CS390S, Week 5: Format String Vulnerabilities & Integer Overflows

Pascal Meunier, Ph.D., M.Sc., CISSP
February 6, 2008
Developed thanks to support and contributions from Symantec Corporation, support from the NSF SFS Capacity Building Program (Award Number 0113725) and the Purdue e-Enterprise Center

Copyright (2004) Purdue Research Foundation. All rights reserved.
Note on Return Into libC (Buffer Overflow)

- Point return address to a library function
  - e.g., “system”
    - Execute a program based on string argument
- Put arguments for the function on the stack
  - As if they had been pushed for a regular function call
- Library function thinks it was just called
  - Later returns
    - “ret” instruction can be made to use another address on stack
      - Chain calls to construct program!
Format String Issues: Outline

- Introduction to format strings
- Mostly a "C" problem
- Examples
- Definition
- Importance
- Survey of unsafe functions
- Case study: analysis of cfingerd 1.4.3 vulnerabilities
- Preventing format string vulnerabilities without programming
- Tools to find string format issues
What is a Format String?

- Encode where and how to print variables
- Python example:

```python
>>> print "Mary has %d lambs" % (3)
Mary has 3 lambs
```

- `%d` specifies a decimal number (from an int)
- `%s` would specify a string argument,
- `%X` would specify an unsigned uppercase hexadecimal (from an int)
- etc...
Resource Consumption

>>> print "Mary has %99s lambs" % ("pretty")
Mary has

    pretty lambs

>>> 

• Fill logs, memory (who choses the format string?)

>>> print "Mary has %999999s lambs" % ("pretty")
Mary has
Fundamental "C" Problem

- No way to count arguments passed to a "C" function, so missing arguments are not detected
  - printf(const char *format, ...);
  - printf("Mary has %d cats", cats);
- What happens if the following code is run?
  - int main () {
    printf("Mary has %d cats");
    return 0;
  }
Result

- % ./a.out
  Mary has -1073742416 cats
- Format string is interpreted: it blinds the difference between code and data
- Program reads missing arguments off the stack!
  - And gets garbage (or interesting stuff if you want to probe the stack)
Probing the Stack

- Read values off stack
- Confidentiality violations
- `printf("\%08X\")`
  - `x (X)` is unsigned hexadecimal
  - 0: with ‘0’ padding
  - 8 characters wide: ‘0XAA03BF54’
  - 4 bytes = pointer on stack, canary, etc...
What happens if the following code is run, assuming there always is an argument input by a user?

```c
int main(int argc, char *argv[])
{
    printf(argv[1]);
    exit(0);
}
```

Try it and input "%s%s%s%s%s%s%s%s%s%s%s%s%s"
How many "%s" arguments do you need to crash it?
Result

- `./a.out "%s%s%s%s%s%s"
  Bus error

- Program was terminated by OS
  - Segmentation fault, bus error, etc... because the program attempted to read where it wasn't supposed to

- User input is interpreted as string format (e.g., %s, %d, etc...)

- Anything can happen, depending on input!

- How would you correct the program?
Corrected Program

- int main(int argc, char *argv[]) {
  printf("%s", argv[1]);
  exit(0);
}
- % ./a.out "%s%s%s%s%s%s"
  %s%s%s%s%s%s%s
Format String Vulnerabilities

- Discovered relatively recently ~2000
  - “Uncontrolled Format String” CWE ID 134
- Limitation of “C” family languages
- Versatile
  - Can affect various memory locations
  - Can be used to create buffer overflows
  - Can be used to read the stack
- Not straightforward to exploit, but examples of root compromise scripts are available on the web
  - "Modify and hack from example"
Definition of a Format String Vulnerability

- A call to a function with a format string argument, where the format string is either:
  - Possibly under the control of an attacker
  - Not followed by the appropriate number of arguments

- As it is difficult to establish whether a data string could possibly be affected by an attacker, it is considered very bad practice to place a string to print as the format string argument.
  - Sometimes the bad practice is confused with the actual presence of a format string vulnerability
How Important Are Format String Vulnerabilities?

