CSE 331: Project 2

Radu Sion, Soroush Meghdadizanjani

October 30, 2022

1 Overview

Thanks to Professor R. Sekar for providing the exploit programs!

In this project you will learn about stack smashing attacks and how an attacker can use a vulnerable program to exploit the system. It is recommended that this project be done by pairs of students. You can, of course, choose to do it individually, but it is obviously going to be more work. Besides, exploit writing is an inexact process, so there may be times when you get stuck. With two people working on the assignment, it is less likely that both will get stuck in the same way; and even if you do, you can work in parallel to find a work-around.

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 `vuln` accepts commands on its input and executes them. Examine the source code to see what the commands are. (Until you read that code, you cannot fully understand the rest of this project description.) *Unless you do the exploits yourself, you will not be able to grasp the complexities of exploit writing or be able to answer questions in exam related to this.* Also note that memory layouts are different for each group/individual, so the exploit that works for one group will fail for another group. This variation is implemented using the environment variable `GRP_ID` that should contain the group id that will be assigned to your group. Since the exploits are different with different groups, we have made a fully working sample exploit for the data-only exploit that overwrites authd. This exploit works when you set `GRP_ID` to 1000. This example will give you a road map on how to construct your exploit code, and how to structure it. (You will of course need to submit exploits that work for your assigned group id.) For simplicity, you can use the last 3 digits of your SBU ID as the group id. Unless there are people in the class for whom the group ID would conflict, we will reassign you a different group ID.

If you decide to do the project in groups/individually, please notify the TA via email about your group along with your SBU ID. For a group of 2 students, use the group ID (last 3 digits of SBU ID) which is smaller.

Note that `vuln` 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:

- a format string vulnerability in `main_loop`,
- 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`.

Some of these vulnerabilities can be exploited in more than one way.

Note that you don't need to disable ASLR, stack protection or fool around with $W \oplus X$ (stack attack mitigation techniques) 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 saved base pointer. By reading it, you can overcome randomization of the stack base address. To cope with possible randomization of code memory, you can read the return addresses off the stack. 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.

Note that Makefile automatically generates an assembly code version of `vuln` in `vuln.s`. To make the assembly file easier to understand, it now embeds source code lines within assembly, so that you will know what line of source code results in which assembly instructions.

2 Stack Smashing

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

- 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.
- Use a return-to-libc attack that returns to `ownme`. *Do not hard-code the address of* `ownme` *in your exploit.* Such a technique won't work if the base address of the executable is randomized. Instead, read the return address off the stack (using the format string vulnerability) and then compute the address of `ownme` from this information.
- A simple stack smashing attack that executes injected code on the stack that calls `ownme()`.
- 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 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.

Note that in some instances, you don't know the exact starting address of injected code. In those cases, attackers precede their code with a *NOP-sled.* This is simply a sequence of NOPs, which are 1-byte instructions in the x86 architecture. Now, you can jump into any byte of the NOP-sled, and then execution will flow through the NOPs to the following code.

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 injected code can simply call `ownme()`.

For this attack, *you should not overwrite the canary* — you should selectively target the return address of `main_loop`, so that execution is diverted to the injected code when the quit command is sent to `vuln`, and it returns from `main_loop`.

4 Submission

Your submission will be in the form of C-programs. In particular, for each exploit, you will create a version of `driver.c`. Compiling and running this exploit program should lead to a successful exploit. *Note that you need to submit the source code for the exploits.* You should not change `vuln.c` or any of the other material provided to you.

You should create a tar-gzipped archive of all your exploit programs. Give them descriptive names such as `driver-smash-data.c`, `driver-heap.c`, etc.

5 Tips

- Use the 32-bit VM image provided to you. Your submission will be tested on this VM, so you might as well work on the same VM. You can download the VM from here:
  - `https://drive.google.com/file/d/1Pl8Hjr7ovPNdgq0VVCBtr0hZqNcZt5fs/view`

  I have tested the exploits on a 64-bit Ubuntu system, when every thing is compiled with the `-m32` flag that produces 32-bit binaries. However, before you submit, please ensure that your exploits work correctly on the provided VM.

- **Don't change the** `Makefile`, except possibly for adding additional lines for compiling additional exploit programs.

- **Review carefully the example exploit program** `driver_authd_exp.c`. You will gain a better understanding of how to structure your exploits, and also save time on other exploits.

