# CS 526: Information Security

## Software Vulnerabilities

# Definition

- Software is secure if it does not cause violations of security policy
  1. No violations of the software's own security policy (i.e., it behaves as its designer intended)
  2. No violations of the security policy of the system on which it runs, i.e., it behaves as expected by those deploying and using it
- A system might use individually secure software components, yet not be secure

# Universal Composability Framework

- Model for the analysis of security protocols
  - Provides strong security properties
- If each of A, B, C, …etc is universally composable, then so are arbitrary combinations of A, B, C, …
  - Achieving that property is not easy, and is provably impossible for some tasks
- For most modules used in large software (like a browser), security is not composable

# Race Conditions

# TOCTOU Race Condition

- "**T**ime **O**f **C**heck **T**ime **O**f **U**se" vulnerability
- System first (i) checks whether a condition is true, then (ii) it uses the result of that check
- E.g., (i) checks that user U is authorized for resource X, if so it (ii) grants U access to X
  - TOCTOU attack would consist of modifying U or X in between the above steps (i) and (ii)
  - Often it consists of U causing the change in X

# TOCTOU Example

1. A setuid root program uses the **access()** system call to check whether the user U running it has access permission for file X (this comes before it does any operations on file X on behalf of U)
2. If the above test checks out, then the setuid root program uses **fopen()** and carries out the operations on file X on behalf of user U

- Attack: In between steps 1 and 2, U replaces X with a symbolic link pointing to "/etc/passwd"
  - p = probability to achieve the right timing (low!)

# Intractability of TOCTOU

- Impossibility result
  - There is no portable, deterministic technique for avoiding TOCTOU vulnerabilities
- Probabilistic solution: Yes
  - Assumes attacker can always, with enough effort, carry out a TOCTOU race against a setuid program
  - Make sure “enough effort” is impractical for the adversary
  - Similar to crypto assumption about a secret key

# TOCTOU Probabilistic Solution

- Call to access/fopen is followed by k "strengthening rounds"
  - Each round consists of an additional access/fopen followed by a check to verify that the file that was opened was the same as had been opened previously (e.g., by comparing inodes)
  - For attack to succeed, every access must be on same good file, *and* every fopen must be on same bad file: prob of success ~ $p^{2k+1}$ ; p = success prob for 1 race
  - Defensive effort is linear in k

# Lack of Input Validation

# Command Line as Source of Input

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

- **Programmer's intent**: 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**"

# Input Validation in Web Applications

- PHP passthru example:

```
echo 'Your usage log:<br />';
$username = $_GET['username'];
passthru(“cat /logs/usage/$username”);
```

- PHP **passthru(string)** executes command
- **What can go wrong:** Attackers can put “;” to input to run desired commands, e.g., “**username=John;cat%20/etc/passwd**”

# Incomplete Mediation

- = “Incomplete Checking”
- Example from an online shopping web site:
  - Shopping data input to a web form, validated on the client, then transferred to the server by including it in a URL:

www.buystuff.com/orders/final&custID=117&
part=67B&qty=10&price=15&shipping=5&total=155

  - The flaw: Not re-validated at the server
  - Dishonest buyer can send with a lower price

# Buffer Overflow

- Buffer overflow occurs when a program or process tries to store more data in a buffer than the buffer can hold
  - Especially dangerous when the attacker gets to choose that data
- May allow the attacker to damage availability, confidentiality, integrity:
  - Crash programs (damage availability)
  - Obtain sensitive info (damage confidentiality)
  - Execute arbitrary code with elevated privileges (damage integrity)

