System Security
Memory Vulnerabilities

Not graded

In this exercise session, we will perform different buffer overflows and format string attacks to understand how they can lead to real-world vulnerabilities [1, 2].

Setup. For this exercise, we will use a docker container. To get started, extract *memory-vuln.tar.gz*. Then, build and run the docker container.

```
docker build -t exercise5:v1 .
docker run -dit exercise5:v1
```

The exercise files are located at `/home/exercise5/` in the docker container. Note that, the files for Sections 1-4 are compiled using `-fno-stack-protector` and with debug symbols (`-g`), see the `Makefile`. Execute `make` in `/home/exercise5/` to recompile the binaries before you start this exercise. We have tested the contents in this sheet in the docker container. While it might be possible to run the binaries directly on your machines, note that the sizes, addresses and other behavior might be different.

1 Primer on stacks and heaps

For this part, use the `stack_heap` binary and the corresponding code in `stack_heap.c`.

Questions:

1. How many local variables are allocated in `foo()`? What are their sizes?

2. Use `gdb` to inspect the `stack_heap` binary. Step through the execution till the instruction pointer (IP) is at `<foo+12>`. Fill in the values in the yellow and grey boxes in Figure 1.

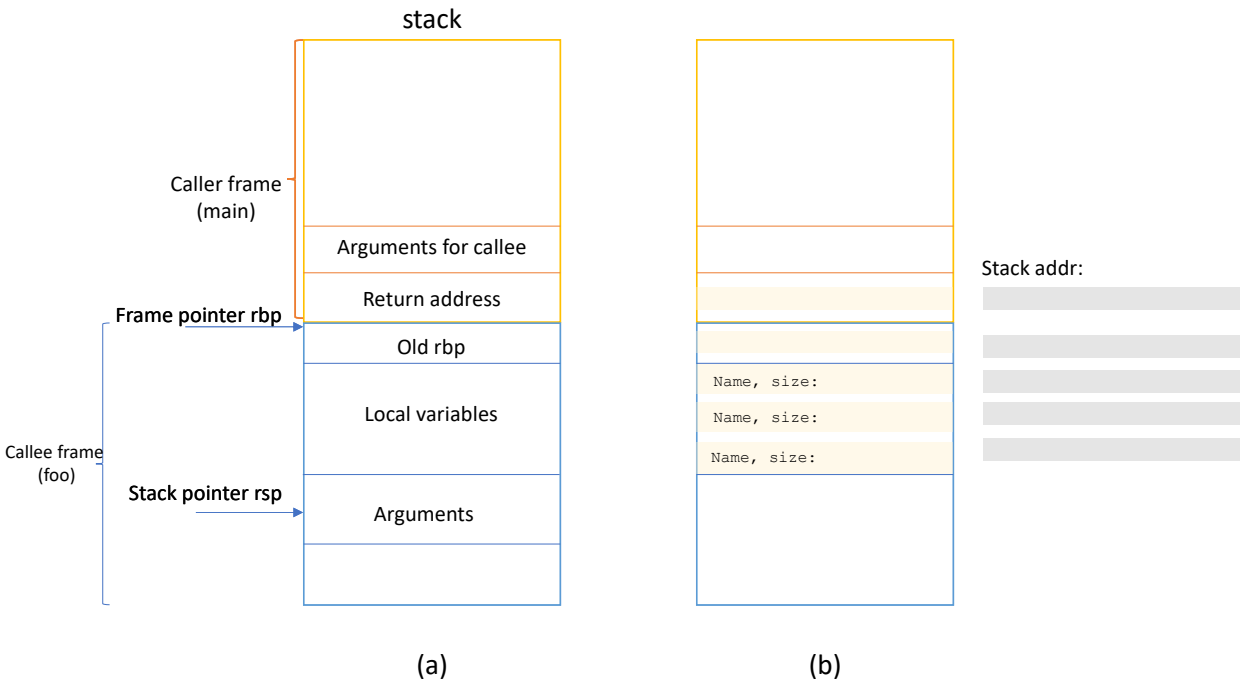

Figure 1: (a): generic layout of the caller and callee stack frame. (b): Fill this in

3. When the instruction pointer is at `<foo+12>`,
   (a) Where is `buf` stored?
   (b) Where is `buf_ptr` stored?
   (c) Which address does `buf_ptr` point to?

4. When the instruction pointer is at `<foo+12>`, `i` is not initialised yet. What is the value of i on the stack?

5. Now execute the function till `stack_heap.c:13`. Print the stack and look at the values of `buf`. Have they changed? Print the heap where `buf_ptr` points to. Do you see the same values?

2 Format String Vulnerability

For this part, use the `format_string` binary and the corresponding source code in `format_string.c`. The program has a format string vulnerability that can be used to leak the `password` on line 11.

Questions:

1. Create a string that prints out the password ("secret!") when given as input to the binary `format_string`. Note: multiple format strings can print the secret.

3 Overflow in the BSS segment

For this part, use the `overflow` binary and the corresponding source code in `overflow.c`. Read `overflow.c` carefully. Use `gdb` to inspect the binary.

Questions:

1. Where are the variables `PASSWORD` and `USER_PASSWORD` stored? What are their addresses?

2. As an attacker, you do not know the correct password. Can you execute `overflow` such that the program prints "Correct password!"?

3. Bonus: Try to create `buf_ptr` like in the previous section using `malloc`. Now change the function to copy the input string into the buffer instead of `USER_PASSWORD`. Will the same attack work and why?

4 Stack overflow

For this part, use the `stack_overflow_1` and `stack_overflow_2` binary and the corresponding source code in the c files. Note, `gdb` sets some environment variables that can change stack addresses. Therefore, if you use `gdb` to get absolute stack addresses for any of these questions, we recommend that you run the binary also in `gdb` to perform the exploits.

Part 1 For this part, use the `stack_overflow_1` and the code in the corresponding `stack_overflow_1.c` file. Read the code carefully. The program is similar to the previous question, but it reads the input password from the `user_password` file.

Questions

1. Where are the variables `password` and `user_password` stored?

2. As an attacker, you do not know the correct password. Can you execute `stack_overflow` such that the program prints "Correct password!" (e.g., by writing a string into `user_password` file)? If yes, what is the command? If not, why?

3. If `password` was a global variable, would your approach to the previous question still work?

Part 2 For this part, use the `stack_overflow_2` and the code in the corresponding `stack_overflow_2.c` file.

Questions

1. Observe that the declarations of `password` and `user_password` are reversed. What are their addresses?

2. Why would your exploit for Part 1 not work?

3. You cannot force the program to set `res = 0` and so print "Correct password!". Think about what other information that is used to determine the program flow is stored on the stack. For example, can you tweak the stack such that the condition `res=0` is never checked? *Hint: copy paste the following text into an editor* —

—

References

[1] Heap-based buffer overflow. https://cwe.mitre.org/data/definitions/122.html.

[2] Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html.