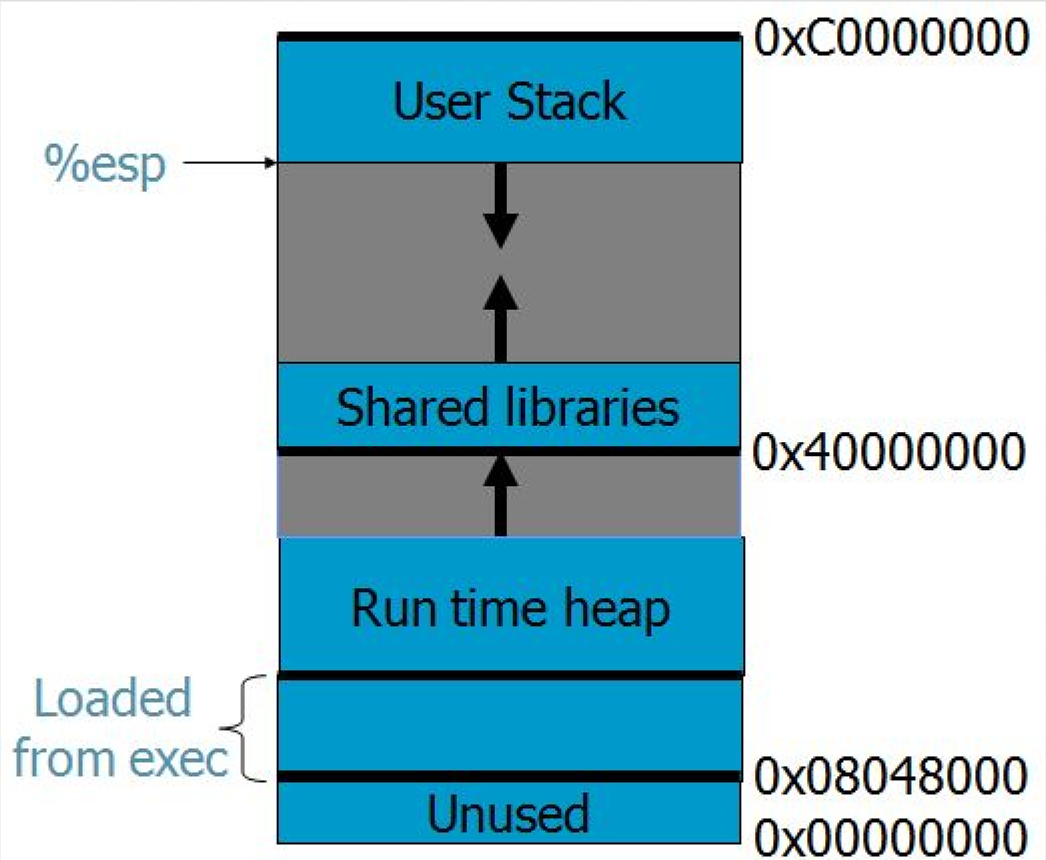
- The operating system creates a process by assigning memory and other resources
- Stack:
  - keeps track of the point to which each active subroutine should return control when it finishes executing;
  - stores variables that are local to functions
- Heap:
  - dynamic memory for variables that are created 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



