Computer Security CS 426

Lecture 11

Software Vulnerabilities: Input Validation Issues & Buffer Overflows

Steps in a standard break-in (Getting in)

- Get your foot in the door
  - Steal a password file and run dictionary attack
  - Try to guess a password online
  - 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 (After Getting in)

- Set up some way to return
  - Install login program or web server with back door
- Cover your tracks
  - Disable intrusion detection, virus protection,
  - Install rootkits,
- Perform desired attacks
  - break into other machines
  - taking over the machine
  - Steal useful information (e.g., credit card numbers)

Common Software Vulnerabilities

- Input validation
- Buffer overflows
- Format string problems
- Integer overflows
- Race conditions…

Input Validation

- Sources of input
  - Command line arguments
  - Environment variables
  - Function calls from other modules
  - Configuration files
  - Network packets
- Sources of input for web applications
  - Web form input
  - Scripting languages with string input

Weak Input Validation

- What are some things that the attacker may try to achieve?
  - Crash programs
  - Execute arbitrary code
    - setuid or setgid programs
  - Obtain sensitive information

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 . (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")

Even more fun

- 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

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

What is needed to understand Buffer Overflow

- Understanding C functions and the stack.
- Some familiarity with machine code.
- Know how systems calls are made.
- The exec() system call.

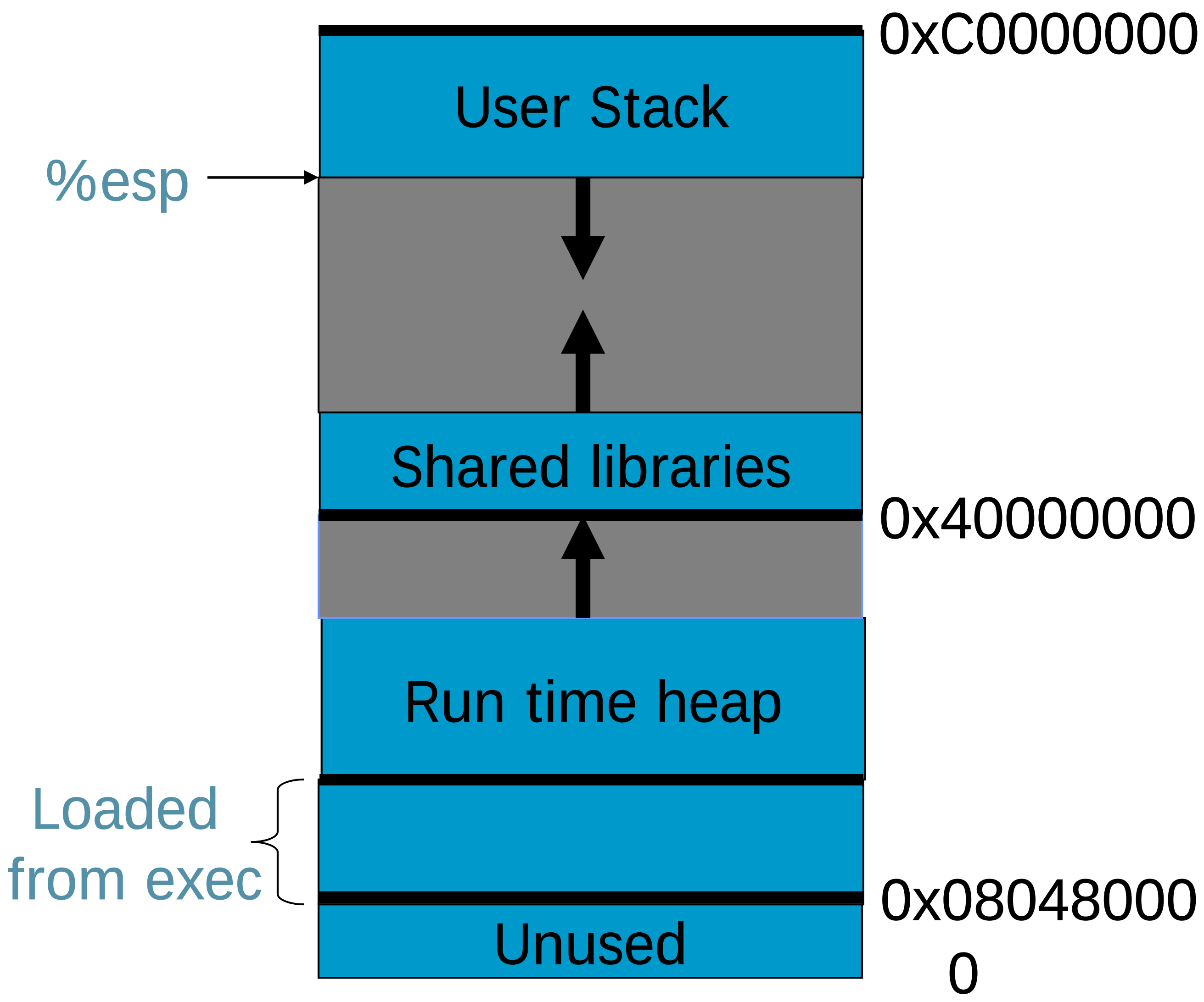
- Attacker needs to know which CPU and OS are running on the target machine.
  - Our examples are for x86 running Linux.
  - Details vary slightly between CPU's and OS:
    - Stack growth direction.
    - big endian vs. little endian.

