CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 26th, 2011

Format string attacks

Format string bugs

Format string bugs allow **arbitrary memory writes**. A format string bug will allow you to set $memory[i] = value$.

```
int printf(const char * fmt, ...);
int snprintf(char * buff, int size, const char * fmt, ...);
```

If you had a string you wanted to print and you were lazy, you might just do: `printf(str);` As long as `str` does not contain `%` characters, you're good. However, the right (safe) way to print any string would be: `printf("%s", str);`

Important details

Consider the following line of code: `printf(username);`

- What happens if a user has a username which contains `%` characters? If it has a `%d`, the function will attempt to take an argument off the stack. Sometimes that argument could be missing and `printf` will end up using something else off the stack.
- Remember that `printf` can tell you the number of bytes written up to a certain point.
- For instance, `printf("%s%n%d", str, &cnt, x);` will store the number of bytes outputted up until the `%n` in the format string into the variable `cnt`. In this case, the number of bytes outputted until that point would be just `strlen(str)`.
- Even if the buffer given to `snprintf` is too small, `snprintf` will still report the bytes that would have been written up to a certain point in the `%n` variables.

Conclusion: If an attacker has control over the format-string argument of `printf` then maybe he can get `printf` to do something interesting for him.

How does `printf` know where to get its next format-string argument from? `printf`-like functions have an **argument pointer** (ARGP) which points to the first format-string argument on the stack. After such an argument is used, this pointer is incremented by 4-bytes to go to the next argument.

The format string arguments will be pushed first on the stack, since arguments are pushed in reverse order on the stack. So the stack will look something like this after a `snprintf(char * buff, int size, const char * fmt, ...);` call:

Caller activation record	
0xABCD	variables
`snprintf` with format string arguments	
0xABC9	fmt-str-arg 3
0xABC5	fmt-str-arg 2
0xABC1	fmt-str-arg 1
fmt	
size	
buff	
retaddr	
ARGP (**points to 0xABC1**)	
other snprintf local variables	

Caller activation record	
0xABCD	variables
`snprintf` without format string arguments	
fmt	
size	
buff	
retaddr	
ARGP (**points to 0xABCD**)	
other snprintf local variables	

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 26th, 2011
Note: Each `printf` argument will be a pointer (4 bytes) or an integer/floating point value (4 bytes).

Vulnerable code

Consider the following vulnerable function:

```
void log_user(char * user)
{
    char buff[512];
    int x = 1;

    snprintf(buff, sizeof(buff), user);
}
```

This is what the stack will look like when `snprintf` executes:

The stack	
`log_user` activation record	
0x1111 F127	user ptr (4 bytes)
0x1111 F123	retaddr
0x1111 EC11	buff(512 bytes)
0x1111 EC0D	x (4 bytes)
`snprintf` activation record	
0x1111 EC09	fmt = user ptr
size = 512	
buffer = &buff = 0xEC11	
retaddr = &log_user	
ARGP (**points to 0xEC09**)	
other snprintf local variables	

In our attack, we will show how to **modify the value of `x` at address `0xEC0D`** to equal the value 100. Similar attacks can be constructed to modify the return address of **`log_user`** of or virtually any other location in memory.

Note: Attention has to be paid whether the system is **little endian** or **big endian.** We are assuming big endian here (the most significant byte stored in the lower address) just so it is easier to understand the memory address placed in the format string.

The attack

What if the username provided to `log_user` was something like this:

`user = "\x11\x11\xEC\x0D%96d%n"`

Step 1

Since there are no format-string arguments in the call to `snprintf`, **ARGP will point where it would normally expect those arguments to be, just above the `fmt` string, at location 0x1111 EC0D**, which happens to be the address of `x`.

Step 2

When `snprintf` executes, it will store *0x1111 EC0D* in the first 4 bytes of `buff` (note this is the address of `x`).

- `snprintf`'s **outputted bytes count** will be incremented and will equal 4 bytes.

Step 3

Then, the `%96d` specifier will print (with 96 space-padding) the first format string argument, which according to ARGP will be whatever is at address 0x1111 EC0D. Since `x` is at that address, the value of `x` will be copied in the buffer `buff` (with the 96 spaces, minus the length of `x`).

- `snprintf`'s **outputted bytes count** will be incremented by another 96 bytes and will equal 100 bytes.
- Since ARGP was used to read one format-string argument (`x` in our case), ARGP will now be incremented by 4 bytes to point to the next format string argument. Guess what that might be?

Note: As you will later see, the only reason we had to print `x` inside `buff`, with 96 spaces padding was to get `snprintf`'s byte count to equal 100. We really did not care about reading `x` or copying `x` inside `buff`.

Step 4

Now the next thing `snprintf` has to do is handle the `%n` specifier in our format string. As we said before, ARGP has been incremented and now points to 0x1111 EC0D + 4 = 0x1111 EC11. This is the address of the first 4 bytes of `buff`.

Remember that in step 2 we stored 0x1111 EC0D (the address of `x`) in those 4 bytes. We did that for a reason. Now `snprintf` will handle the `%n` specifier and store the **outputted bytes count** at the address specified by the next format-string argument, which according to ARGP (which points to the first 4 bytes of `buff` at 0x1111 EC11) is 0x1111 EC0D.

The net result is that the value 100 (the number of outputted bytes) will be stored at 0x1111 EC0D, which is the address of `x`. We just changed `x` arbitrarily.

More examples

More information about format string attacks can be found in the papers below:

- Format string attacks, by Tim Newsham
- Exploiting format string vulnerabilities, by scut / team teso
- Analysis of format string bugs, by Andreas Thuemmel