Format String Vulnerabilities

Most slides courtesy Wenliang Du @ Syracuse Univ. (with modifications)
Outline

● Format String
● Access optional arguments
● How `printf()` works
● Format string attack
● How to exploit the vulnerability
● Countermeasures
printf()
Format String

printf() - To print out a string according to a format.

```c
int printf(const char *format, ...);
```

The argument list of `printf()` consists of:

- One concrete argument format
- Zero or more optional arguments

Hence, compilers don’t complain if fewer arguments are passed to `printf()` during invocation.
Access Optional Arguments

```c
#include <stdio.h>
#include <stdarg.h>

int myprint(int Narg, ... )
{
    int i;
    va_list ap;
    va_start(ap, Narg);
    for(i=0; i<Narg; i++) {
        printf("%d ", va_arg(ap, int));
        printf("%f\n", va_arg(ap, double));
    }
    va_end(ap);
}

int main()
{
    myprint(1, 2, 3.5);
    myprint(2, 2, 3.5, 3, 4.5);
    return 1;
}
```

- myprint() shows how printf() actually works.
- Consider myprintf() is invoked in line 7.
- va_list pointer (line 1) accesses the optional arguments.
- va_start() macro (line 2) calculates the initial position of va_list based on the second argument Narg (last argument before the optional arguments begin)
Access Optional Arguments

- **va_start()** macro gets the start address of Narg, finds the size based on the data type and sets the value for va_list pointer.

- va_list pointer advances using **va_arg()** macro.

- **va_arg(ap, int)** : Moves the ap pointer (va_list) up by 4 bytes.

- When all the optional arguments are accessed, **va_end()** is called.
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.
- `printf()` calls `va_arg()`, which returns the optional argument pointed by `va_list` and advances it to the next argument.
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.
- `va_list` is then moved to the position 2.
va_arg()

The `va_arg()` macro expands to an expression that has the type and value of the next argument in the call. The argument `ap` is the `va_list ap` initialized by `va_start()`. Each call to `va_arg()` modifies `ap` so that the next call returns the next argument. The argument `type` is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a `*` to `type`.

The first use of the `va_arg()` macro after that of the `va_start()` macro returns the argument after `last`. Successive invocations return the values of the remaining arguments.

If there is no next argument, or if `type` is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), random errors will occur.

If `ap` is passed to a function that uses `va_arg(ap, type)` then the value of `ap` is undefined after the return of that function.
Missing Optional Arguments

```c
#include <stdio.h>

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

- `va_arg()` macro doesn’t understand if it reached the end of the optional argument list.
- It continues fetching data from the stack and advancing `va_list` pointer.
Format String Vulnerability

```c
printf(user_input);
```

```c
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

```c
sprintf(format, "%s %s", getenv("PWD"), ": %d");
printf(format, program_data);
```

In these three examples, user’s input (user_input) becomes part of a format string.

What will happen if `user_input` contains format specifiers?
Vulnerable Code

```c
#include <stdio.h>

void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    /* print out information for experiment purpose */
    printf("Target address: %x\n", (unsigned) &var);
    printf("Data at target address: 0x%lx\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input)-1, stdin);

    // The vulnerable place
    printf(input);

    printf("Data at target address: 0x%lx\n", var);
}

void main() { fmtstr(); }
```
Vulnerable Program’s Stack

Inside `printf()`, the starting point of the optional arguments (va_list pointer) is the position right above the format string argument.
What Can We Achieve?

Attack 1 : Crash program

Attack 2 : Print out data on the stack

Attack 3 : Change the program’s data in the memory

Attack 4 : Change the program’s data to specific value

Attack 5 : Inject Malicious Code
Attack 1: Crash Program

Use input: \%s%\%s\%s\%s\%s\%s\%s\%s

printf() parses the format string.
For each \%s, it fetches a value where va_list points to and advances va_list to the next position.
As we give \%s, printf() treats the value as address and fetches data from that address. If the value is not a valid address, the program crashes.
Suppose a variable on the stack contains a secret (constant) and we need to print it out.

Use user input: \%x\%x\%x\%x\%x\%x\%x\%x

printf() prints out the integer value pointed by va_list pointer and advances it by 4 bytes.

