

Course 2: Programming Issues, Section 2

Pascal Meunier, Ph.D., M.Sc., CISSP

May 2004; updated July 30, 2004

Developed thanks to the support of Symantec Corporation,
NSF SFS Capacity Building Program (Award Number 0113725)
and the Purdue e-Enterprise Center

Copyright (2004) Purdue Research Foundation. All rights reserved.



Course 2 Learning Plan

- Buffer Overflows
- **Format String Vulnerabilities**
- Code Injection and Input Validation
- Cross-site Scripting Vulnerabilities
- Links and Race Conditions
- Temporary Files and Randomness
- Canonicalization and Directory Traversal

Learning objectives

- Learn that format strings are interpreted, therefore are similar to code
- Understand the definition of a format string vulnerability
- Know how they happen
- Know how to format strings safely with regular "C" functions
- Learn other defenses against the exploitation of format string vulnerabilities

Format String Issues: Outline

- Introduction to format strings
- Fundamental "C" problem
- Examples
- Definition
- Importance
- Survey of unsafe functions
- Case study: analysis of cfingerd 1.4.3 vulnerabilities
- Preventing format string vulnerabilities without programming
- Lab: Find and fix format string vulnerabilities
- Tools to find string format issues

What is a Format String?

- In “C”, you can print using a format string:
- `printf(const char *format, ...);`
- `printf(“Mary has %d cats”, cats);`
 - %d specifies a decimal number (from an int)
 - %s would specify a string argument,
 - %X would specify an unsigned uppercase hexadecimal (from an int)
 - %f expects a double and converts it into decimal notation, rounding as specified by a precision argument
 - etc...

Fundamental "C" Problem

- No way to count arguments passed to a "C" function, so missing arguments are not detected
- Format string is interpreted: it mixes code and data
- What happens if the following code is run?
- ```
int main () {
 printf("Mary has %d cats");
}
```

## Result

- % ./a.out  
Mary has -1073742416 cats
- 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...

## User-specified Format String

- What happens if the following code is run, assuming there always is an argument input by a user?
- ```
int main(int argc, char *argv[])  
{  
    printf(argv[1]);  
    exit(0);  
}
```
- Try it and input "%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%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%s
```

Format String Vulnerabilities

- Discovered relatively recently ~2000
- 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 (icat) for “format string”:
 - 115 records in 2002
 - 153 total in 2003
 - 173 total in April 2004
 - 363 in February 2006
- 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

Write Anything Anywhere

- "%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"

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[BUFLLEN], 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, BUFLLEN, "%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)

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
- 2) 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
- 3) If the above two practices are not possible, use defenses such as FormatGuard (see next slides)
 - Rare at design time
 - Perhaps a way to keep using a legacy application and keep costs down
 - Increase trust that a third-party application will be safe

Windows

- Demo code for format string exploit in Howard and Leblanc (2nd Ed.)
 - Same mechanisms as in UNIX-type systems
 - Prevented the same way

Defenses Against Exploitation

- FormatGuard
 - Use compiler macro tricks to count arguments passed
 - ❖ Special header file
 - Patch to glibc
 - ❖ Printf wrapper that counts the arguments needed by format string and verifies against the count of arguments passed
 - Kill process if mismatch
 - ❖ What's the problem with that?

FormatGuard Limitations

- What do you do if there's a mismatch in the argument count?
 - Terminate it (kill)
 - ❖ Not complete fix, but DoS preferable to root compromise
 - If process is an important process that gets killed, Denial-of-Service attacks are still possible
 - ❖ Although if you only manage to kill a "child process" processing your own attack, there's no harm done

FormatGuard Limitations (Cont.)

- Doesn't work when program bypasses FormatGuard by using own printf version or library
 - wu-ftpd had its own printf
 - gftp used Glib library
 - Side note: See how custom versions of standard functions make retrofit solutions more difficult?
 - ❖ Code duplication makes patching more difficult
 - Secure programming is the most secure option

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/>

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:
 - ❖

```
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

Lab

- Jared wrote a server program with examples of format string vulnerabilities
 - Get it from the USB keyring, web site or FTP server (follow the instructor's directions)
- Usage
 - Compile with 'make vuln_server'
 - Run with './vuln_server 5555'
 - Open another shell window and type 'telnet localhost 5555'
 - Find and fix all format string vulnerabilities
 - Try the gcc switches

First Lab Vulnerability

- What happens when you type in the string "Hello world!"? (it's printed back in reverse)
- Type in a long string (more than 100 characters). It should crash. Where is the buffer overflow?
- Fix the buffer overflow, recompile, and demonstrate that it doesn't crash on long input lines any more.
- Bonus: Can you get a shell?
 - We didn't teach how to do that because our primary goal is to teach how to avoid the vulnerabilities, and as this lab demonstrates, you can do that without knowing how to get a shell

Second Lab Vulnerability

- 1) Where is the format string problem?
- 2) How do you crash the program? Hint: use %s
- 3) How do you print the contents of memory to divulge the secret which is 0xdeadcd0de? Hint: use %08x
- 4) Bonus: Can you get a shell?

Third Lab Vulnerability

- Latent vulnerability hidden somewhere...

Questions or Comments?

§

About These Slides

- You are free to copy, distribute, display, and perform the work; and to make derivative works, under the following conditions.
 - You must give the original author and other contributors credit
 - The work will be used for personal or non-commercial educational uses only, and not for commercial activities and purposes
 - For any reuse or distribution, you must make clear to others the terms of use for this work
 - Derivative works must retain and be subject to the same conditions, and contain a note identifying the new contributor(s) and date of modification
 - For other uses please contact the Purdue Office of Technology Commercialization.
- Developed thanks to the support of Symantec Corporation



Pascal Meunier

pmeunier@purdue.edu

Contributors:

Jared Robinson, Alan Krassowski, Craig Ozancin, Tim Brown, Wes Higaki, Melissa Dark, Chris Clifton, Gustavo Rodriguez-Rivera

