Format String Attacks

A string error which can lead to security vulnerabilities is *format string attacks*. Recall that *printf* takes a format string (a string with *%* signs to indicate format conversions, such as *%d* to convert a decimal integer), and then an appropriate number of other arguments for the values to convert. A format string vulnerability arises whenever there is a possibility that the user may affect the format string in such a way that they can introduce extra format conversions.

As a simple example, imagine that there were a *readAString* () function which reads a string from the user. Consider the following vulnerable code, which attempts to read a string then print it back:

```
1

2

3

// BAD CODE: DO NOT DO!

char * input = readAString();

printf(input);
```

If an attacker inputs a string with *%* signs in it, it can cause the program to behave in ways that it should not. Notice that the call to *printf* in the above code uses the input read from the user as the format string. Even though there are no arguments for format specifiers to convert, *printf* is not deterred. If the user input contains *%*-conversion, *printf* will take the data where these arguments *should be*, format them as directed, and print them. At a minimum, the attacker can cause the program to reveal data by placing *%d* or *%s* format specifiers in his input.

What information this attack reveals depends on what information resides in the place that *printf* looks for those extra arguments. To make this vulnerability even worse, *printf* has a format specifier (*%n*) which writes the number of characters printed so far to a memory location specified by an **int** * passed as the appropriate argument. A clever attacker can use this format conversion to modify the memory of the program in malicious ways, changing its behavior, and possibly executing arbitrary code.

The correct way to use *printf* with a string input (or potentially affected) by the user is to use the *%s* conversion:

```
1
2
3
// CORRECT CODE
char * input = readAString();
printf("%s", input);
```

Note that *gcc* will give you a warning if your format string is not a literal, and you have no format arguments (*i.e.*, the format string is the only argument to *printf*), however, it will not warn you if there are other arguments. This behavior may seem odd, but exists for a good reason. If you have nothing to convert, you should use "*%s*" for the string. However, there may be times when you have an argument and want to compute the format string. For example, the following code is fine:

```
1
2
3
4
5
const char * fmt = "%d\n";
if (printInHex) {
  fmt = "%x\n";
}
printf(fmt, someNumber);
```

Here, we are sure that the format string is either *%d\n* (print a number as decimal) or *%x\n* (print a number as hex), either of which is fine to print someNumber (which we assume is an **int**). However, we must be cautious whenever we write code which computes a format string to ensure that the user may not affect it in malicious ways.

Format string vulnerabilities fall into a larger category of security flaws where a program uses *unsanitized inputs*. More generally, if a program uses strings in a way that certain characters are special, it must take care to remove or escape those characters in input read from the user. In the case of format strings for *printf*, these special characters are *%* signs. If we wanted to let the user control the format string, we could do so safely if (and only if), we took care to *sanitize* the string first—iterating over it and modifying *%* signs to remove their special meaning (*i.e.*, by removing them or converting them to *%%*—the format specifier which prints a literal percent sign). However, in the case of printf there is no reason to take this approach—it is simpler (and thus less error prone) to simply use the *%s* specifier to print the string literally. If we need format specifiers in a user-dependent way, our code should build the format string itself.

There are, however, other situations where we may wish to read a string from the user and include it in some context where characters have special meanings. Two of the most common cases are commands that are passed to the command shell, and information passed to databases.

The command shell considers many characters to be special, but one particularly dangerous one is `—text enclosed in back-ticks is executed as its own command. Suppose our program reads some input from the user, and passes it as an argument to a shell command—that is, it executes *someCommand stringFromUser*. If a malicious user enters *`rm -rf *`*, then the command shell will perform back-tick expansion, and run the command *rm -rf **, which will erase all files in the current directory. While this command is destructive, a more insidious user could find far better commands to execute—ones which give them access to the system to gain and/or modify information.

A similar problem can occur with improper use of databases, where the program passes SQL commands to the database as a string. We will not go into the details of SQL, but imagine we can illustrate the point without a full understanding of them. Suppose the program wants to run the command *SELECT * from Users WHERE name='strFromUser'*, where strFromUser is a string read from the user (*e.g.*, you have asked them for their user name, and they have entered it). If we are not careful, the user may type a **'** (terminating the literal which name is matched against), and a **;** to end the current command, followed by an arbitrary command of his choosing. Such a vulnerability allows the attacker to modify information in the database however he wants. The web-comic xkcd has a nice cartoon on this topic: http://xkcd.com/327/.

Note that sanitizing inputs is a task which must be performed with care. A sanitization function which catches half of the cases is barely better than not sanitizing at all—a clever attacker will try all the possible special characters, and eventually find the one allowing her to compromise your system. Whenever you need to sanitize your inputs, the best thing to do first is to check if there is already a function available to you, written by experts. For example, some database interfaces have prepared statements, which allow you to write the SQL query with *?s* in the place of various inputs, bind values to those inputs, and then the database library ensures that there are no input sanitization issues.

Completed