String Format Attack

Chetan Mistry

November 7, 2019

Contents

1 Abstract

A string format attack works by exploiting vulnerabilities in source code to examine and/or modify the values stored in variables.

2 String Formatting in C

Some String manipulation functions in C have vulnerabilities which can be exploited by an attacker. An example of this is the function `scanf()` which allows the user to provide an arbitrary input.

2.1 `scanf()`

`scanf()` is a function which reads from stdin and stores it into a variable. This input can be anything (strings, numbers, etc), and can have any length.

2.2 `printf()`

`printf()` is a function which takes in a string and any parameters to that string (e.g variables) and will output to stdout. The variables can be printed by using special tokens in the string which will map to specific output formats, for example `%d` outputs a number in denary format, whereas `%x` outputs in hex. The way this works is that the variables to be printed are placed into the function stack, and then whenever a token is encountered, it simply replaces it with whatever is at the current stack pointer, and then moving the stack pointer so that it is now at the next variable.

2.3 `%n`

`%n` is a special formatting character which, instead reading the value stored at the stack pointer, will instead overwrite it with the number of bytes read in so far.

3 Attacks

3.1 Denial of Service (DoS)

`%s` is interpreted by `printf()` as a command which will use the current value in function stack as a pointer to a null-terminated string. This means that the program will unconditionally dereference a value, this leads to 2 different situations:

1. The address is valid and accessible to the program, in which

case it will continually print the characters found at that address until the null-character (`\0`) is met.

1. The address is invalid/out-of-bounds to the current program,

resulting in a segmentation-fault (seg-fault), which causes the OS to terminate execution.

Denial of Service works by entering enough `%s` characters until the value in the stack is interpreted as an invalid pointer.

An example input to cause the second situation is `%s%s%s%s%s%s%s%s%s%s` This continually moves the stack pointer, dereferencing the values and printing what is stored in those addresses.

Another DoS attack can be done by what is known as "Stack Smashing" which is when a large number of formatting characters are input (eg. `%d`, `%x`, `%c`, etc.), if enough of these formatting characters are entered, the stack pointer will move out of the function call stack.

3.2 Reading Values on the Stack

When `printf()` is called, it loads the values to be printed into its call stack. These values are then read off every time the string formatting characters are met. It is possible to read values which aren't in the `printf()` call stack by moving the stack pointer outside of the scope of the function by continually entering the format characters. This then allows access to variables stored by the program.

3.2.1 Variables vs Pointers

Some variables are statically allocated in the program stack at compile time (eg `char[6]`), these values can be directly output by using specific formatting characters (`%x`).

Other variables are dynamically allocated (`char*`) in the program heap, and can only be accessed by dereferencing a pointer to them. To print them, specific formatting characters (`%s`) must be used as they interpret the value on the stack as an address rather than a value.

3.2.2 Data Structures

Data Structures are typically dynamically allocated, as a result the pointer to them will only point to the first value stored. In order to print off the whole structure, we can utilise the fact that data is stored in contiguous blocks of memory. This means that if an attacker knows the address of the first element, it is possible to calculate the addresses of further variables by noting the type of the data structure (eg int, char, bool, etc) and using the size of the type as the size to step through in memory.

3.3 Writing Values to the Stack

This attack utilises the `%n` operator described earlier. If an attacker wants to change the value stored in a particular location in memory, then the address must first be found in the program stack, then `%n` can be used to change the value to be the number of bytes read by `printf()` so far. This attack requires the address of the variable to be changed.

4 Lab 1: String Format Attack

4.1 Task 1.i: Crash the Program

The program can be crashed by entering the string `%s%s%s%s%s%s%s%s%s%s`

4.2 Task 1.ii: Print out the Value of `secret[1]`

The address of `secret[1]` is printed out by the source program. As a result we know the exact address to access. The program allows for 2 values to be input, one is a number, the other is a string. By putting the decimal encoded address of `secret[1]` as the number to be input, it gets placed in the stack in some location. This value can then be found and accessed by entering special `printf()` formatting characters, e.g. `%x` which prints out values as a hex-encoded number. Once the address that was placed earlier has been found in the resulting output string, the attacker can then replace the corresponding `%x` character with `%s` which then prints out the value stored in that address (which in this case is `secret[1]`).

4.3 Task 1.iii: Modify the value of `secret[1]`

By accessing the value of `secret[1]` as explained in **Task 1.ii**, instead of simply printing the value of `secret[1]` with `%s`, it is possible to change the value stored by instead using `%n` which replaced the value stored by the number of bytes printed by `printf()` so far.

4.4 Task 1.iv: Modify the value of `secret[1]` to a chosen value

From the previous task, it is possible for the attacker to modify the value of `secret[1]`. From this point, it is possible to modify `secret[1]` by using the same method of attack from **Task 1.iii**, but before utilising `%n` to change the value, an arbitrary number of characters (e.g. *1*) can be entered which will increment the value updated by `%n` by 1 for each character printed.

4.5 Task 2.i: Cause the program to Crash

This can be done in exactly the same way as **Task 1.i**, by entering the string `%s%s%s%s%s%s%s%s%s%s`.

4.6 Task 2.ii: Print out the Value of `secret[1]`

The modified program no longer allows the attacker to enter the address of `secret[1]` directly. As a result, the attacker needs to enter the address and access that address in a single string.

Due to the fact that Virtual Address Randomisation has been turned off, it means that during every instance of the attack, the address of `secret[1]` remains the same. Using this knowledge, it is possible to input that address as a ascii-encoded bytes which can be passed into the program from a file. This address can then be accessed in a similar way to the previous attack by repeatedly entering `%x` in the ascii-encoded file until that address is located. Finally, the attacker can replace the `%x` corresponding to the address of `secret[1]` with a `%s` which will print out the value stored there.

4.7 Task 2.iii: Modify the Value of `secret[1]`

By utilising the steps taken in **Task 2.ii**, it is possible to modify the value stored in `secret[1]` by replacing the `%s` which prints the value stored in `secret[1]` with `%n` which will modify the value.

4.8 Task 2.iv: Modify the Value of `secret[1]` to a chosen value

Similarly to tasks: **2.iii** and **1.iv**, it is possible to modify the value in `secret[1]` to a chosen value by first being able to modify the value in general, and then utilising the fact that `%n` increases for every byte printed so far. From this, it means that it is possible for an attacker to generate an ascii-encoded file with the number of bytes desired to result in modifying `secret[1]` to the desired value.

5 Possibilities and Limitations

Using this attack, attackers can do the following:

- Overwrite important program flags that control access privileges: if

the program you're running has Set-UID privilages, you can leverage the permissions of the privilaged program to write higher privilages to a malicious program.

- Overwrite return addresses on the stack, function pointers, etc Writing any value, as demonstrated above, is possible through the use of dummy output characters. To write a value of 1000, a simple padding of 1000 dummy characters would do for example.

6 History

String formatting to change behaviour has been known since 1989, but as an attack vector, celebrates it's 20th year anniversary this year. The attack itself was uncovered while running a security audit on the "ProFTPD" program: essentially a feature rich FTP server, during which it was shown that by passing certain values into an unguarded `printf()` function, privilage escalation could be achieved. Since this was uncovered, a number of prevention mechanisms have been devised to assist programmer's in detecting the vulnerability.

7 Prevention

- Compiler Support

- Address randomization: just like the countermeasures used to protect against buffer-overflow attacks, address randomization makes it difficult for the attackers to find out what address they want to read/write

8 Detection

- x86 compiled binary analysis