Number of \%x is decided by the distance between the starting point of the va_list pointer and the variable. It can be achieved by trial and error.
Attack 3: Change Program’s Data in the Memory

Goal: change the value of `var` variable from 0x11223344 to some other value.

- `%n`: Writes the number of characters printed out so far into memory.
- `printf("hello%n", &i)` ⇒ When `printf()` gets to `%n`, it has already printed 5 characters, so it stores 5 to the provided memory address.
- `%n` treats the value pointed by the `va_list` pointer as a memory address and writes into that location.
- Hence, if we want to write a value to a memory location, we need to have it’s address on the stack.
Attack 3 : Change Program’s Data in the Memory

Assuming the address of var is 0xbfffffff304 (can be obtained using gdb)

$ echo $(printf "\04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input

- The address of var is given in the beginning of the input so that it is stored on the stack.
- $(command): Command substitution. Allows the output of the command to replace the command itself.
- "\x04" : Indicates that “04” is an actual number and not as two ascii characters.
Attack 3 : Change Program’s Data in the Memory

- var’s address (0xbffffff304) is on the stack.
- **Goal**: To move the va_list pointer to this location and then use %n to store some value.
- %x is used to advance the va_list pointer.
- How many %x are required?
Attack 3 : Change Program’s Data in the Memory

Using trial and error, we check how many \%x are needed to print out 0xbffff304.

Here we need 6 \%x format specifiers, indicating 5 \%x and 1 \%n.

After the attack, data in the target address is modified to 0x2c (44 in decimal).

Because 44 characters have been printed out before \%n.
Attack 4 : Change Program’s Data to a Specific Value

Goal: To change the value of `var` from \texttt{0x11223344} to \texttt{0x9896a9}

```bash
$ echo $(printf "\x04\xf3\xff\xbf")_%.8x_%.8x_%.8x_%.8x_.10000000x%n > input
$ uv1 < input
Target address: bfffff304
Data at target address: 0x11223344
Please enter a string:
   ****_00000063_b7fc5ac0_b7eb8309_bfffff33f_000000
```

`printf()` has already printed out 41 characters before `.10000000x`, so, 
\(10000000+41 = 10000041\) (0x9896a9) \textit{will be stored in 0xbfffff304}. \[sic\]
# include <stdio.h>
void main()
{
    int a, b, c;
    a = b = c = 0x11223344;

    printf("12345\n", &a);
    printf("The value of a: 0x%x\n", a);
    printf("12345\n", &b);
    printf("The value of b: 0x%x\n", b);
    printf("12345\n", &c);
    printf("The value of c: 0x%x\n", c);
}
Attack 4 : A Faster Approach

Goal: change the value of \texttt{var} to 0x66887799

- Use \texttt{%hn} to modify the \texttt{var} variable two bytes at a time.

- Break the memory of \texttt{var} into two parts, each with two bytes.

- Most computers use the Little-Endian architecture
  - The 2 least significant bytes (0x7799) are stored at address 0xbffff304
  - The 2 significant bytes (0x6688) are stored at 0xbffff306

- If the first \texttt{%hn} gets value $x$, and before the next \texttt{%hn}, $t$ more characters are printed, the second \texttt{%hn} will get value $x+t$. 
Attack 4: A Faster Approach

- Overwrite the bytes at 0xbfffff306 with 0x6688.
- Print some more characters so that when we reach 0xbfffff304, the number of characters will be increased to 0x7799.

```
$ echo $(printf "\x06\x0f\x0f\x0f\x0f\x04\x06\x0f\x0f\x0f"
         _%8x_%.8x_%8x_%.8x_%8x_.26199x%hn_%4368x%hn > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****@@@*****00000063_b7fc5ac0_b7eb8309_bffff33f_00000
0000 (many 0’s omitted) 000040404040
Data at target address: 0x66887799
```
Attack 4 : Faster Approach

- Write 0x7799 to Address B
- Move va_list for 4 bytes

- Write 0x6688 to Address A
- Move va_list to 2

- Print 0x6688 characters
- Move the va_list pointer for 20 bytes from its starting point to 1