- Search NVD for “format string”:
  - 115 records in 2002
  - 173 total in April 2004
  - 363 in February 2006
  - 460 in January 2007
  - 577 in February 2008 (~100/year)

- Various applications
  - Databases (Oracle)
  - Unix services (syslog, ftp,...)
  - Linux “super” (for managing setuid functions)
  - cfingerd CVE-2001-0609

- Arbitrary code execution is a frequent consequence
Functions Using Format Strings

- `printf` - prints to "stdout" stream
- `fprintf` - prints to stream
- `warn` - standard error output
- `err` - standard error output
- `setproctitle` - sets the invoking process's title
- `sprintf(char *str, const char *format, ...)``
  - `sprintf` prints to a buffer
  - What’s the problem with that?
Sprintf Double Whammy

- format string AND buffer overflow issues!
- Buffer and format string are usually on the stack
- Buffer overflow rewrites the stack using values in the format string
Better Functions Than sprintf

- Note that these don't prevent format string vulnerabilities:
  - snprintf(char *str, size_t size, const char *format, ...);
    - sprintf with length check for "size"
  - asprintf(char **ret, const char *format, ...);
    - sets *ret to be a pointer to a buffer sufficiently large to hold the formatted string (note the potential memory leak).
Custom Functions Using Format Strings

- It is possible to define custom functions taking arguments similar to printf.
- wu-ftpd 2.6.1 proto.h
  - void reply(int, char *fmt,...);
  - void lreply(int, char *fmt,...);
  - etc...
- Can produce the same kinds of vulnerabilities if an attacker can control the format string.
"%n" format command

Writes a number to the location specified by argument on the stack

– Argument treated as int pointer
  – Often either the buffer being written to, or the raw input, are somewhere on the stack
    – Attacker controls the pointer value!
  – Writes the number of characters written so far
    – Keeps counting even if buffer size limit was reached!
    – “Count these characters %n”

All the gory details you don't really need to know:

– Newsham T (2000) "Format String Attacks"
String Format Vulnerabilities in Libraries

- Perl code wrapper around syscall(3) function
- Resulted in CVE-2005-3912
- Wrapper didn't perform appropriate input validation
  - hole in Perl's safe handling of format strings

- eval { syslog("info", $msg); };
  # wrong

- eval { syslog("info", "%s", $msg); };
  # safe
Preventing Format String Vulnerabilities

1) Always specify a format string
   - Most format string vulnerabilities are solved by specifying "%s" as format string and not using the data string as format string
   - If possible, make the format string a constant
     - Extract all the variable parts as other arguments to the call
     - Difficult to do with some internationalization libraries
   - Use compiler switches, code scanners and run-time defenses
   - Be wary of calling C libraries from other languages
Code Scanners

- Pscan searches for format string functions called with the data string as format string
  - Can also look for custom functions
    - Needs a helper file that can be generated automatically
      - Pscan helper file generator at http://www.cerias.purdue.edu/homes/pmeunier/dir_pscan.html
    - Few false positives

- http://www.striker.ottawa.on.ca/~aland/pscan/
Case Study: Cfingerd 1.4.3

- Finger replacement
  - Runs as root
  - Pscan output: (CVE-2001-0609)
    - defines.h:22 SECURITY: printf call should have "%s" as argument 0
    - main.c:245 SECURITY: syslog call should have "%s" as argument 1
    - main.c:258 SECURITY: syslog call should have "%s" as argument 1
    - standard.c:765 SECURITY: printf call should have "%s" as argument 0
    - etc... (10 instances total)
  - Discovery: Megyer Laszlo, a.k.a. "Lez"
Cfingerd Analysis

