CSE 360 Project I

February 23, 2017

This project will be completed by groups of 3 students each, but in rare cases, groups may be permitted to have 2 or 4 members. Working in groups helps you learn from each other. Moreover, it allows the work to be split up, so that different exploits may be developed in parallel by different students. Note that there are three parts to this assignment, and each is comparable in terms of complexity/effort, although it may seem as if some parts are much longer than the others.

You are given a vulnerable program `vuln.c` and a vulnerable heap implementation `my_malloc.c`. These programs, together with a Makefile, are provided as a tar-gzipped archive. Note that the program and the vulnerabilities are based on the problems in the homework.

Note that `vuln` accepts commands on its input and executes them. Examine the source code to see what the commands are. Note that it uses `read` rather than `scanf` or `gets`. This means you can input arbitrary values as input, a capability you need if you want to input arbitrary binary data that may include code or pointer values.

There are three basic vulnerabilities that you can exploit; for full credit, you will need to develop exploits for each of them. The vulnerabilities are:

- a format string vulnerability in `main`,
- a heap overflow vulnerability in the version of `malloc` defined in `my_malloc.c` and used in `vuln.c`,
- a stack overflow vulnerability in `auth`.

The exploits you need to implement are described in more detail in separate sections below.

Note that you don't need to disable ASLR, stack protection or fool around with $W \oplus X$ to get your exploits to work. Instead, you will use the printf vulnerability to leak as much of the memory contents as you want. Initially, you will leak the contents of the stack. The stack will contain stack cookie — gcc uses the same value of the cookie for all functions, so you can read and reuse them. The stack will also contain return address, which is within code segment. By dumping code memory, you can read information such as the address of functions in libraries (e.g., `bcopy`), and from there, you can compute the location of a more useful function such as `execl`. Finally, to overcome $W \oplus X$, note that the Makefile already makes the stack executable. In addition, `my_malloc` ensures that its heap blocks are executable.

Your project submission will be in the form of exploit files. More details on the exact content/format will be posted next week.

1 Stack Smashing

Using the buffer overflow vulnerability in `auth`, implement the following:

- A simple stack smashing attack that executes injected code on the stack that calls `ownme()`. First do this by redefining `RANDOM` to be 0 in the Makefile. Once that works, try to revert it back to `random()` and see if you can compensate using a NOP-sled.

  For all of the following parts, define `RANDOM=0` in the Makefile.

- Use stack smashing to modify saved BP value on the stack frame of `auth` so that when control returns to `g`, you have control of the local variables of `g`, and can use this to set `s2` to `/bin/bash` even when `auth` returns 0.
- Use a return-to-libc attack that calls `execl` (or another function with a similar functionality) in `libc`, the standard C library. You should control the arguments so that you get a shell.
- Use a data-only-attack on the local variable `authd`. In particular, use stack smashing in `auth` to go past the stack frame of `auth` into its caller's frame, and modify the value of `authd` there.

2 Heap Overflow

Note that the heap overflow vulnerability resides within `heap_delete` function in `my_malloc.c`. This function is called within `my_malloc` as well as `my_free`. In theory, one could exploit it from either place. However, the heap bocks have to be arranged in a certain way in order for this work. So, you may need to use the `u`, `p` and `l` commands a few times to make sure that heap blocks are ordered in just the right way for your attack to work.

- Exercise a heap overflow in `my_malloc` to overwrite the pointer to `bcopy` in the global offset table (GOT) so that it instead points to `ownme`.
- Exercise a heap overflow in `my_free` to overwrite the pointer to `bcopy` in the global offset table (GOT) so that it instead points to `ownme`.
- Repeat one of the above, but overwrite the return address on the stack instead of a GOT address. Your exploit should result in your injected code exploit to be triggered when `main` returns. (As before, the exploit can simply call `owmne()`.

3 Format String Attack

Implement an attack that uses only the format string vulnerability. Your goal is to execute arbitrary code injected by the attacker. (Your code can simply call `ownme()`.) Follow the strategy given in your homework solution.

For this attack, do not overwrite the canary — you should target and modify the return address without changing any other value on the stack.