Buffer Overflow

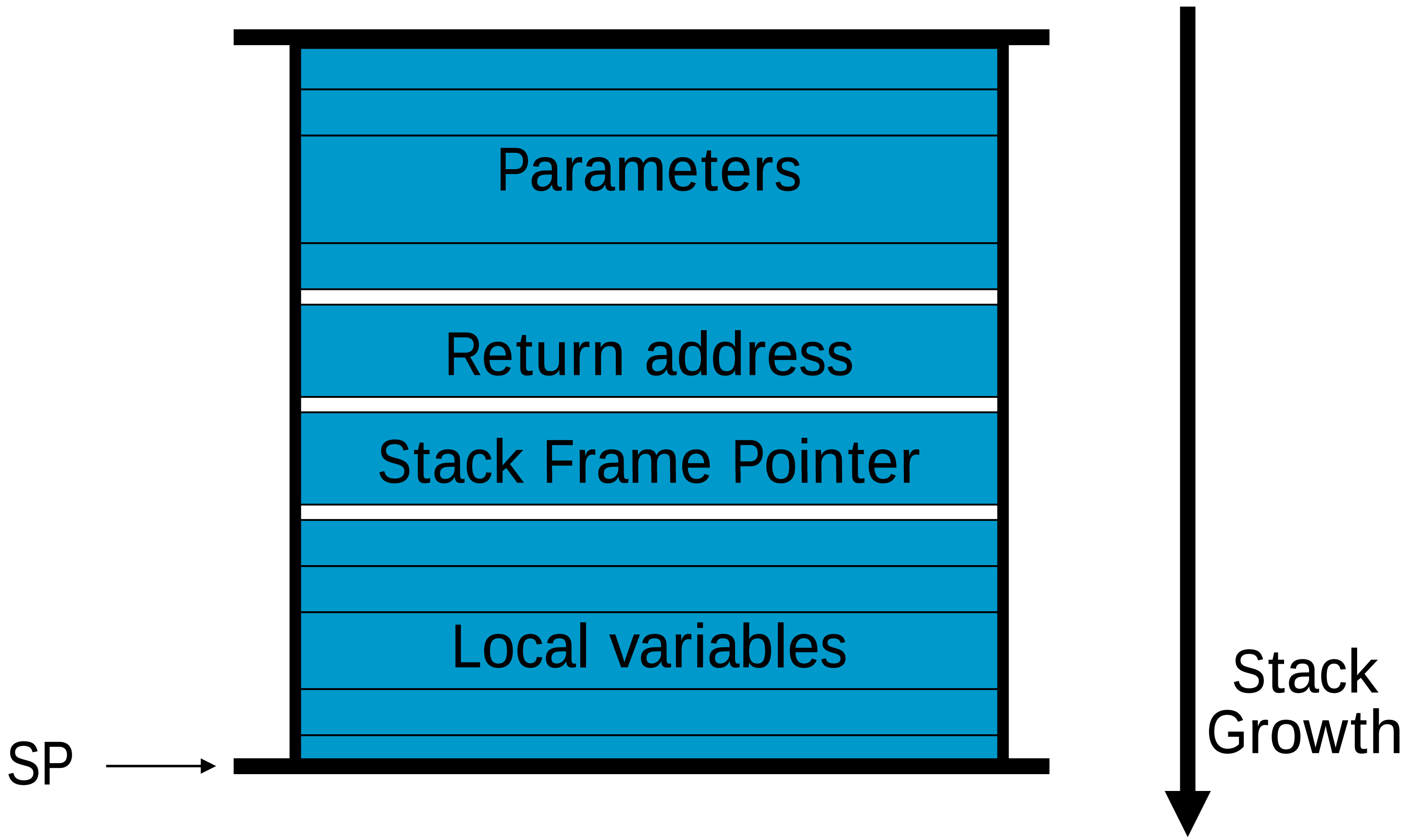
- Stack overflow
  - Shell code
  - Return-to-libc
    - Overflow sets ret-addr to address of libc function
  - Off-by-one
  - Overflow function pointers & longjmp buffers

- Heap overflow

Linux process memory layout



Stack Frame


Parameters
Return address
Stack Frame Pointer
Local variables
SP
Stack
Growth

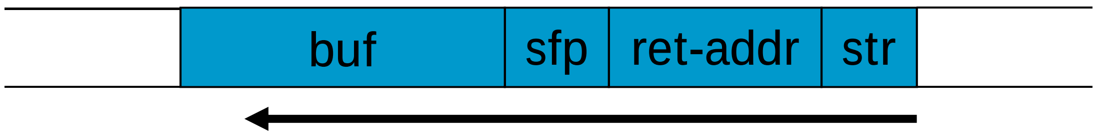
What are buffer overflows?

