Systems Security lab 1: Format strings

Background – printf and format strings

The printf command produces formatted output. In C, it takes the form

```
int printf(const char *format, ...)
```

where the first parameter is a format string and the following parameters are its arguments. For example,

```
printf("Hello, %s! Your user id is %d.\n", name, uid);
```

In a format string, format specifiers start with a % sign. For each specifier in the format string, there should be (usually) one parameter of a matching type in the argument list:

%d	signed int
%u	unsigned int
%x	unsigned int – prints in hex
%s	null-terminated string
%n	expects an int* argument and writes the number of bytes output so far to this parameter.
%%	prints '%' - no coresponding value in the argument list

Many of these options can take a further numeric argument between the % sign and the type specifier. For example, %08x prints an integer in hexadecimal using exactly 8 bytes, padding with leading zeroes if necessary (so 255 prints as 000000ff).

Integer specifiers can also take a length modifier. For example, %hd means a short int and %hhd is a char whereas %ld is a long int and %lld a long long int.

%n

%n is unusual in that it writes rather than reads a value – like all integer specifiers it too can take length modifiers. If 8 bytes have been written so far, %n writes the 4-byte value 0x00000008 to its argument whereas %hhn writes the single byte (char) 0x08.

Why %n exists at all is debatable but it does allow some tricks such as this one courtesy of http://stackoverflow.com/a/3402415:

```
int n;
printf("%s: %nFoo\n", "hello", &n);
printf("%*sBar\n", n, "");
```

which prints the following, the point being that “Foo” and “Bar” are aligned however long the leading string (in this case “hello: ”) is:

```
hello: Foo
       Bar
```

The * length modifier, as in `%*s`, specifies that this argument takes two parameters in the argument list, the first being an int that gives a length specification, the second being the actual argument. For example, `printf("%*d", len, var)` where len=8 has the same effect as `printf("%8d", var)` – except of course that in the former the length can be adjusted at runtime. You do not need to use the * trick in this lab however.

%n has proven much more useful for format string attacks due to its ability to overwrite memory locations, than it has ever been for legitimate uses – but attempts to remove it in C have met with resistance because there are apparently legitimate programs that use it too. Re-implementations of printf for other languages can choose not to implement %n though: python's print / format mechanism does not have the %n "feature".

For more information on printf's many options, type "man 3 printf" or look on the internet.

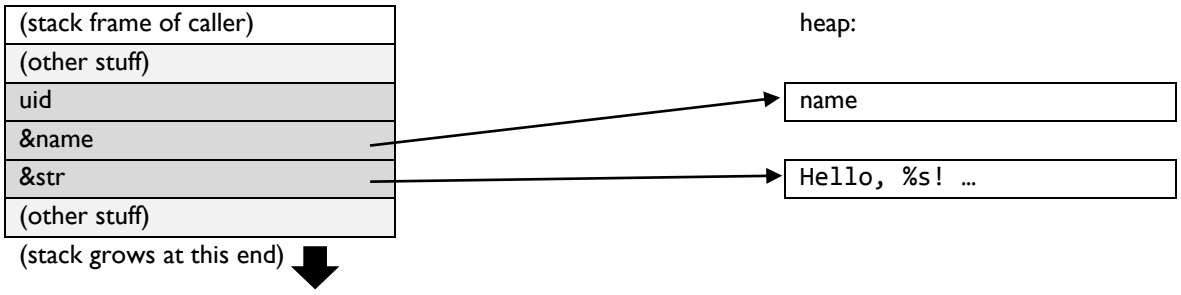
Background – printf, varargs and the stack

Printf uses C's varargs (variable number of arguments) mechanism. This lets it accept any number and type of arguments, but at the cost of no type-checking. Although the compiler can complain that e.g. `printf("Hello %s!")` is probably a mistake, as it knows the semantics of printf, the compiler cannot tell whether `printf(mystr, name)` is correct – it can only warn you that this is dangerous.