- Print 0x1111 more characters
- Move the va_list pointer to 3

- Address A : first part of address of var ( 4 chars )
- Address B : second part of address of var ( 4 chars)
- 4 %.8x : To move va_list to reach Address 1 (Trial and error, 4x8=32)
- @@@@@@ : 4 chars
- 5 _ : 5 chars
- Total : 12+5+32 = 49 chars
Attack 4: Faster Approach

- To print 0x6688 (26248), we need 26248 - 49 = 26199 characters as precision field of %x.
- If we use %hn after first address, va_list will point to the second address and same value will be stored.
- Hence, we put @@@@ between two addresses so that we can insert one more %x and increase the number of printed characters to 0x7799.
- After first %hn, va_list pointer points to @@@@, the pointer will advance to the second address. Precision field is set to 4368 =30617 - 26248 -1 in order to print 0x7799 (30617) when we reach second %hn.
Attack 5: Inject Malicious Code

**Goal**: To modify the return address of the vulnerable code and let it point it to the malicious code (e.g., shellcode to execute /bin/sh). Get root access if vulnerable code is a SET-UID program.

**Challenges**:

- Inject Malicious code in the stack
- Find starting address (A) of the injected code
- Find return address (B) of the vulnerable code
- Write value A to B
Attack 5 : Inject Malicious Code

- Using gdb to get the return address and start address of the malicious code.
- Assume that the return address is 0xbfffff38c
- Assume that the start address of the malicious code is 0xbfffe358

**Goal:** Write the value 0xbfffe358 to address 0xbffff38c

**Steps:**

- Break 0xbfffff38c into two contiguous 2-byte memory locations: 0xbfffe38c and 0xbfffe38e.
- Store 0xbfffe into 0xbfffe38e and 0xf358 into 0xbfffe38c
Attack 5 : Inject Malicious Code

- Number of characters printed before first `\%hn` = 12 + (4x8) + 5 + 49102 = 49151 (`0xbfff`).
- After first `\%hn`, 13144 + 1 = 13145 are printed.
- 49151 + 13145 = 62296 (`0xbffff358`) is printed on `0xbffff38c`
Countermeasures: Developer

- Avoid using untrusted user inputs for format strings in functions like `printf`, `sprintf`, `fprintf`, `vprintf`, `scanf`, `vfscanf`.

// Vulnerable version (user inputs become part of the format string):
    sprintf(format, "%s %s", user_input, "\": %d");
    printf(format, program_data);

// Safe version (user inputs are not part of the format string):
    strcpy(format, "%s: %d");
    printf(format, user_input, program_data);
Countermeasures: Compiler

Compilers can detect potential format string vulnerabilities

- Use two compilers to compile the program: `gcc` and `clang`.
- We can see that there is a mismatch in the format string.

```c
#include <stdio.h>

int main()
{
    char *format = "Hello %x%x%x\n";
    printf("Hello %x%x%x\n", 5, 4);  // ①
    printf(format, 5, 4);          // ②
    return 0;
}
```
With default settings, both compilers gave warning for the first `printf()`.

No warning was given out for the second one.
Countermeasures: Compiler

$ gcc -Wformat=2 test_compiler.c
  test_compiler.c:7:4: ... (omitted, same as before)
  test_compiler.c:8:4: warning: format not a string literal, argument
types not checked
[-Wformat-nonliteral]

$ clang -Wformat=2 test_compiler.c
  test_compiler.c:7:23: ... (omitted, same as before)
  test_compiler.c:8:11: warning: format string is not a string literal
[-Wformat-nonliteral]
  printf(format, 5, 4);
      ~~~~~
  2 warnings generated.

- On giving an option -wformat=2, both compilers give warnings for both printf statements stating that the format string is not a string literal.

- These warnings just act as reminders to the developers that there is a potential problem but nevertheless compile the programs.
Countermeasures

- **Address randomization**: Makes it difficult for the attackers to guess the address of the target memory (return address, address of the malicious code).

- **Non-executable Stack/Heap**: This will not work. Attackers can use the return-to-libc technique to defeat the countermeasure.

- **StackGuard**: This will not work. Unlike buffer overflow, using format string vulnerabilities, we can ensure that only the target memory is modified; no other memory is affected.
Summary

- How format string works
- Format string vulnerability
- Exploiting the vulnerability
- Injecting malicious code by exploiting the vulnerability