Format String Attacks

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application. In this way, the attacker could execute code, read the stack, or cause a segmentation fault in the running application, causing new behaviors that could compromise the security or the stability of the system.

To understand the attack, it's necessary to understand the components that constitute it.

•The **Format Function** is an ANSI C conversion function, like **printf, fprintf**, which converts a primitive variable of the programming language into a human-readable string representation.

•The **Format String** is the argument of the Format Function and is an ASCII Z string which contains text and format parameters, like: **printf ("The magic number is: %d\n", 1911)**;

•The **Format String Parameter**, like **%x %s** defines the type of conversion of the format function.

The attack could be executed when the application doesn't properly validate the submitted input. In this case, if a Format String parameter, like %x, is inserted into the posted data, the string is parsed by the Format Function, and the conversion specified in the parameters is executed. However, the Format Function is expecting more arguments as input, and if these arguments are not supplied, the function could read or write the stack.

In this way, it is possible to define a well-crafted input that could change the behavior of the format function, permitting the attacker to cause denial of service or to execute arbitrary commands.

If the application uses Format Functions in the source-code, which is able to interpret formatting characters, the attacker could explore the vulnerability by inserting formatting characters in a form of the website. For example, if the **printf** function is used to print the username inserted in some fields of the page, the website could be vulnerable to this kind of attack, as showed below:

```
printf (userName);
```

Following are some examples of Format Functions, which if not treated, can expose the application to the Format String Attack.

Table 1. Format Functions

Format function	Description
fprint	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length

Format function	Description
vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

Below are some format parameters which can be used and their consequences:

•"%x" Read data from the stack

•"%s" Read character strings from the process' memory

•"%n" Write an integer to locations in the process' memory

Example

```
#include  <stdio.h>
void main(int argc, char **argv)
{
// This line is safe
printf("%s", argv[1]);

// This line is vulnerable
printf(argv[1]);}
```

Safe Code

The line `printf("%s", argv[1]);` in the example is safe, if you compile the program and run it:

```
./example "Hello World %s%s%s%s%s%s"
```

The `printf` in the first line will not interpret the "%s%s%s%s%s%s" in the input string, and the output will be: "Hello World %s%s%s%s%s%s"

Vulnerable Code

The line `printf(argv[1]);` in the example is vulnerable, if you compile the program and run it:

```
./example "Hello World %s%s%s%s%s%s"
```

The `printf` in the second line will interpret the `%s%s%s%s%s%s` in the input string as a reference to string pointers, so it will try to interpret every %s as a pointer to a string, starting from the location of the buffer (probably on the Stack). At some point, it will get to an invalid address, and attempting to access it will cause the program to crash.

Different Payloads

An attacker can also use this to get information, not just crash the software. For example, running:

```
./example "Hello World %p %p %p %p %p %p"
```

Will print the lines:

```
Hello World %p %p %p %p %p %p
Hello World 000E133E 000E133E 0057F000 CCCCCCCC CCCCCCCC CCCCCCCC
```

The first line is printed from the non-vulnerable version of `printf`, and the second line from the vulnerable line. The values printed after the “Hello World” text, are the values on the stack of my computer at the moment of running this example.

Also reading and writing to any memory location is possible in some conditions, and even code execution.