1. Format String Exploits

a. Format String bugs

The printf family consists of functions with variable arguments –

i. printf (char* format, …)

ii. sprintf (char* dest, char* format, …)

Etc.

**printf internals / varargs functions**

The following is how printf behaves with its arguments –

Like in the case of other functions, the function calling printf pushes the arguments into the stack in the Right to Left order. That is, it first pushes in the variable arguments and then the format string fmt.

printf maintains a pointer ARGP which points just above the format string fmt in the stack. Each time a format code (%d, %s, %c, etc) is encountered in the printf’s format string, the ARGP pointer takes the bytes at the location pointed by it, interprets it according to the format code and prints it out. It then increments the ARGP pointer.

In addition, it also maintains a count variable somewhere on the stack which maintains a count of how many characters have been printed. This count can be retrieved with the %n format code. E.g –

```c
int x;
printf(“Hello %n, OK.”, &x)
```

This code will copy the number of characters printed so far into the variable x.
By passing extra format codes into the format string fmt, the attacker can trick the function into printing the contents of the memory locations on the stack, beyond the arguments that were passed. For example, if the format string is “%d %d %d” in the above case, that is – the function call is as follows – 
printf(“%d %d %d”, a,b);
Then, printf will interpret the locations beyond b as integers to be printed and print it out.

Compilers typically warn when the number of format codes don’t match with the number of variable arguments. But when the format string is accepted from the user or is generated at run-time, it is not possible for the compiler to detect it. For e.g. in the following code snippet which was present in the wu-ftp server, user is a string provided by the user.

```c
void logger(char* user)
{
    char buffer[512];
    snprintf(buffer, sizeof(buffer), user);
    .
    .
    .
    write(log_file);
}
```
The compiler can’t check the string “user” as it is provided by the user. The attacker can enter “%x %x %x %x …” to print the contents of the stack to the log_file. And later, he can manage to read the log files.

This technique can also be used to overcome the ASLR present on systems. For example, suppose there is a friendly function run_cmd in wu-ftpd. But because the system has ASLR enabled, the attacker doesn’t know where the function is located. But the attacker knows that the function logger returns to handle_user_login function and that handle_user_login is at 1500 bytes (say) from run_cmd. As ASLR doesn’t move the functions internally within an executable, the attacker can continue reading off the stack using the above described technique till he figures out the return address of the handle_user_login function. Once, he figures that out, he can easily compute the location of the run_cmd function as he knows that it is 1500 bytes from handle_user_login function.

Also, this technique doesn’t crash the application, since the attacker is only reading values off the stack. So, the ASLR won’t run again and randomize the memory location.
2. Using %n to overwrite arbitrary memory.

We aim to design a format string that does --
Memory[a] ← V

We’ll be able to string these format strings together to achieve complex attacks.

Suppose the ARGP pointer points to the starting location of the buffer array. Then, the attacker can send in a user string in the following manner --

`user = <address to be written> ____________________ %n`

The number of underscores is the value that the attacker wants to be written to the specified address.

`printf` will write the address and the underscores to the buffer and will increment its internal `count` variable each time a character is written to the buffer. When the `%n` is encountered, it will write the value of the internal `count` variable to the memory location pointed by the value on the stack that `ARGP` points to. In this case, since `ARGP` points to the address the attacker wants to write to and the `count` contains the value the attacker wants to write, this achieves the target of writing the value to the desired memory location.

To write to a series of address locations, the attacker can string together the addresses,
followed by correctly positioned %n format codes separated by correct number of underscores.

user = <a1> <a2> <a3> <a4> ____ (enough number of underscores to increment the count to the required value) ____ %n ______________ %n _________ %n ______________%n

for e.g. if the attacker wants to write 0x12, 0x34, 0x56, 0x78 to memory locations a1, a2, a3, a4, he’ll form the string user as follows –

user = <a1><a2><a3><a4> __ (enough underscores so that count ends in 78)__ %n __ (enough underscores so that count ends up 56) __ %n __ (enough underscores so that count ends in 34) __ %n __ (enough underscores so that count ends in 12) __ %n

**How to prevent**

Varargs can be modified to accept the number of vararg parameters. This way, the extra format codes can be detected and the ARGP will not proceed beyond the number of vararg parameters passed actually. This way, the attack can be prevented.