- You can print a specific offset that is, say, 100 words from the top of the stack using `printf("%100$x")` instead of having to use 100 instances of `%x`'s. (Note that this may end up printing something that is a few words off, say, 97 words from the top of the stack.)

- Within gdb, registers can be accessed by prefixing them with `$`, e.g., `print $esp` will print the stack pointer register.

- Within gdb, you can print arbitrary memory locations by casting them into pointers and dereferencing them, e.g., `print *(int *)0xbfffff7c`. You can control the format, e.g., print it in hex using `print /x *(int *)0xbfffff7c`.

- You need to use the printf vulnerability to leak several pieces of information. The first is the stack canary value. The second is the saved ebp value that you need in order to figure out the base of the stack frames. (You cannot hard-code stack base address because the stack base is (re)randomized on each execution.) Finally, you need to leak return address on the stack, or the address of library functions in the GOT (Global Offset Table).

  The driver program is necessary because of the need to leak these pieces information. You will structure your exploits as follows. First, you will use the `e` command to leak the above pieces of information. You will extract the information into variables in the `driver` program, which will then construct an exploit string and send it to `vuln`.

- You can debug an already running process by using `gdb` to attach to it. (Some times you may need `root` privilege to attach to an existing process.) To attach to an existing process, e.g., `vuln`, type `ps ax|grep vuln` at the bash command prompt. It will produce a list of processes that have the name `vuln`. Note down the pid, fire up gdb, and at its command line, type `attach` to that pid.

  This ability is invaluable for tracking down problems with your exploits.

- You can also use `objdump` to disassemble the executable. *An executable contains code that won't be in the object file* `vuln.o`*, or the assembly file* `vuln.s`. Use `objdump -d vuln` to disassemble the executable. Then you will see how library calls are made, and how you can hijack them.

Although the stack and code layout is going to be different for each team, the layout does not change from one run to another. So you can use gdb to figure out the layout once, and then use it repeatedly in your exploits. Specifically, you need to know the size of the stack frames of `main_loop` and `auth`, and you can find this by running `vuln` within gdb, setting break points in these functions, and printing the values of `ebp` and `esp` registers. Make sure that you print `esp` value after the calls to `alloca`. (This function allocates storage on the stack, and hence will change the value of `esp`.)

In order to succeed in this project, you have to get good at using gdb if you are not already there.

5.1 Working with assembly/object code

Some exploits require you to use binary code. You can do this by writing a small assembly code snippet and then compiling it using an assembler. One option is to use `as`, the default assembler on your system. You can invoke it as:

```
as -a --32 test.s
```

where `test.s` is the file containing your assembly code. This command dumps the assembled code on the screen. Note that `as` uses AT&T syntax for assembly. Alternatively, you can `nasm` which supports Intel format. (I have not used nasm.)

Instead of trying to use direct jumps or calls to absolute memory locations, you should try to use indirect jumps and indirect calls. First move the target address into a register, and then use an indirect jump or call using that register. Various other points to note:

- Make sure you get your assembly syntax right for various addressing modes and operands. Specifically, for `as`, make sure you prefix immediate operands with a `$`, and register operands with a `%`. For instance, `mov $0x20, %eax` moves the decimal number 32 into the register `eax`, while `mov 0x20, %eax` moves *the contents of memory location* `0x20` *into* `eax`. *Also make sure that you use a* `*` *for indirect calls and jumps, e.g.,* `call *%eax` *is an indirect call to the address contained in* `eax`. *(However,* `call *(%eax)` *first dereferences the location whose address is in* `eax`*, and then fetches the value stored at this memory location, and then calls that location.)*

- *You can use* `gdb` *to work at the assembly level. You can use* `layout asm` *to see your code in assembly. You can use* `stepi` *to single-step assembly instructions. The following pages may be helpful in this regard:*
  - *http://web.cecs.pdx.edu/~apt/cs491/gdb.pdf*
  - *https://sourceware.org/gdb/current/onlinedocs/gdb/Machine-Code.html*
  - *https://cs.nyu.edu/courses/fall03/V22.0201-003/c_att_syntax.html*

5.2 Additional Resources

Professor Sekar has made the following videos from his class available for reference. It is highly recommended that you go through the following videos to understand how to craft your exploit code and get a better understanding of each of the vulnerability in the program.

Exploit project Discussion

- **Project Overview**: Link 1, Link 2
- **Help session with gdb and code** Link 3, Link 4, Link 5
- **Follow-up discussion with Q&A session** Link 6
- **Behind the working of gdb** Link 7