# Why Do Buffer Overflows Happen?

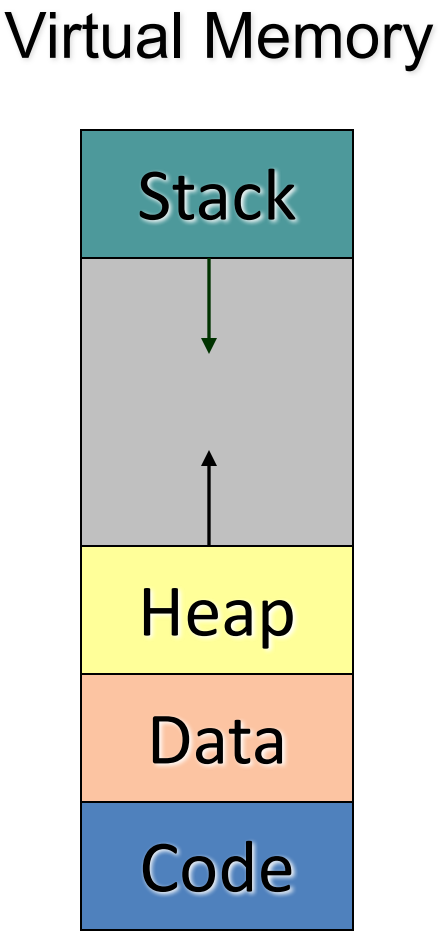
- No checks on bounds
  - Programming languages give user too much control
  - Programming languages have unsafe functions
  - Users do not write safe code
- C and C++, are more vulnerable because they provide no built-in protection against accessing or overwriting data in any part of memory
  - Can't 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

# Background: Programs and Memory

- The operating system creates a process by assigning memory and other resources
- **Code**: the program instructions to be executed
- **Data**: initialized variables including global and static variables, un-initialized variables
- **Heap**: dynamic memory for variables that are created (e.g., with *malloc)* and disposed of (with *free*)
- **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

