COM-402 Stack Smashing and Fuzzing Homework

In this exercise, you will be exploiting buffer overflows and other memory-safety bugs to obtain unauthorized root access. More precisely, you will interact with some flawed C programs, the `targets`, divert their execution flow and spawn a root shell.

You will get four target programs to crack. The **first three** are mandatory in order to prepare for the quiz. The fourth one is optional (we will not ask questions about it).

You will also try to fuzz an executable in order to find an input that makes the binary crash. This exercise is also part of the material that we might reference in the quiz.

Setup

You will test your exploits within the Docker container we provide that has all the tools to build, run, and debug x86_32/i386 binaries. It also comes with Address Space Layout Randomization (ASLR) disabled. Note that such a setup is not really realistic in real-world deployments anymore, but it provides a solid and slightly simplified baseline for your experiments in this homework.

To run the container, you can simply execute in the terminal (in the directory with the `compose.yaml` after unpacking the handout archive):

```
docker compose run --rm -it stacksmashing
```

Accounts and File Locations

There are two accounts in the container:

- Unprivileged user account username:password: `user:user`
- Superuser account username:password: `root:root`

You'll find all the files required for the exercices in `/home/user/targets`, `/home/user/sploits` and `/home/user/fuzzing` in the container. If you modify any of the source files on your host, the changes are reflected in the container (and vice versa) via the mounts specified in the `compose.yaml`.

Target Programs

The `targets` directory contains the source code for the stack smashing targets, along with a Makefile for building them. In real life, most likely you wouldn't have access to the source code. In this homework, you can use it to get a better understanding of what is going on.

The `sploits` directory contains the exploit skeletons that you can use for exploiting the previously mentioned targets.

The `fuzzing` directory contains the source code for the fuzzing target, along with a Makefile for building it.

To build the targets, change to the corresponding directory and type `make` on the command line; the Makefile will take care of building the targets with the correct options. For the sake of this exercise, you should not change these sources nor the Makefile, but you can look at the process.

Then, to make the stack smashing target binaries owned by root, run *as user*:

```
su     # enter the root password; now you are "root"
make setuid
exit   # back to "user"
```

Any exploits you write should assume that the compiled target programs are owned by root.

Ask yourself: what is the setuid bit, and why is it needed here? Make sure to understand who *owns* and who *runs* the targets. Notably, how is it possible to run a *root* shell if the target is run by the *user*? What's the difference in your exploits if you skip the `make setuid` step?

Stack Smashing

For targets 1-4, you are expected to find the vulnerabilities present in the programs and to implement an exploit that causes the target program to spawn a shell.

Executing

You should run and debug the binaries wrapped in the `qemu-i386` emulator which emulates 32-bit x86 code. Due to the different architecture, an emulator is required if your host system is Arm-based (e.g., Apple Silicon). While x86_64-based hosts technically could also execute the code natively, we only support running it through the emulator (simply for providing a unified and reproducible environment across architectures). We provide two convenience commands for this purpose: `run` and `debug`.

Launch `run ./targetX` to execute a target binary, arguments can be given in the same manner:

```
run ./targetX arg1 arg2 ...
```

To run your exploit that you build and output in `sploitX.c`, you can therefore simply run

```
run ./targetX "$(~/sploits/sploitX)"
```

after compiling your exploit file. This command executes your exploit in a subshell and passes its output as an argument to the target.

The same procedure applies for `debug` (whereas `debug` already automatically launches the GNU debugger `gdb` for you).

These two commands facilitate the execution of the exploitable binaries on both x86 and Arm machines. You do not need to use them but we suggest doing so for simplicity. Feel free to check out the user's `.bashrc` in case you prefer not to use our commands or if you would like to adapt them.

Additionally, you are not required to build your exploit code in C with the provided exploit skeletons but we strongly suggest so (and that's the only solution we actively support).

Other Commands

- To compile your exploits/targets:

  ```
  make
  ```

- To remove the exploit/target binaries:

  ```
  make clean
  ```

Required Reading

Before reading on and starting to work on writing the exploits, you should read this article:

- *Smashing The Stack For Fun And Profit*, Aleph One

It is absolutely critical for this homework that you understand how the stack works and how data is arranged on the stack!

Shellcode

A buffer overflow can be used to make a program crash, to divert its execution to another place in the same program (how?), or to spawn other programs. The classical program to spawn is a shell, which gives you interactive access to the machine on which the shell is run.

To spawn another program, you will have to write a shellcode at the appropriate location in the memory of the program. Many variants exist, tailored to specific applications and scenarios; in this exercise, you are provided with an appropriate shellcode (the Aleph One shellcode) in `sploits/shellcode.h`.

