



Format Strings

- What is a format string vulnerability
- How it can happen
- How to format strings safely with regular "C" functions
- Other defenses against the exploitation of format string vulnerabilities

1



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
 - ...

2



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");  
}
```

3



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)

4



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...

5



User-specified Format String

- What happens if the following code is run, assuming there 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%s" How many "%s" arguments do you need to crash it?

6



Result

```
% ./a.out "%s%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 was not 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?

7



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%s%s
```

8



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 available on the web
 - "Modify and hack from example"


9



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 appropriate number of arguments
- Difficult to establish whether a data string could possibly be affected by an attacker; 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

10



How Important Are Format String Vulnerabilities?

- Search **National Vulnerability Database** (NIST) for "format string":
 - Over 700 records overall
 - 171 last 3 years
- Search **Database a Mitre** (cve.mitre.org) for "format string":
 - 135 records of vulnerabilities
- Various applications
 - Databases (Oracle)
 - Unix services (syslog, ftp,...)
 - Linux "super" (for managing setuid functions)
 - cfingerd CAN 2001-0609
- Arbitrary code execution is a frequent consequence

11



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?

12



Sprintf Double Whammy

- `int sprintf(char *s, const char *format, /* args*/ ...);`
- **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

13



Better functions than sprintf

Note: do not prevent format string vulnerabilities:

- `snprintf(char *str, size_t size, const char *format, ...);`
 - `sprintf` with length check for "size"
 - Does not guarantee NUL-termination of `s` on some platforms (Microsoft, Sun)
 - MacOS X: NUL-termination guaranteed
 - Check with "man sprintf"
- `asprintf(char **ret, const char *format, ...);`
 - sets `*ret` to be a pointer to a buffer sufficiently large to hold the formatted string.

14



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

15



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"

16



Case Study: Cfingerd 1.4.3

- Finger replacement
 - Runs as root
 - Pscan output: (CAN 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)

17



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

18



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)

19



## 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

20



## 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

21



## Consequences of Off-by-one Error

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

22



## 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
  - 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

23



## 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](http://www.cerias.purdue.edu/homes/pmeunier/dir_pscan.html)
    - Few false positives
- <http://www.striker.ottawa.on.ca/~aland/pscan/>

24