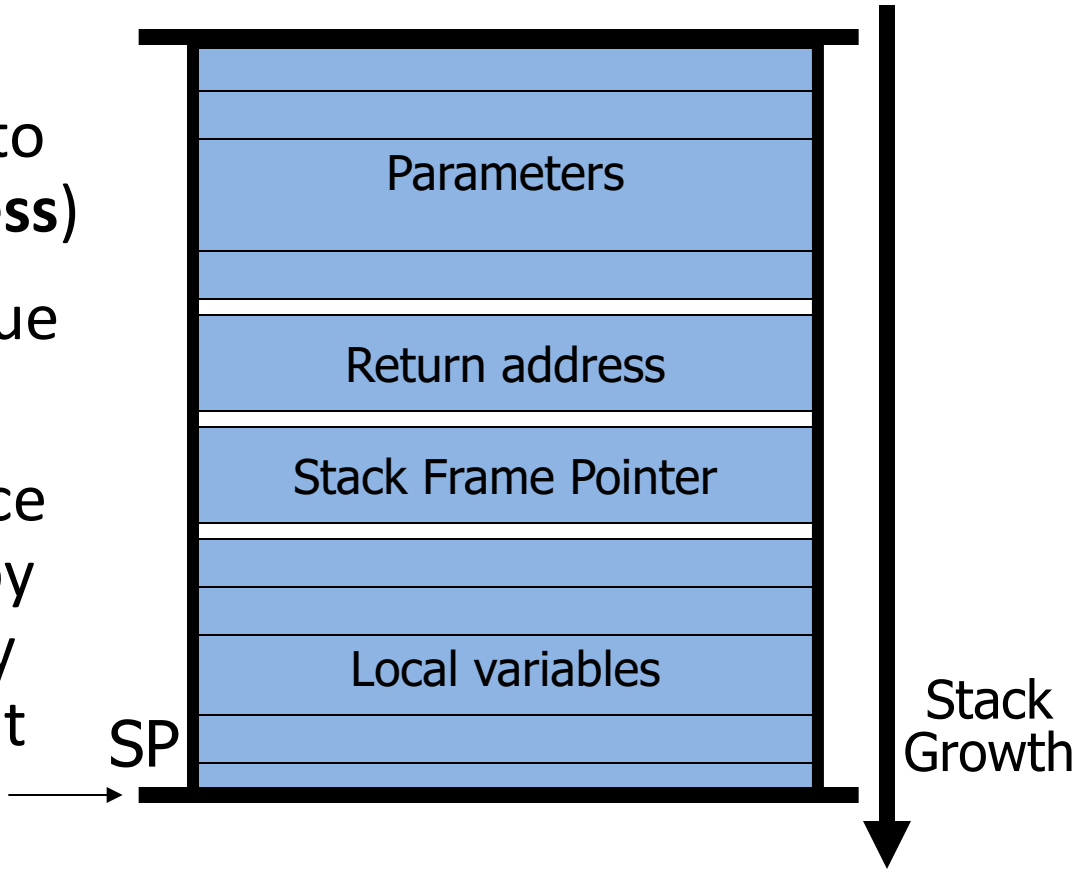


# Background: Program Execution

- 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

# Background: 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



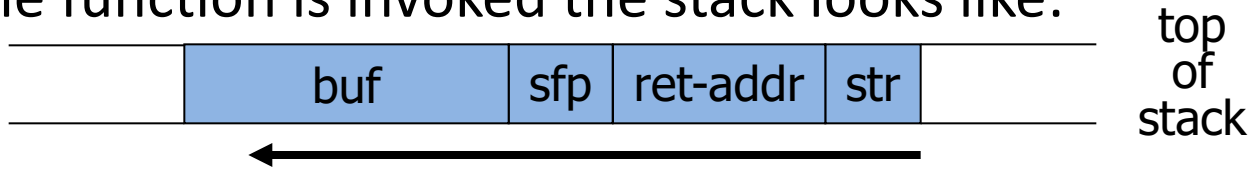
# Example of a Stack-based Buffer Overflow

- Suppose a web server contains a function:

