Software Vulnerabilities and Exploits

Brad Karp
UCL Computer Science

CS GZ03 / M030
9th December, 2009
Imperfect Software

• To be useful, software must process input
  – From files, network connections, keyboard...
• Programmer typically intends his code to manipulate input in particular way
  – e.g., parse HTTP request, retrieve matching content, return it to requestor
• Programs are complex, and often include subtle bugs unforeseen by the programmer
• Fundamentally hard to prevent all programmer error
  – Design itself may use flawed logic
  – Even formal reasoning may not capture all ways in which program may deviate from desired behavior
  – Remember: security is a negative goal...
Imperfect Software (2)

• Even if logic correct, implementation may vary from programmer intent

• C and C++ particularly dangerous
  – Allow arbitrary manipulation of pointers
  – Require programmer-directed allocation and freeing of memory
  – Don’t provide memory safety; very difficult to reason about which portions of memory a line of C changes
  – Offer high performance, so extremely prevalent, especially in network servers and OSes

• Java offers memory safety, but not a panacea
  – JRE written in (many thousands of lines of) C!
Software Vulnerabilities and Exploits

- **Vulnerability**: broadly speaking, input-dependent bug that can cause program to complete operations that deviate from programmer’s intent
- **Exploit**: input that, when presented to program, triggers a particular vulnerability
- Attacker can use exploit to execute operations without authorization on vulnerable host
- Vulnerable program executes with some privilege level
  - Many network servers execute as superuser
  - Users run applications with their own user ID
  - Result: great opportunity for exploits to do harm
Software Vulnerabilities and Exploits

• **Vulnerability:** broadly speaking, input-dependent bug that can cause program to complete operations that deviate from programmer’s intent

• **Exploit:** input that, when presented to program, triggers a particular vulnerability

• Attacker can use exploit to execute operations without authorization on vulnerable host

• Vulnerable program executes with some privilege level
  – Many network servers execute as superuser
  – Users run applications with their own user ID
  – Result: great opportunity for exploits to do harm

Today: vulnerabilities in C programs that allow an attacker to execute his own arbitrary code within the vulnerable program without authorization on vulnerable host
Buffer Overflows in C: General Idea

- Buffers (arrays) in C manipulated using pointers
- C allows arbitrary arithmetic on pointers
  - Compiler has no notion of size of object pointed to
  - So programmers must explicitly check in code that pointer remains within intended object
  - But programmers often do not do so; vulnerability!
- Buffer overflows used in many exploits:
  - Input long data that runs past end of programmer’s buffer, over memory that guides program control flow
  - Enclose code you want executed within data
  - Overwrite control flow info with address of your code!
Memory Map of a UNIX Process

- **Text**: executable instructions, read-only data; size fixed at compile time
- **Data**: initialized and uninitialized; grows towards higher addresses
- **Stack**: LIFO, holds function arguments and local variables; grows toward lower addresses
Intel X86 Stack: Stack Frames

• Region of stack used within C function: stack frame
• Within function, local variables allocated on stack
• SP register: stack pointer, points to top of stack
• BP register: frame pointer (aka base pointer), points to bottom of stack frame of currently executing function
Intel X86 Stack: Calling and Returning from Functions

• To call function $f()$, allocate new stack frame:
  – Push arguments, e.g., $f(a, b, c)$
  – Push return address: next instruction (IP) in caller
  – Set IP = address of $f()$; jump to callee
  – Push saved frame pointer: BP for caller’s stack frame
  – Set BP = SP; sets frame pointer to start of new frame
  – Set SP -= sizeof(locals); allocates local variables

• Upon return from $f()$, deallocate stack frame:
  – Set SP += sizeof(locals); deallocates local variables
  – Set BP = saved frame pointer from stack; change to caller’s stack frame
  – Set IP = saved return address from stack; return to next instruction in caller
Example: Simple C Function Call

```c
void dorequest(int a, int b) {
    char request[256];
    scanf("%s", request);
    /* process the request... */
    ...
    return;
}

int main(int argc, char **argv) {
    while (1) {
        dorequest(17, 38);
        fprintf(log, "completed\n");
    }
}
```
### Stack Smashing Exploits: Basic Idea

- Return address **stored on stack** directly influences program control flow
- Stack frame layout: local variables allocated **just before return address**
- If programmer allocates buffer as local on stack, reads input, and writes it into buffer without checking input fits in buffer:
  - Send input containing **shellcode** you wish to run
  - Write past end of buffer, and overwrite return address with **address of your code within stack buffer**
  - When function returns, **your code executes!**
Example: Stack Smashing

```c
void dorequest(int a, int b) {
    char request[256];
    scanf("%s", request);
    /* process the request... */
    ...
    return;
}

int main(int argc, char **argv) {
    while (1) {
        dorequest(17, 38);
        fprintf (log, "completed\n");
    }
}
```