When you call

```
printf("Hello, %s! Your user id is %d.\n", name, uid);
```

the stack frame of printf looks something like this:



The integer *uid* is passed by value (copied onto the stack), as is the pointer *name* (of type char*, presumably) – but not the string that it points to. The parameters are pushed onto the stack in reverse order, followed by the format string itself. (This has nothing to do with printf, it is simply how varargs work.)

When printf executes, it sets up an internal pointer to its first argument, the address of the format string. Then it iterates over the format string. Characters that are not a % simply get printed out. Each time it encounters a % parameter in the format string (except %%) it moves its pointer up one step on the stack and processes the argument.

If you call printf with too few parameters, the effect is that the pointer ends up pointing at addresses on the stack beyond the printf parameters – C calls this "undefined behaviour", we call it a vulnerability. The effect is that each %-parameter in the format string that does not have a matching argument causes printf to read (or in the case of %n, even write) to a memory address related to something further up on the stack. Above the stack frame of printf lies the stack frame of the function that called it, with all its local variables that we can target in a format string attack.

The classical format string vulnerability is a line like

```
printf(str);
```

where str is user input. There is no reason ever to do this, as there's no formatting involved: since you're just printing a string, `puts(str)` would do the same job. Or `printf("%s", str)` if you must. Most compilers nowadays treat a one-argument printf call as an error, or at least warn you about it.

Background – setuid root

Why would we attack a program if we can just open the program file ourselves?

Normally, any program we run in a UNIX-based operating system has the same permissions as our user account. However, some tasks such as changing our password require access to files we can't normally write to, such as the system password file.

Setuid programs run as the program's owner instead of its caller. For example:

```
seed@seed:~$ls -l $(which passwd)
-rwsr-xr-x 1 root root 47032 Jan 27  2016 /usr/bin/passwd
```

The "s" in the fourth column indicates that this is a setuid program. The r-x in columns 8-10 means that any logged in user can execute this program, and make a copy of the program file and inspect, debug or modify this copy should they wish to – but making a copy creates a new file owned by themselves and turns off the setuid bit again. Only the original program will run with root rights. If there were a format string vulnerability in this program, we might be able to exploit it to gain root rights on the machine we're working on.

Of course we already have a way to gain root rights on the lab vm (sudo) but this is just to make setting up and investigating the attacks easier – a fully developed format string attack should of course work against a program on a machine where we have no legitimate way of getting root rights.

Tasks

Download the program formatstring.c from the unit website and compile it with

```
gcc formatstring.c -o formatstring
cp formatstring formatstr-root
sudo chown root formatstr-root
sudo chmod +s formatstr-root
```

Enter your password (dees) when prompted for the "sudo" and note the compiler warning after the first command. You now have two programs to play with, formatstring itself and a setuid-root version (formatstr-root).

Have a look at the program. It contains some "secret" values that we'll try to access and some user inputs (yes, there's a buffer overflow vulnerability here too, but that's not today's topic).

To make the attack slightly easier, the program prints out the memory addresses where the secrets are stored, when you run it. Notice that if you run the program several times, you will get different addresses each time.

Consider the following code in lines 27-34:

```
printf("Please enter a decimal integer\n");
scanf("%d", &int_input); /* getting an input from user */
printf("Please enter a string\n");
scanf("%s", user_input); /* getting a string from user */

/* Vulnerable place */
printf(user_input);
printf("\n");
```

The vulnerable printf directly prints out a string that came from the input, using it as a format string. Try running the program and entering a few strings, including ones like (without quotes) "%s" or "%x" and see what happens.

Your tasks for this lab are

1. With the unmodified setuid-root program, find ways to do the following:
    a. **Crash the program.**
    b. **Print out the value of secret[1].**
    c. **Modify the value of secret[1].**
    d. **Modify the value of secret[1] to a value of your choosing.**
2. Comment out lines 27-28:

```
printf("Please enter a decimal integer\n");
scanf("%d", &int_input); /* getting an input from user */
```

Then recompile the normal and setuid-root programs.