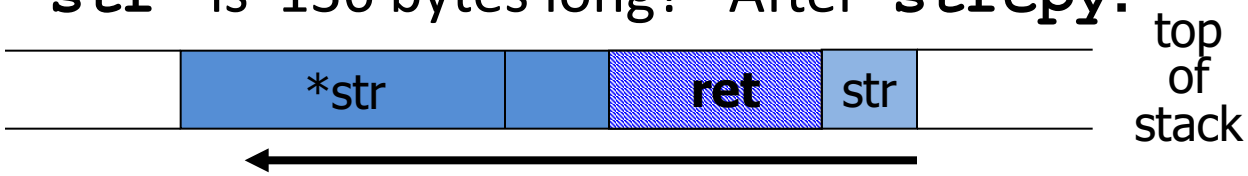
```
void my_func(char *str) {
     char buf[128];

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

- When the function is invoked the stack looks like:




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

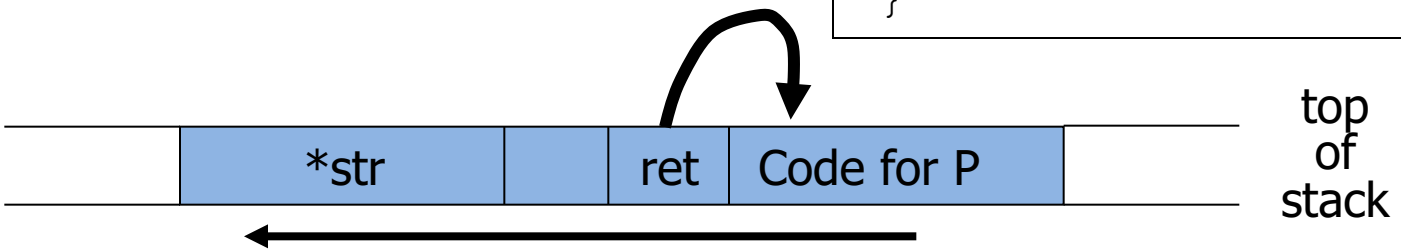


# Basic Stack Exploit

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

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

- Suppose ***str** is such that after **strcpy** stack looks like:




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

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

# Carrying out this Attack Requires

- Determining the location of injected code position on stack when my_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 my_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, … )

printf (const char *format, … )

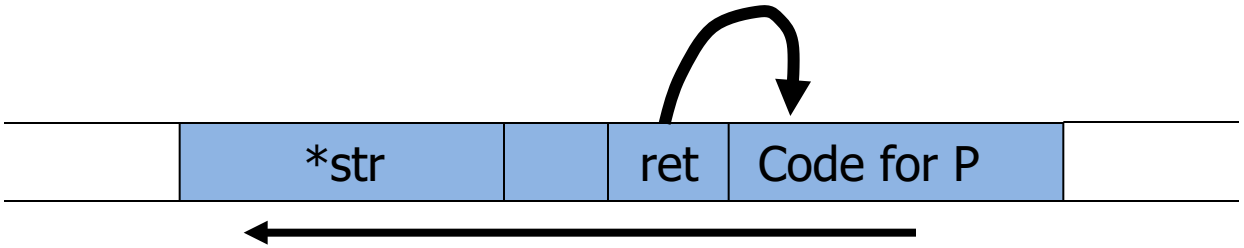
# Non-Executable Stack

- Basic stack exploit can be prevented by hardware support to mark stack segment as non-executable
  - Support in all major operating systems

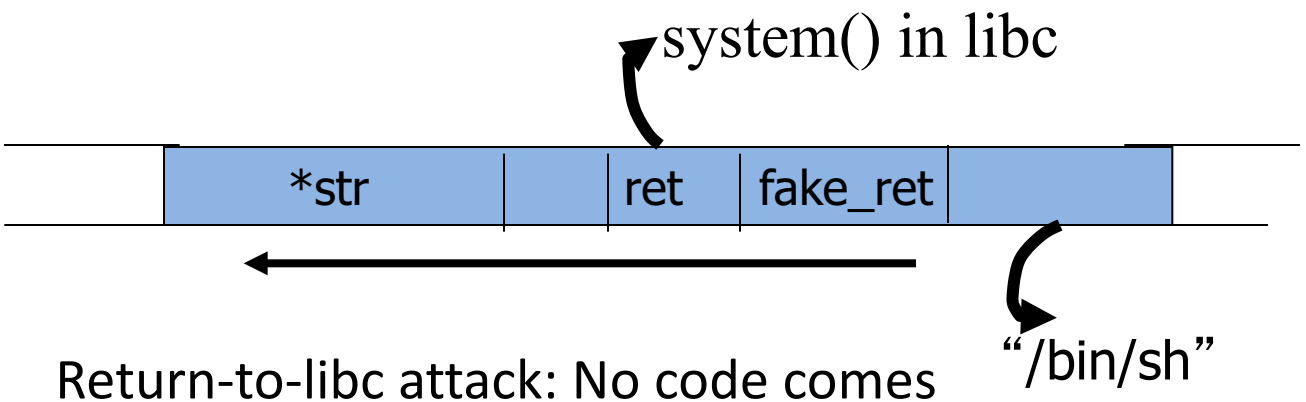
- Does not defend against all attacks
  - E.g., “return-to-libc”

# return-to-libc attack

- “Bypassing **non-executable-stack** during exploitation using return-to-libc” by c0ntex




Shell code attack: Program P: **`exec( “/bin/sh” )`**




Return-to-libc attack: No code comes after ret (only the arg for the call)

# Prevention mechanisms

# Preventing Buffer Overflow Attacks

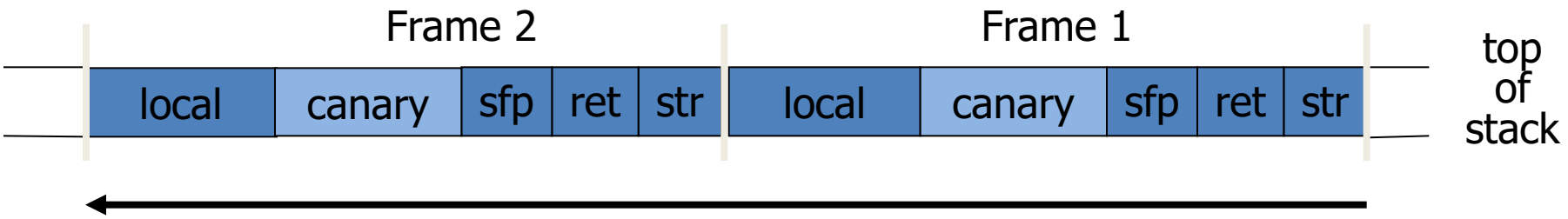
- Use type safe languages (e.g., Java, C#)
- Use safe library functions
- Static source code analysis
- Non-executable stack
- Run time checking (e.g., StackGuard)
- Address space layout randomization
- Instruction set randomization
- Detecting deviation of program behavior

# Static Source Code Analysis

- Statically check source code to detect buffer overflows
- Automate the code review process
  - Several tools exist
  - Expensive (exponential)
  - Typically done for short programs of critical importance
- Can find lots of bugs (no “all” guarantee)

# Run Time Checking: StackGuard

- StackGuard checks for stack integrity at run time
  - E.g., 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
  - Weakness: Adversary knows canary
- Canaries do not offer full protection

# Other Run Time Checking

- Validate sufficient space (LibSafe)
  - E.g., intercept calls to strcpy (dest, src) and check that:
    **`|frame-pointer – dest| > strlen(src)`**
  - If so do strcpy, else terminate application

- Copying to a safe location (StackShield)
  - E.g., at function prologue, copy return address to a safe location, and upon return check that return address still equals the saved copy

# Randomization: Motivations

- Buffer overflow and return-to-libc exploits need to know the (virtual) address to which to 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.
  - Attacker would have to repetitively guess randomized address (or use non-ASLR modules)
- Supported on Windows Vista, 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
- Significant performance overhead
  - Not used in practice (yet)

# Architectural issues

- 64 bit Harder to attack than 32 bit
  - Twice the number of general purpose registers (hence less use of stack than in 32 bit)
  - No-execute bit (set by default) makes stack contents not executable
- Not a substitute for secure programming
  - Attacks still possible (just more work)

# Software Security: Often not in the Design

- “Iterative design” too often means: “Here we tolerate errors, we fix them as users report them”