Example: Stack Smashing

- **Malicious Input**: The input can be crafted to overwrite the return address on the stack.
- **Shell Code**: The shell code is executed after the request is processed.
- **Increasing Memory Addresses**: The memory addresses increase as the function calls proceed.
- **Local Vars**: The local variables are stored on the stack.
- **Saved FP**: The saved frame pointer is used to restore the original state after function calls.
- **Return Addr**: The return address is overwritten by the malicious input.
- **Main()’s Stack Frame**: The stack frame is shown with the original return address at the top.
void dorequest(int a, int b) {
    char request[256];

    scanf("%s", request);
    request...
    return;
}

int main(int argc, char **argv) {
    while (1) {
        dorequest(17, 38);
        fprintf(log, "completed\n");
    }
}

Example: Stack Smashing

void dorequest(int a, int b) {
    char request[256];

    scanf("%s", request);
    request... */
    return;
}

int main(int argc, char **argv) {
    while (1) {
        dorequest(17, 38);
        fprintf (log, "completed\n");
    }
}

Example: Stack Smashing

void dorequest(int a, int b) {
    char request[256];

    scanf("%s", request);
    request... */
    return;
}

int main(int argc, char **argv) {
    while (1) {
        dorequest(17, 38);
        fprintf (log, "completed\n");
    }
}
Designing a Stack Smashing Exploit

- In our example, attacker had to know:
  - existence of stack-allocated buffer without bounds check in program
  - exact address for start of stack-allocated buffer
  - exact offset of return address beyond buffer start
- Hard to predict these exact values:
  - stack size before call to function containing vulnerability may vary, changing exact buffer address
  - attacker may not know exact buffer size
- Don’t need to know either exact value, though!
Designing a Stack Smashing Exploit (2)

- No need to know exact return address:
  - Precede shellcode with **NOP slide**: long sequence of NOPs (or equivalent instructions)
  - So long as **jump into NOP slide**, shellcode executes
  - Effect: range of return addresses works

- No need to know exact offset of return address beyond buffer start:
  - **Repeat shellcode’s address many times** in input
  - So long as first instance occurs before return address’s location on stack, and enough repeats, will overwrite it
void dorequest(int a, int b) {
    char request[256];
    
    scanf("%s", request);
    /* process the request... */
    ...
    return;
}

int main(int argc, char **argv) {
    while (1) {
        dorequest(17, 38);
        fprintf(log, "completed\n");
    }
}

Example: Stack Smashing “2.0”
Designing Practical Shellcode

• Shellcode normally executes /bin/sh; gives attacker a shell on exploited machine
• shellcode.c:

```c
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    exit(0); /* if execve fails, don’t */
}  /* dump core */
```
Designing Practical Shellcode (2)

• Compile shellcode.c, disassemble in gdb to get hex representation of instructions

• Problem: to call execve(), must know exact address of string “/bin/sh” in memory (i.e., within stack buffer)
  – Difficult to predict, as before
Both jmp and call instructions allow IP-relative addressing
- Specify target by offset from current IP, not by absolute address

Finding absolute address of “/bin/sh” at runtime:
- add call instruction at end of shellcode, with target of first shellcode instruction, using relative addressing
- place “/bin/sh” immediately after call instruction
- call will push next “instruction’s” address onto stack
- precede first shellcode instruction with jmp to call, using relative addressing
- after call, stack will contain address of “/bin/sh”
Practical Shellcode Example

```assembly
jmp 0x2a # 3 bytes
popl %esi # 1 byte
movl %esi,0x8(%esi) # 3 bytes
movb $0x0,0x7(%esi) # 4 bytes
movl $0x0,0xc(%esi) # 7 bytes
movl $0xb,%eax # 5 bytes
movl %esi,%ebx # 2 bytes
leal 0x8(%esi),%ecx # 3 bytes
leal 0xc(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1, %eax # 5 bytes
movl $0x0, %ebx # 5 bytes
int $0x80 # 2 bytes
call -0x2f # 5 bytes
.string "/bin/sh" # 8 bytes
```

Pops string address from stack!

Writes string address on stack!
Eliminating Null Bytes in Shellcode

• Often vulnerability copies string into buffer
• C marks end of string with zero byte
  – So functions like strcpy() will stop copying if they encounter zero byte in shellcode instructions!
• Solution: replace shellcode instructions containing zero bytes with equivalent instructions that don’t contain zeroes in their encodings
Defensive Coding to Avoid Buffer Overflows

• Always explicitly check input length against target buffer size

• Avoid C library calls that don’t do length checking:
  – e.g., `sprintf(buf, ...), scanf(“%s”, buf), strcpy(buf, input)`

• Better:
  – `snprintf(buf, buflen, ...),
    scanf(“%256s”, buf),
    strncpy(buf, input, 256)`
Overview: Format String Vulnerabilities and Exploits

• Recall C’s printf-like functions:
  – `printf(char *fmtstr, arg1, arg2, ...)`
  – e.g., `printf("%d %d", 17, 42);`
  – Format string in 1st argument specifies number and type of further arguments

• Vulnerability:
  – If programmer allows input to be used as format string, attacker can force printf-like function to overwrite memory
  – So attacker can devise exploit input that includes shellcode, overwrites return address...
Background: \%n Format String Specifier

- \texttt{\%n} format string specifier directs printf to write number of bytes written thus far into the integer pointed to by the matching int * argument

- Example:
  