Next, we are going to turn off address space layout randomisation. Type the following command – it stays off until you reboot or reactivate it:

```
sudo sysctl -w kernel.randomize_va_space=0
```

Check if this works by running the formatstring program twice in a row – it should give you the same address for secret[0] both times.

Repeat tasks (a)-(d) for the modified program.

Note – you may get unlucky and encounter a fixed address for secret[1] which contains control characters such as 0x00 or 0x0c. If this happens, ask a lab assistant for help.

3. If you have completed tasks 1 and 2 and have time to spare, you can try either or both of the following optional extensions for extra credit:
    a. **Task 2 (modified program) but with ASLR turned on (set the value to 2).**
    b. **Write a program in a language of your choice that carries out attack (2b), i.e. it calls formatstr-root as a subprocess. You will need to use something like python's pexpect to handle bidirectional communication between programs. Upload your program as an extra file to SAFE and explain anything necessary to run it in your report.**

Hints

Your attacks must work against the setuid-root programs indicated but you are welcome to create variations on the programs to explore what is going on. For example, you can set the `-g` compiler option and then debug the (non-setuid) program with **`gdb formatstring`** to investigate its exact stack layout.

Instead of typing your attack by hand each time, you can put it in a file and run

```
$ ./formatstr-root < file
```

where "file" is the file with your attack strings. Place one input on each line in the file, for example if you want to answer 42 to the numeric input and "sss" to the string one, the file should look like this with a newline at the end of each line:

```
42
sss
```

This technique also allows you to create format strings with some non-printable characters. If you create a hex-encoded file "file.hex" like this:

```
61 73 64 66 0a
```

and run **`xxd -p -r file.hex > file`** then you will get the decoded data in "file". You can hex-encode too by leaving off the `-r` option. (You can add spaces between the bytes – they are ignored when decoding.) Use 0a for a newline in a hex-encoded file.

Certain bytes will not work in a format string even if you write the string in hex and decode it to a file. For example, a null byte (00) terminates a string in C. Other bytes are fine in format strings but scanf, which reads in your string in the first place, will stop if it sees them – this includes space (0x20), newline (0x0a) and 0x0c / 0x0d (these are used in newlines on other platforms). *If your secret[1] value lands at an address containing one of these bytes after you have turned ASLR off in task 2, speak to a lab assistant.*

The techniques you'll need to use include reading from a memory location on the stack and printing out its value; reading from a memory location not on the stack to print its value and writing to a memory location on the stack. What do the different format string parameters do from an attacker's perspective?

The secret values themselves are on the heap (since malloc is used to allocate them) but their address, in the form of the pointer *secret, is on the stack which is helpful for your format string attack. However, this is a pointer to `secret[0]`; the challenge in task 2 in particular is to attack `secret[1]` the address of which is not on the stack – not yet.

The format string itself comes from the variable `user_input` in main. This is a local array, not a malloc'd one, so the format string will be located in main's stack frame, just above that of printf. If we nudge printf's internal pointer up enough times, it will point at the format string itself, where we have almost complete control over the contents.

Coursework 1 notes

As explained in the general coursework guide on the unit website, for coursework 1 you have to write a technical part and a reflective part for this lab and the following one (so four parts in total) together with your group partner. Remember also to follow the general guidelines on the website, such as indicating your usernames and the distribution of work in the report.

For the technical part of the report for this lab, you should document

- How to make the attacks work – if it's a simple matter of choosing the right string such as in task (1a) then give the string; if it's a more involved procedure such as in task (1d) then give the procedure.
- What the attacks do / why they work – in this lab for example, how they interact with the stack layout.
- For extra credit, explain the principles involved, e.g. in this lab you could talk about what the uses of the different format specifiers %x, %s, %n etc. are for an attacker.

For the reflective part, see the guidance on the unit website.