Secure Programming

Format Strings

1

Ahmet Burak Can
Hacettepe University

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 and importance
- Survey of unsafe functions

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

- Sample execuiton

```
$ ./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...

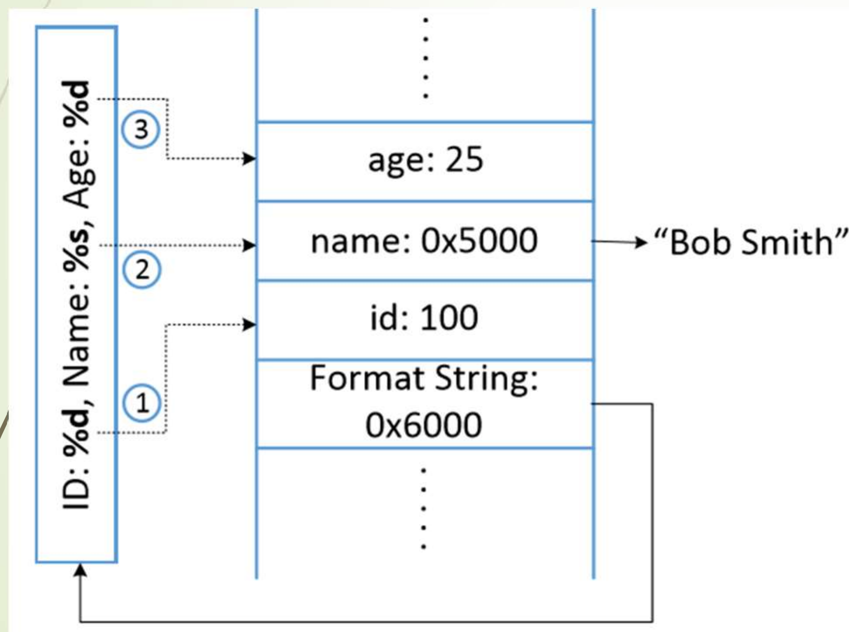
How `printf()` Access Optional Arguments

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

- Here, `printf()` has three optional arguments. Elements starting with "%" are called format specifiers.
- `printf()` scans the format string and prints out each character until "%" is encountered.

How `printf()` Access Optional Arguments



- When printf() is invoked, the arguments are pushed onto the stack in reverse order.
- When it scans and prints the format string, printf() replaces %d with the value from the first optional argument and prints out the value.

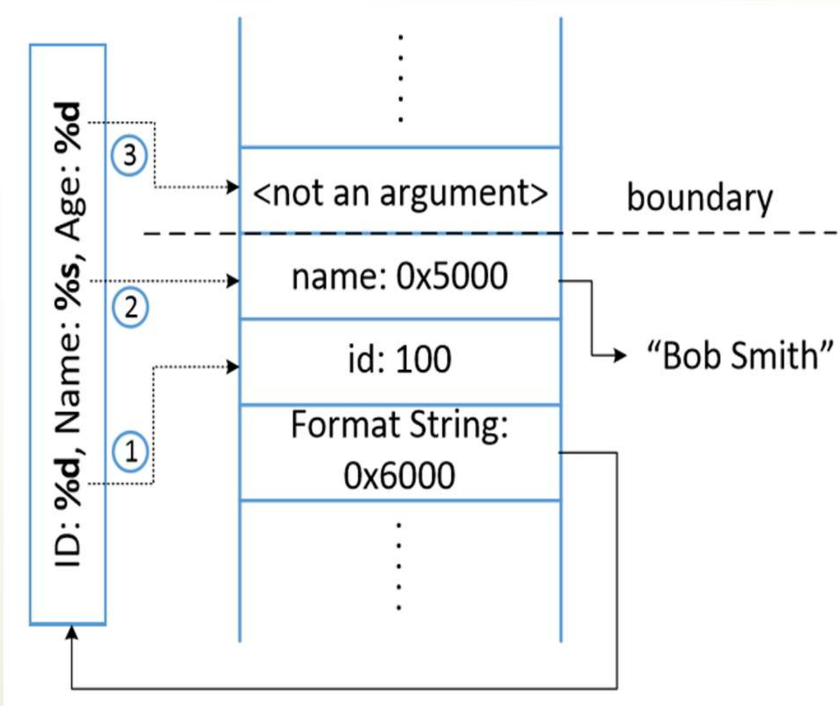
Missing Optional Arguments

```
#include <stdio.h>

int main()
{
    int id=100, age=25; char *name = "Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n", id, name);
}
```

- printf() doesn't understand if it reached the end of the optional argument list.
- It continues fetching data from the stack.



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%s"
- How many "%s" arguments do you need to crash it?

Result

- Sample execution:

```
$ ./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);
}
```

- Sample execution:

```
$ ./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 a 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

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

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

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

- 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://deployingradius.com/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
      ```