  ```c
  int i;
  printf("foobar\%n\n", (int *) &i));
  printf("i = \%d\n", i);
  ```

- Output:
  
  ```
  foobar
  i = 6
  ```
Abusing `printf` to Overwrite Memory

- `printf`'s caller often allocates format string buffer on stack
- C pushes parameters onto stack in right-to-left order
  - format string pointer on top of stack, last arg on bottom
- `printf()` increments pointer to point to successive arguments
  
  [suppose input = "%d%d%d\n"]
  
  ```c
  char fmt[16];
  strncpy(fmt, input, 15);
  printf(fmt, 1, 2, 3);
  ```
Abusing %n to Overwrite Memory (2)

- Idea:
  - Use specifiers in format string to increment printf()’s arg pointer so it points to format string itself
  - Supply target address to write at start of format string
  - Supply “%n” at end of format string

[input = "\xc0\xc8\xff\xbf\%08x\%08x\%08x\%08x\%n"]
char fmt[16];
strncpy(fmt, input, 15);
printf(fmt, 1, 2, 3);
Abusing `%n` to Overwrite Memory (2)

- Idea:
  - Use specifiers in format string to increment printf()'s arg pointer so it points to format string itself
  - Supply target address to write at start of format string
  - Supply “%n” at end of format string

[input = "\xc0\xc8\xff\xbf%08x%08x%08x%08x%08x%n"]

char fmt[16];
strncpy(fmt, input, 15);
printf(fmt, 1, 2, 3);

Result: can overwrite chosen location with small integer
Still need to choose value we overwrite with…
Controlling Value Written by %n

- %n writes number of bytes printed
- But number of bytes printed controlled by format string!
  - Format specifiers allow indication of exactly how many characters to output
  - e.g., “%20u” means “use 20 digits when printing this unsigned integer”
- So we can use “%[N]u%n” format specifier to set least significant byte of target address to value [N]!
Example: Using %[N]u%n

- Example format string:
  "[spop]\x01\x01\x01\x01\xc0\xc8\xff\xbf%50u%n"
- [spop] is sequence of "%08x" values, to advance printf ()'s arg pointer to first byte after [spop]
- \x01\x01\x01\x01 is dummy integer, to be consumed by %50u
- \xc0\xc8\xff\xbf is address of integer whose least significant byte will be changed by %n
- %50u sets number of output bytes to 50 (0x32)
- %n writes number of output bytes to target address
- Result: least significant byte of 4-byte value at 0xbfff8c0 overwritten with number of bytes printed total: 0x32 + 0x08 + [bytes printed by spop]
Overwriting Full 4-Byte Values

• Template for format string:
  
  [4 non-zero bytes (dummy int)]
  [4 bytes target address]
  [dummy int][4 bytes (target address + 1)]
  [dummy int][4 bytes (target address + 2)]
  [dummy int][4 bytes (target address + 3)]
  [spop]
  %[1st byte value to write]u%n
  %[2nd byte value to write]u%n
  %[3rd byte value to write]u%n
  %[4th byte value to write]u%n

• N.B. LSB always in lowest memory address
  (Intel is little-endian)
Overwriting 4-Byte Values (2)

- Counter for %n is **cumulative**
- But **only least significant byte written matters**
- Say %n count is x so far, want next overwritten byte to have value y
- **Next %u should be %[N]u, where:**

  \[
  N = (0x100 + y - (x \mod 0x100)) \mod 0x100
  \]

  if \(N < 10\)

  \[
  N += 0x100
  \]
Format String Vulnerabilities Are Real and Versatile

• Example: `wu-ftpd <= 2.6.0`:

```c
{
    char buffer[512];
    snprintf (buffer, sizeof (buffer), user);
    buffer[sizeof (buffer) - 1] = '\0';
}
```

• Ability to overwrite arbitrary memory makes format string vulnerabilities versatile:
  – Sure, can overwrite return address to return to shellcode, but other ways to attack, too
  – If server contains “superuser” flag (0 or 1), just overwrite that flag to be 1...
• More scrutiny of software than ever
• Little progress in producing vulnerability-free software
Disclosure and Patching of Vulnerabilities

- Software vendors and open-source developers audit code, *release vulnerability reports*
  - Usually describe vulnerability, but don’t give exploit
  - Often include announcement of *patch*

- Race after disclosure: users patch, attackers devise exploit
  - Users often lazy or unwilling to patch; “patches” can *break software*, or include *new vulnerabilities*

- Attackers prize exploits for undisclosed vulnerabilities: *zero-day exploits*

- Disclosure best for users: can *patch or disable*, vs. risk of *widest harm by zero-day exploit*
Summary

• Many categories of vulnerabilities in C/C++ binaries; 2 we’ve seen hardly exhaustive
• Incentives for attackers to find vulnerabilities and design exploits are high
  – Arbitrary code injection allows:
    • Defacing of widely viewed web site
    • Stealing valuable confidential data from server
    • Destruction of data on server
    • Recruitment of zombies to botnets (spam, DoS)
  – Market in vulnerabilities and exploits!
• Preventing all exploits extremely challenging
  – Stopping one category leads attackers to use others
  – New categories continually arising