- Suppose a web server contains a function:

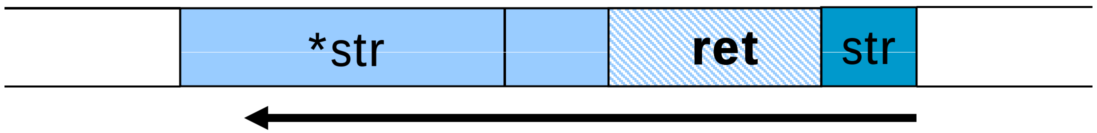
```
void func(char *str) {
  char buf[128];

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

- When the function is invoked the stack looks like:

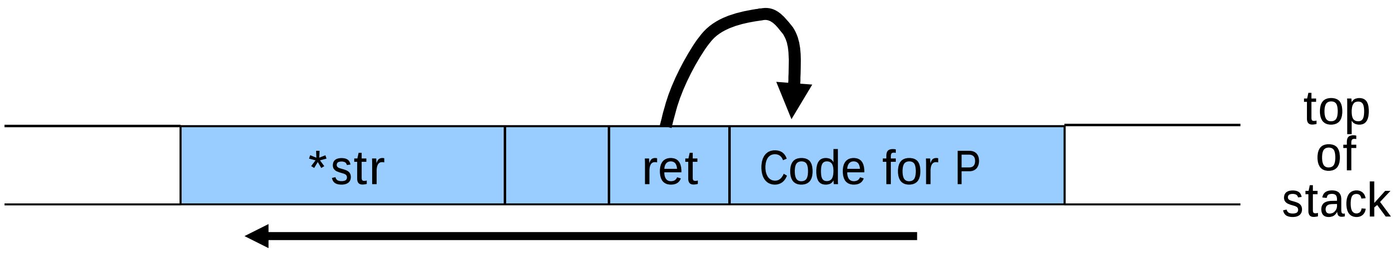


- What if `*str` is 136 bytes long? After `strcpy`:



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, …)

sprintf (conts char *format, …)

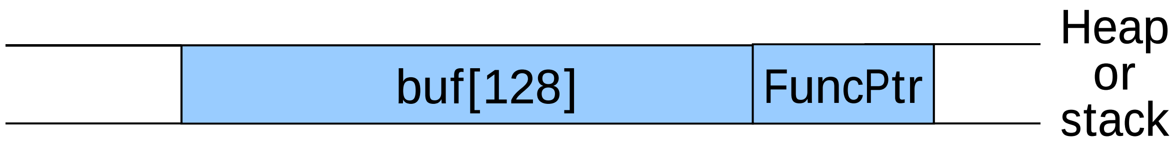
⋮

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 for stack overflows:
  - Program P should not contain the '\0' character.
  - Overflow should not crash program before func() exits.

Other 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)

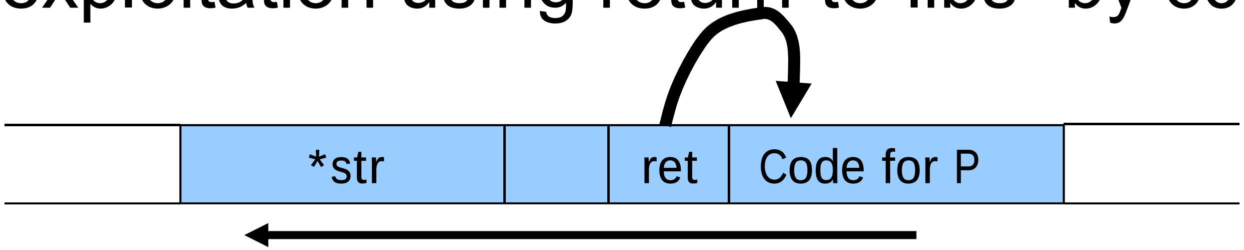


  - Overflowing buf will override function pointer.

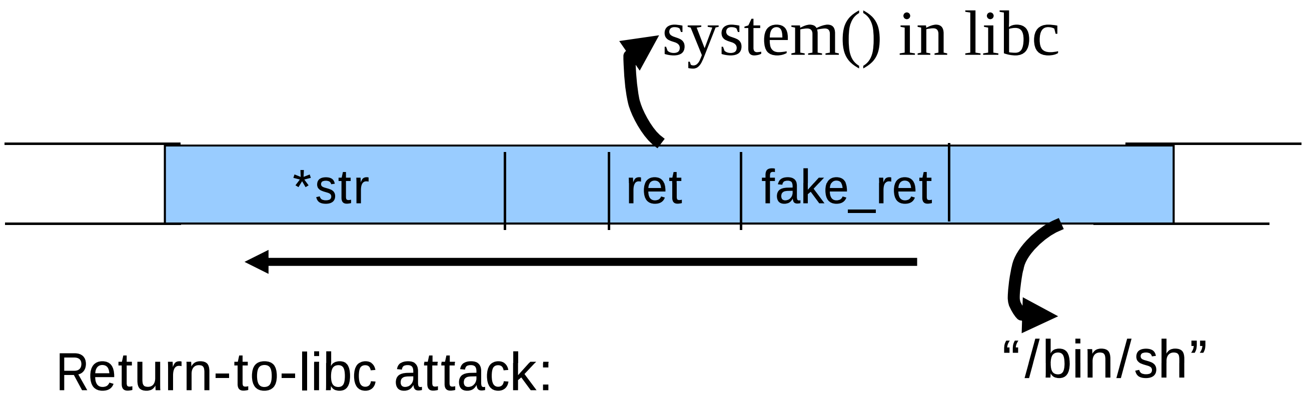
- Longjmp buffers: longjmp(pos) (used in attack on Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

return-to-libc attack

- "Bypassing non-executable-stack during exploitation using return-to-libs" by c0ntex



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



Return-to-libc attack:

Off by one buffer overflow

- Sample code

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

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 date 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.

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.
- 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.
  - Several consulting companies.
- Main idea: 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 (e.g. null ptr ref)