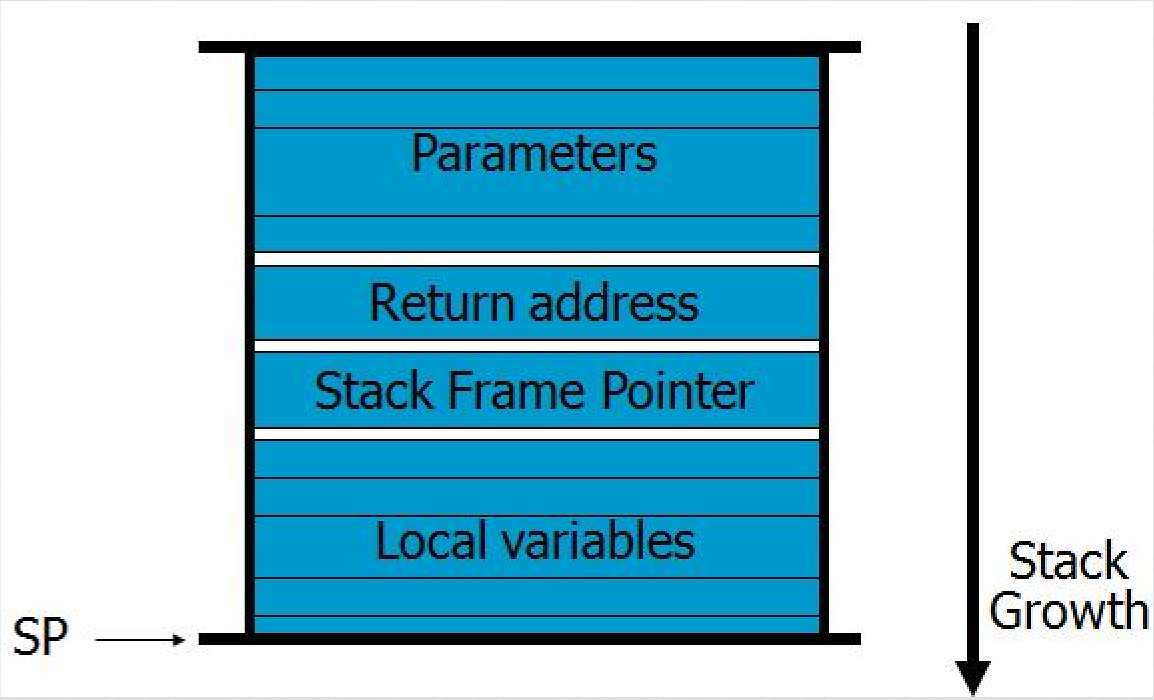
Stack Overflow

- Stack overflow.
  - Shell code
  - Overflow function pointers.
  - Return-to-libc
    - Overflow sets ret-addr to address of libc function
  - Off-by-one.

Linux process memory layout



Stack Frame



Parameters
Return address
Stack Frame Pointer
Local variables
SP
Stack
Growth

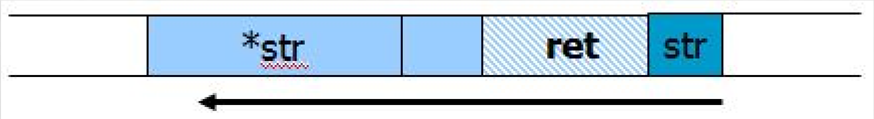
How Does a Buffer Overflow Happen?

- Suppose a web server contains a function:

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

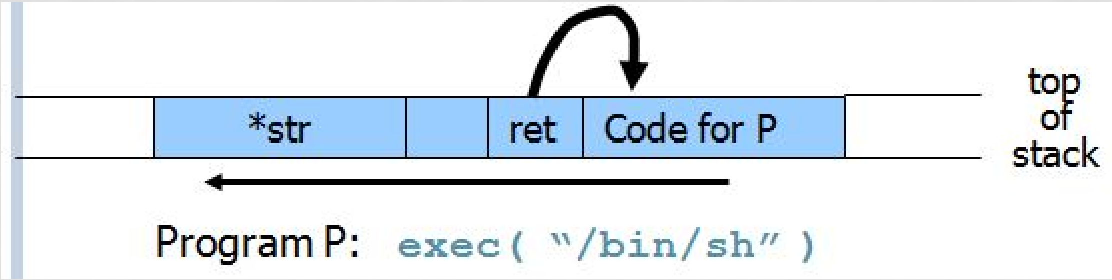
- When the function is invoked the stack looks like:

- 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:



- When `func()` exits, the user will be given a shell!!
- Code now is in the 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, …)
- sprintf (char *str, conts char *format, …)

Exploiting Buffer Overflow Vulnerability

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

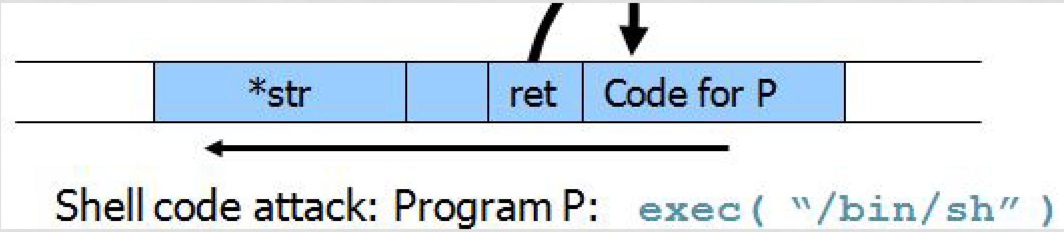
Buffer Overflow Protection

- When you overflow the buffer the attackers code is inserted in the Stack.
- To address the issue, mark the stack as un-executable Stack.
- Does it solve all the buffer overflow problems?