- Most of these issues are not exploitable, but one is, indirectly at that...
- Algorithm (simplified):
  - Receive an incoming connection
    - get the fingered username
  - Perform an ident check (RFC 1413) to learn and log the identity of the remote user
  - Copy the remote username into a buffer
  - Copy that again into "username@remote_address"
    - remote_address would identify attack source
  - Answer the finger request
  - Log it
Cfingerd Vulnerabilities

- A string format vulnerability giving root access:
  - Remote data (ident_user) is used to construct the format string:
    - `snprintf(syslog_str, sizeof(syslog_str), "%s fingered from %s", username, ident_user);
      syslog(LOG_NOTICE, (char *) syslog_str);

- An off-by-one string manipulation (buffer overflow) vulnerability that
  - prevents remote_address from being logged (useful if attack is unsuccessful, or just to be anonymous)
  - Allows ident_user to be larger (and contain shell code)
Cfingerd Buffer Overflow Vulnerability

- memset(uname, 0, sizeof(uname));
  for (xp=uname;
       *cp!='\0' && *cp!='\r' && *cp!='\n' && strlen(uname) < sizeof(uname);
       cp++)
    * (xp++) = *cp;

- Off-by-one string handling error
  - uname is not NUL-terminated!
  - because strlen doesn't count the NUL

- It will stop copying when strlen goes reading off outside the buffer
Direct Effect of Off-by-one Error

- char buf[BUFLEN], uname[64];
- "uname" and "buf" are "joined" as one string!
- So, even if only 64 characters from the input are copied into "uname", string manipulation functions will work with "uname+buf" as a single entity
- "buf" was used to read the response from the ident server so it is the raw input
Consequences of Off-by-one Error

1) Remote address is not logged due to size restriction:
   - `snprintf(bleah, BUFLEN, "%s@%s", uname, remote_addr);`
   - Can keep trying various technical adjustments (alignments, etc...) until the attack works, anonymously

2) There's enough space for format strings, alignment characters and shell code in buf (~60 bytes for shell code):
   - Rooted (root compromise) when syslog call is made
     - i.e., cracker gains root privileges on the computer (equivalent to LocalSystem account)
gcc Options

- **-Wformat** (man gcc)
  - "Check calls to "printf" and "scanf", etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense."
  - Also checks for null format arguments for several functions
    - -Wformat also implies -Wnonnull

- **-Wformat-nonliteral** (man gcc)
  - "If -Wformat is specified, also warn if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a "va_list"."
gcc Options

- **-Wformat-security (man gcc)**
  - "If -Wformat is specified, also warn about uses of format functions that represent possible security problems. At present, this warns about calls to "printf" and "scanf" functions where the format string is not a string literal and there are no format arguments, as in "printf (foo);". This may be a security hole if the format string came from untrusted input and contains %n. (This is currently a subset of what -Wformat-nonliteral warns about, but in future warnings may be added to -Wformat-security that are not included in -Wformat-nonliteral.)"

- **-Wformat=2**
  - Equivalent to -Wformat -Wformat-nonliteral -Wformat-security.
Making gcc Look for Custom Functions

- Function attributes
  - Keyword "__attribute__" followed by specification
  - For format strings, use "__attribute__ ((format))"
  - Example:
    ```c
    my_printf (void *my_object,
                const char *my_format, ...)
    __attribute__ ((format (printf, 2, 3)));
    ```

- gcc can help you find functions that might benefit from a format attribute:
  - Switch: "-Wmissing-format-attribute"
  - Prints "warning: function might be possible candidate for `printf' format attribute" when appropriate
Integer Overflows

- Know the internal representation of integers
- Be able to determine when an integer overflow can occur
- Understand the consequences of integer overflows
Possible Consequences

- Logic Errors
- Memory Exhaustion
- Buffer Overflows
  - Incl. out-of-bound reads
Integers

- Fixed number of bytes: 1, 2, 4, 8...
- Signed and unsigned
- Types:
  - Char
    - “char” is ambiguous: neither "unsigned char" or "signed char"
    - What happens when converted to an int?
      - Behavior depends on platform, compiler
  - Short
  - Int
  - Long
- Extended types
  - uint_least16_t (integer of at least 16 bits)
Internal Representation, Unsigned

- Unsigned Short:
  - 65535 is FFFF
  - 0 is 0000

If $a = 0$, what is $a-1$?