    - 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

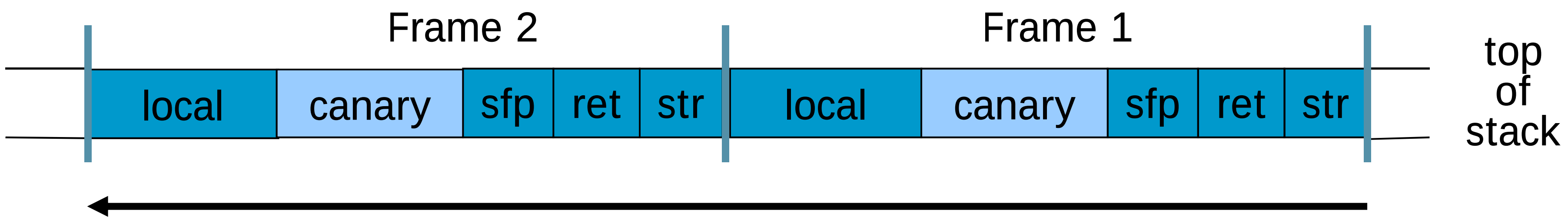
Marking stack as non-execute

- Basic stack exploit can be prevented by marking stack segment as non-executable.
  - Support in Windows SP2. Code patches exist for Linux, Solaris.

Problems:
  - Does not defend against `return-to-libc' exploit.
  - Some apps need executable stack (e.g. LISP interpreters).
  - Does not block more general overflow exploits:
    - Overflow on heap, overflow func pointer.

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 (Cont.)

- 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 don't offer fullproof protection.
  - Some stack smashing attacks can leave canaries untouched.

Randomization: Motivations.

- 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.
  - 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

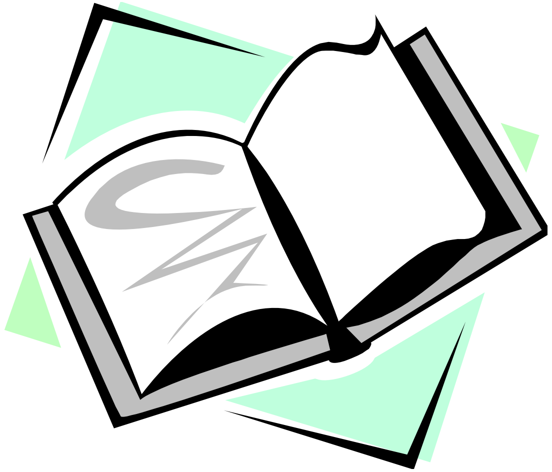
- Instruction Set Randomization (ISR)
  - Each program has a *different* and *secret* instruction set
  - Use translator to randomize instructions at load-time
  - Attacker cannot execute its own code.
- What constitutes instruction set depends on the environment.
  - for binary code, it is CPU instruction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

Instruction Set Randomization

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

Readings for This Lecture

- Wikipedia
  - **Privilege escalation**
  - **Buffer overflow**
  - **Stack buffer overflow**
  - **Buffer overflow protection**



Coming Attractions …

- Other Software vulnerabilities