


PURDUE UNIVERSITY

Computer Security

CS 426

Lecture 8

Program Security

 CERIAS
Center for Education and Research
in Information Assurance and Security


Elisa Bertino
Purdue University
IN, USA
bertino@cs.purdue.edu

1

PURDUE UNIVERSITY

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

- An unexpected behavior is called *program security flaw*

2

PURDUE UNIVERSITY

Common Software Vulnerabilities

- Buffer overflows
- Input validation
- Format string problems
- Integer overflows
- Failing to handle errors
- Other exploitable logic errors

3

PURDUE UNIVERSITY

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

4

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



5

Why Buffer Overflow Matters

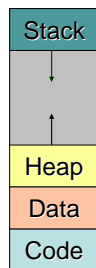
- Overwrites:
 - other buffers
 - variables
 - program flow data
- Results in:
 - erratic program behavior
 - a memory access exception
 - program termination
 - incorrect results
 - **breach of system security**

6

Programs and Memory

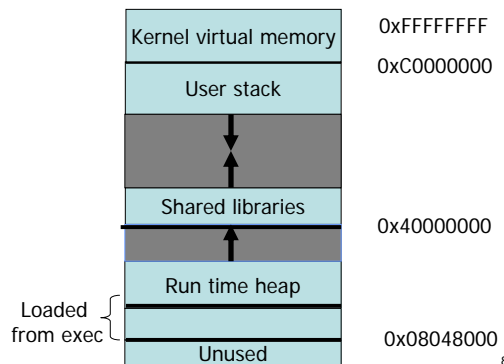
- The operating system creates a process by assigning memory and other resources
- **Stack**: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions
- **Heap**: dynamic memory for variables that are created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

Virtual Memory



7

Example: Linux Process Memory Layout



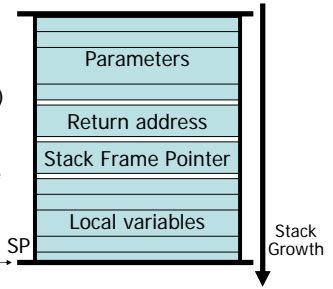
8

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



Buffer Overflow Attacks

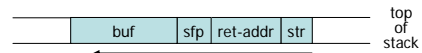
- Extremely common attack
 - First major exploit: 1988 Internet Worm. fingerd.
- 15 years later: $\approx 50\%$ of all CERT advisories:
 - 1998: 9 out of 13
 - 2001: 14 out of 37
 - 2003: 13 out of 28
- **May lead to total compromise of host or a crash**
- Developing buffer overflow attacks:
 - Locate buffer overflow within an application.
 - Design an exploit.

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**:



Findings Buffer Overflow

- One strategy:
 - Run web server on local machine.
 - Issue requests with long tags ending with “\$\$\$\$\$”.
 - If web server crashes, search core dump for “\$\$\$\$\$” to find overflow location.
- Use automated tools: eEye Retina, ISIC.



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, ... )
```

⋮

Stack-based Buffer Overflow Attacks

- Overwrite **the return address** in a stack frame. Once the function returns, execution will resume at the return address as specified by the attacker, usually a user input filled buffer
- Overwrite **a function pointer**, or exception handler, which is subsequently executed
- Overwrite **a local variable** that is near the buffer in memory on the stack to change the behavior of the program

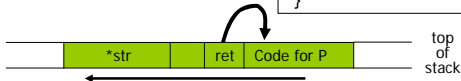
Control Hijacking Opportunities

- 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.



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.

Exploiting Buffer Overflows

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

More Examples

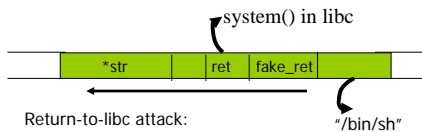
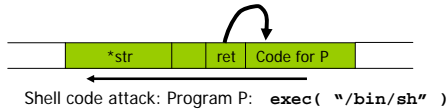
(All fixed)

- MSFT indexing service, an extension to IIS
 - telnet <site> 80
 - GET /somefile.idq?<long buffer>
 - Telnet to port 80 and send http GET with buffer over 240 bytes
 - Attacker can take over server
 - Form of attack used by Code Red to propagate
- TFTP server in Cisco IOS
 - Use overflow vulnerability to take over server (long filename)
- MS Xbox
 - James Bond 007 game has a save game option
 - Code to restore game has buffer overflow vulnerability
 - Can boot linux or run other code using game as "boot loader"

Many, many more examples

return-to-libc attack

- "Bypassing non-executable-stack during exploitation using return-to-libcs" by context

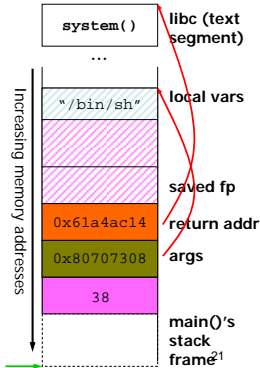


Return-to-libc Attacks

- 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.



Heap-based Buffer Overflow Attacks

- Remember that heap represents 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.

Preventing Buffer Overflow Attacks

- Static source code analysis
- Run time checking: StackGuard, Libsafe, SafeC, (Purify)
- Type safe languages (Java, ML)
- Non-executable stack
- Randomization
- Detection deviation of program behavior
- Sandboxing
- Access control
- Deep packer inspection

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.

Defenses against BO

PURDUE UNIVERSITY

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

25

Defenses against BO

PURDUE UNIVERSITY

Run Time Checking: StackGuard

- Run time tests for stack integrity.
- Embed “canaries” in stack frames and verify their integrity prior to function return.

top of stack

Canaries or canary words are known values that are placed between a buffer and control data on the stack to monitor buffer overflows. When the buffer overflows, the first data to be corrupted will be the canary, and a failed verification of the canary data is therefore an alert of an overflow, which can then be handled, for example, by invalidating the corrupted data.

26

Defenses against BO

PURDUE UNIVERSITY

StackGuard: 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.

27

Defenses against BO

PURDUE UNIVERSITY

StackGuard: 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.

28



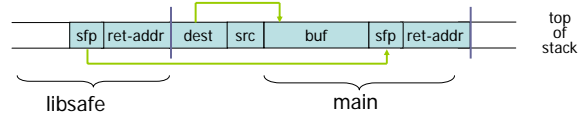
StackGuard: Implementation

- 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 don't 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: $|frame\text{-}pointer - dest| > strlen(src)$
 - 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)



Non-Executable Stack

- Basic stack exploit can be prevented by hardware support to mark stack segment as non-executable (parts of memory are for data only).
 - Support in SP2. Code patches exist for Linux, Solaris.
- Problems:
- Does not defend against all attacks (see 'return-to-libc')
 - 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 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



33

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

34

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

35

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

36

Readings for This Lecture

- **Smashing The Stack For Fun And Profit by Aleph One**
(<http://doc.bughunter.net/buffer-overflow/smash-stack.html>)
- http://en.wikipedia.org/wiki/Buffer_overflow
- Textbook, Section 3.2