Buffer Overflow

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
IA-32/Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)

- **Heap**
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()

- **DLLs (shared libraries)**
  - Dynamically linked libraries
  - Library routines (e.g., printf, gets)
  - Linked into object code when first executed

- **Data**
  - Statically allocated data
  - e.g., arrays & strings declared in code

- **Text**
  - Executable machine instructions
  - Read-only
(gdb) break main
(gdb) run
    Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
    $3 = (void *) 0xbfffffc78
/* Echo Line */
void echo()
{
    // Way too small!
    char buf[4];
    gets(buf);
    puts(buf);
}

int main()
{
    printf("Type: ");
    echo();
    return 0;
}
String Library Code

- Implementation of Unix function gets()
  - No way to specify limit on # of characters to read

```c
/* Get string from stdin */
char *gets(char *dest) {
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other Unix functions
  - `strcpy`: copies string of arbitrary length
  - `scanf/fscanf/sscanf`, given `%s` conversion specification
Buffer Overflow (1)

```c
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

- Stack Frame for `main`
  - Return Address
  - Saved `%ebp`:
    - `%ebp`:
      - `buf`:
    - `%esp`
      - `%eax`
      - Stack Frame for `gets`

```
echo:
    pushl %ebp          # Save `%ebp` on stack
    movl %esp,%ebp
    subl $24,%esp       # Allocate space
    leal -4(%ebp),%eax  # compute `buf` as `%ebp-4`
    movl %eax,(%esp)    # save to the stack
    call gets           # Call `gets`
    ...
```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbfffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d

8048648: call 804857c <echo>
804864d: mov $0,%eax      # Return Point
Buffer Overflow (3)

Before Call to `gets`

- Stack Frame for `main`
  - Return Address
  - Saved `%ebp`
  - `%ebp` buf
  - Stack Frame for `echo`

Input = “123”

- Stack Frame for `main`
  - `%ebp` buf
  - `0xbfffff8f8`
  - `00 33 32 31`

No Problem
Buffer Overflow (4)

Input = “12345”

Stack Frame for main

Return Address
Saved %ebp
buf

Stack Frame for echo

Stack Frame for main

08 04 -86 4d
bf ff 00 35
34 33 32 31

0xbffff8d8
buf
Saved value of %ebp set to 0xbffff0035

Bad news when later attempt to restore %ebp

...<echo code>...

8048600: call 80482c4  # gets
8048605: leal 0xffffffffc(%ebp),%eax
8048608: movl %eax,(%esp)
804860b: call 80482d4  # puts
8048610: leave
            # movl %ebp, %esp; popl %ebp
8048611: ret
Buffer Overflow (5)

Input = “12345678”

Stack Frame for main

Stack Frame for echo

Stack Frame for echo

%ebp

buf

Invalid address

No longer pointing to desired return point

%ebp and return address corrupted

8048648: call 804857c <echo>
804864d: mov $0,%eax  # Return Point
**Buffer Overflow Attack (1)**

- **Malicious use of buffer overflow**
  - Input string contains byte representation of executable code
  - Overwrite return address with address of buffer
  - When `bar()` executes `ret`, will jump to exploit code.

```c
void foo(){
    bar();
    ...
}

void bar() {
    char buf[64];
    gets(buf);
    ...
}
```

Diagram:
- `A` is the return address.
- `B` is the data written by `gets()`.
- The `bar` stack frame.
- The `foo` stack frame.

Stack after call to `gets()`.
Buffer Overflow Attack (2)

- **Exploits based on buffer overflows**
  - Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.
  - Internet worm
    - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
      - `finger kildong@skku.edu`
    - Worm attacked fingerd server by sending phony argument:
      - `finger "exploit-code padding new-return-address"`
      - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
Code Red Worm (1)

- **History**
  - White house must change its IP address. Pentagon shut down public WWW servers for day.

- **Received strings of form**

  ```
  GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a
  HTTP/1.0" 400 325 "-" 
  ```
Code Red Worm (2)

- **Code Red exploit code**
  - Starts 100 threads running
  - Spread self
    - Generate random IP addresses & send attack string
    - Between 1\textsuperscript{st} & 19\textsuperscript{th} of month
  - Denial of service attack to www.whitehouse.gov
    - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Between 21\textsuperscript{st} & 27\textsuperscript{th} of month
  - Deface server’s home page
    - After waiting 2 hours
**Code Red Worm (3)**

- **Code Red effects**
  - Later version even more malicious
    - Code Red II
    - As of April 2002, over 18,000 machines infected
    - Still spreading
  - Paved way for NIMDA
    - Variety of propagation methods
    - One was to exploit vulnerabilities left behind by Code Red II
Nimda Worm

Nimda (2001)

- Five different infection methods:
  - Via e-mail
  - Via open network shares
  - Via browsing of compromised web sites
  - Exploitation of various Microsoft IIS 4.0/5.0 directory traversal vulnerabilities
  - Via back doors left behind by the "Code Red II" and "Sadmind/IIS" worms

- One of the most widespread virus/worm
SQL Slammer Worm

- **SQL slammer (2003)**
  - Exploited two buffer overflow bugs in Microsoft’s SQL Server and Desktop Engine.
  - Infected 75,000 victims within 10 minutes
  - Generate random IP addresses and send itself out to those addresses, slowing down Internet traffic dramatically.
  - 1/25 nationwide Internet shutdown in South Korea
Avoiding Buffer Overflow

- Use library routines that limit string lengths
  - `fgets()` instead of `gets()`
  - `strncpy()` instead of `strcpy()`
  - Don’t use `scanf()` with `%s` conversion specification
    - Use `fgets()` to read the string
    - Or use `%ns` where `n` is a suitable integer

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```
System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code

- Executable space protection
  - Mark certain areas of memory as non-executable
  - Hardware assistance:
    - Intel NX (No eXecute) bit
    - AMD XD (eXecute Disable) bit
Summary

- **Memory layout**
  - OS/machine dependent (including kernel version)
  - Basic partitioning:
    - stack, data, text, heap, DLL found in most machines

- **Avoiding buffer overflow vulnerability**
  - Important to use library routines that limit string lengths

- **Working with strange code**
  - Important to analyze nonstandard cases
    - e.g., what happens when stack corrupted due to buffer overflow
  - Helps to step through with GDB