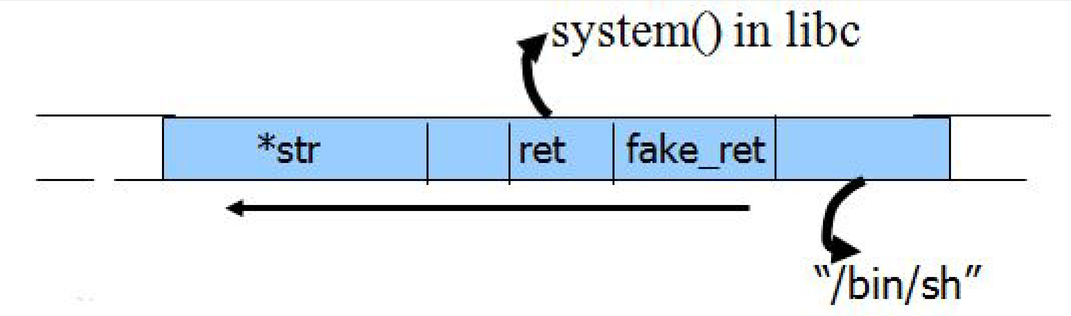
Return-to-libc Attack

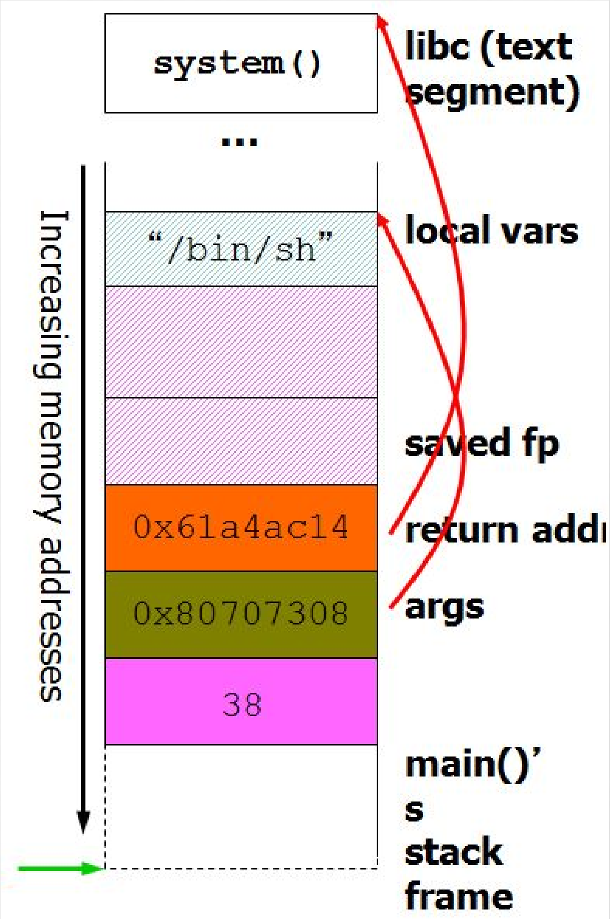
- Bypassing non-executable-stack during exploitation using return-to-libc

- Previous Buffer Overflow attack:



- Return-to-libc attack:




system()
libc (text segment)
...
"/bin/sh"
local vars
saved fp
0x61a4ac14
return addr
0x80707308
args
38
main()'s stack frame
Increasing memory addresses

Other Control Hijacking

- 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.

Off-by-one Buffer Overflow

- Attack goal may be just to crash the program.
- What could go wrong in this code?

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

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.

Preventing Buffer Overflow Attacks

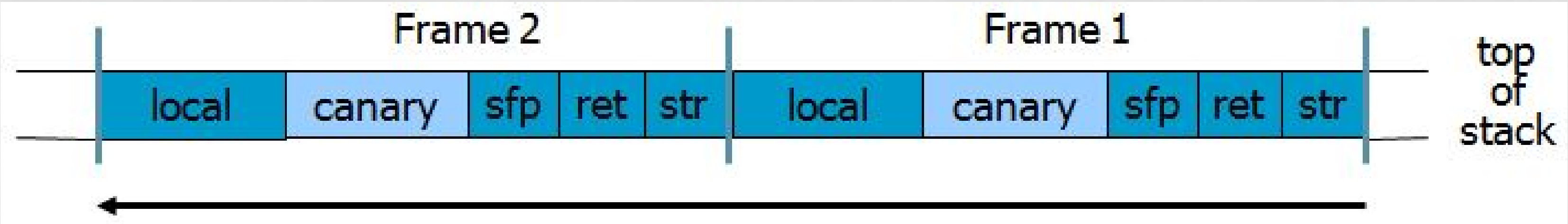
- Use safer languages (e.g. Java).
- Use safer library functions.
- Static source code analysis.
- Non-executable stack.
- Run time checking: StackGuard, Libsafe, SafeC, Purify.
- Address space layout randomization.
- Access control to control aftermath of attacks.

Static Source Code Analysis

- Statically check source code to detect buffer overflows.
- Main idea: automate the code review process.
- Several tools exist:
  - Coverity & Veracode

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.

StackGuard

- StackGuard implemented as a GCC patch.
  - Program must be recompiled.
  - Minimal performance effects.
- Newer version: PointGuard.
  - Protects function pointers and setjmp buffers by placing canaries next to them.
  - More noticeable performance effects.
- Canaries don’t offer full-proof protection.
  - Advanced attacks can overcome it.

Randomization – Motivation

- Buffer overflow and return-to-libc 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.
  - For example, 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
- Windows Vista has this enabled, also 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 – Cont.

- An implementation for x86 using the Bochs emulator
  - network intensive applications doesn't have too much performance overhead.
  - CPU intensive applications have one to two orders of slow-down.