If $a = 65535$, what is $a+1$?
Unsigned Short Overflows

- $65535 + 1 = 0!$
- $0 - 1$ is $65535$
Signed Short:
- -1 is FFFF
- 32767 is 7FFF
- -32768 is 8000
- If \( a = -32768 \), what is \(-a\)?
- If \( a = 32767 \), what is \( a+1 \)?
Signed Short Overflows

-\(-32768\) is -32768!
- 32767 + 1 = -32768
- -32768 - 1 = 32767
Silent Signed to Unsigned Conversions

- No warning, or compiler warning was ignored
- What happens when you pass a negative number to a function expecting an unsigned integer?
- `void *malloc(size_t size);`
Multiplication Overflow

- If you have an unsigned 16 bit integer and do
  - 0x0100 X 0x0100 = 256 X 256 = ?
- If you have a signed 16 bit integer and do
  - 0x0100 X 0x0080 = 256 X 128 = ?
- What if this was how you calculated how much memory to allocate for a buffer?
Example Possible Overflow

```c
widget* p = (widget*)malloc(numWidgets * sizeof(widget));
if (p == NULL) {
    printf("Error: not enough memory\n");
} else {
    //...
    free(p);
}
```
Malloc(0) Attack Scenario

- Overflow in the size calculations can be engineered to allocate no memory
- `Malloc(0)` is legal, but returned value OS-dependent
  - Sun: returns pointer to the "arena"
  - Pointer to buffer of size 0, or a minimum size
- Program happily trashes the arena, or heap
  - "Fandango on core"
How to Avoid Overflows or Underflows

- **INT_MAX** is your friend
  - ANSI standard, defined in limits.h
- Maximum value for an Integer type
- ```c
   if (INT_MAX/sizeof(widget) > numWidgets) {
       // allocate memory
       // ...
   } else {
       printf("Too many widgets requested\n");
   }
```
Other Useful Things in `limit.h`

- `INT_MIN`
- `LONG_MAX`, `LONG_MIN`
- `SHRT_MAX`, `SHRT_MIN`
- `SCHAR_MAX`, `SCHAR_MIN`
- `UCHAR_MAX`
- `ULONGLONG_MAX`
- `USHORT_MAX`
- `CHAR_MAX`, `CHAR_MIN`

Is type `char` signed or unsigned?
- Depends on the compiler!
Questions

- Which other constant could you have used instead of `INT_MAX` for memory allocation?
- `void * malloc(size_t size);`
  - What is a `size_t`?
  - `typedef unsigned long size_t; // (ANSI C)`
- Why isn't there a `SZT_MAX`?
- What is `(size_t) -1`?
- Tip: Make sure that the variable used to store the result of the calculation is of the correct type (e.g., `size_t`)
Wrong Way to Check

```cpp
int main(){
    int nInteger = 0;
    cout << "Enter value: ";
    cin >> nInteger;
    if(nInteger > INT_MAX){
        cout << "ERROR: Input Exceeds INT_MAX\n\n";
    }
    return 0;
}
```
size_t len = strlen(str);
for (size_t i=len-1; i >= 0; --i) {
    // do things
    ...
}
int ReadString(SOCKET s, char *buf, size_t len)
{
UINT32 size;
if(ReadFully(&size, sizeof(size)) != 0)
    return -1;
if(ReadFully(buf, min(len, size)) != 0)
    return -1;
buf[len] = '\0';
return min(len, size);
}
Questions or Comments?