Building Your Exploits

Each exploit, when run, should yield a root shell (`/bin/sh`). Once you get the shell, you can use the command `whoami` to verify that you are indeed `root`.

Using the GDB Debugger

To understand what is going on, it is helpful to run code through gdb, the standard debugger. Our previously mentioned `debug` command is a wrapper around gdb.

When running gdb, you should use the following procedure for setting breakpoints and debugging memory:

1. Set any breakpoints you want in the target (e.g. `break 15` to set a breakpoint at line 15, `break foo` to set a breakpoint at the start of function `foo`, or `break *0xdeadbeef` to set a breakpoint at address `0xdeadbeef`)
2. Resume execution by telling gdb `continue` (or just `c`)
3. Check the content of the stack with `x /20x $esp`

You should find the `x` command useful to examine memory (and the different ways you can print the contents such as `/a /i` after `x`). The `info register` command is useful for printing out the contents of registers such as `ebp` and `esp`.

Check the `disassemble` and `stepi` commands, they might be helpful. Same goes for `layout src`, `layout asm`, `layout regs`.

If you wish, you can instrument the target code with arbitrary assembly using the `__asm__()` pseudofunction. Be sure, however, that your final exploits work against the unmodified targets.

Suggested Reading

These articles and books should be helpful:

- *Smashing The Stack For Fun And Profit*, Aleph One
- *Basic Integer Overflows*, Blexim
- *Exploiting Format String Vulnerabilities*, Scut, Team Teso
- *Low-level Software Security by Example*, U. Erlingsson, Y. Younan, and F. Piessens
- *The Ethical Hacker's Handbook, Ch 11: Basic Linux Exploits*, A. Harper et al.

Additionally, these entries in the Phrack online journal might be useful:

- Aleph One, Smashing the Stack for Fun and Profit, Phrack 49 #14.
- klog, The Frame Pointer Overwrite, Phrack 55 #08.
- Bulba and Kil3r, Bypassing StackGuard and StackShield, Phrack 56 #0x05.
- Silvio Cesare, Shared Library Call Redirection via ELF PLT Infection, Phrack 56 #0x07.
- Michel Kaempf, Vudo - An Object Superstitiously Believed to Embody Magical Powers, Phrack 57 #0x08.
- Anonymous, Once Upon a free()..., Phrack 57 #0x09.
- Gera and Riq, Advances in Format String Exploiting, Phrack 59 #0x04.
- blexim, Basic Integer Overflows, Phrack 60 #0x10.

Fuzzing

For the `solveme` fuzzing target, you are supposed to gather first experiences with a fuzzer. The goal here is to find a secret input string that makes the program crash.

In contrast to the previous targets, you can execute the binary natively as `./solveme` after building it.

Note that this exercise serves only as a very short and simple introduction to using a fuzzer and a fuzzer's capabilities. If this sparked your interest, we recommend you to take CS-412 Software Security in the spring semester for more in-depth information and exercises on fuzzing.

Motivation

For this part of the homework, you are going to fuzz the `solveme` executable with AFL++.

If you take a look at the source code for `solveme`, you'll see a lot of constraints on the input and complicated control flow. In order to reach the `abort()` call in `foo()`, you can of course extract all the constraints from the source code and solve for an input that satisfies all of them manually. However, a way simpler method to find a corresponding input is to let a fuzzer take care of figuring out how to cover as much of the code in the executable as possible. In the process, the fuzzer will progress deeper and deeper into the nested control flow and potentially reach the desired location.

Executing

AFL++ is installed in the Docker container. When you type `make` for building the target binary, you build `solveme` with a special compiler: `afl-clang-fast`, which builds the binary with instrumentation for coverage feedback for greybox fuzzing.

As a result, the only things left to do are

1. Create input and output directories for AFL++: `mkdir in out`
2. Create at least one starting seed for AFL++, e.g., `echo foo > in/foo`
3. Start fuzzing: `afl-fuzz -i in -o out -- ./solveme`
4. Wait for AFL++ to report crashes (should only take a few minutes)!

While you're waiting for the crash counter in AFL++'s interface to increase, you can already take a look at the queue entries AFL++ creates along the way. The queue entries are located in `out/default/queue` and might already provide you with a hint which input might satisfy the constraints.

Once AFL++ reports a crash, you can inspect it in `out/default/crashes`. Does the crashing input contain something that looks familiar to you?

Note that there is not *one single* crashing input: the constraint system is slightly under-specified to allow AFL++ to find a possible input in a reasonable amount of time. As such,

if you restart the fuzzing campaign and find a crash again, it might look slightly different to the first one.

Recommended Reading

As you have seen in this exercise, fuzzers can solve even pretty complex constraints. The author of the original AFL (the precursor of AFL++) has a nice blog post on this: Pulling JPEGs out of thin air