- Software has many desired characteristics (including security), but security is different
  - The absence of most is very noticeable, e.g., performance (“too slow!”), usability (customers notice immediately), stability (“it crashes!”), …
  - Not so for security: Its absence is not easy to see

# (In)security: Invisible, Hard to Fix

- Invisible:  The least-secure software often has great other attributes (fast, easy-to-use, stable)
- Hard(er) to fix
  - What do you tell your software engineers?
  - They know what to do if you say “make it faster”, “improve the user inferface”, “stop the crashing”
  - But less so if you say “make it less vulnerable to attack” (which attack?); it is orders of magnitude more expensive to fix than to properly design

# Security by Design

- Not as easy as it may appear
- Asymmetry of defense vs attack
  - Programmers have limited time and resources: Deadlines to meet (target release dates), budget constraints
  - Attackers can take as long as they want to find vulnerability, have many possible avenues of attack
  - Programmers tasked with eliminating **all** exploitable vulnerabilities
  - Attackers need only find **one** exploitable vulnerability

# Security by Design (cont'd)

- Difficult to substantiate security claims
  - Formal proofs impractical for large programs
  - Heuristic claims ("I couldn't find vulnerabilities") is not convincing (for good reason)
- Avoid low level languages like C and C++?
  - Performance drawbacks
  - High-level may make calls to low-level-written (may merely act as a conduit for attack)
  - High-level often written in low-level

# When to disclose vulnerabilities ?

- Dealing with vulnerabilities
  - Vulnerabilities can be eliminated through a software patch released by software vendor
  - Or their effects neutralized by measures taken by the organizations where the software is deployed
  - Either case requires the vulnerabilities to have been disclosed to the vendor and potential victims
- Which disclosure procedures give advantage to defense and disadvantage to attackers ?

# Non-Public Disclosure

- Disclose only to a few agencies known to behave responsibly (like CERT or similar)
- These agencies immediately notify software vendors, so they can start working on a patch
  - These agencies publicly disclose only after the security patch is available from the software vendor (usually 1 to 6 months after vendor was notified)
- Widely considered the responsible alternative

# Non-Public Disclosure (cont'd)

- Advantages
  - Does not prematurely inform attackers of vulnerability
  - Gives vendors a head start over attackers
- Drawbacks
  - Potential victims can remain unaware of the vulnerability for months
  - Pointless to wait when attackers may have discovered the vulnerability on their own

# Immediate Public Disclosure

- Publicly disclose as soon as vulnerabilities are discovered
  - Use mailing lists (e.g., BugTraq)
- Disadvantage
  - Potential attackers can use the information to their advantage before a patch is available
- Advantages
  - Can force unresponsive software vendors to more promptly develop a patch

# Immediate Public Disclosure (cont'd)

- Advantages (cont'd)
  - Security personnel of potential victims are more alert than if they did not know – they can promptly work on countermeasures, and may develop effective ones before a patch is available
  - Incentivizes software vendors to develop secure software
- Evidence in favor of "more public" disclosures
  - i.e., disclose to more than just software vendor