Homework 1: Stack Smashing

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).

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.
To run the container, you can simply execute in the terminal:

```
docker run --security-opt seccomp=unconfined -it com402/hw1
```

You'll find all the files required for the exercices in `/home/user/targets` and `/home/user/sploits`. If you would like to be able to modify the sources on your host and only compile and run in the container, instead of the previous command run

```
docker run --security-opt seccomp=unconfined -it \
    -v <path_to_source>:/mnt \
    com402/hw1
```

where you substitute `<path_to_source>` with the **absolute** path on your PC where the source files are located. This mounts the provided host directory to the `/mnt` directory in the container.

Accounts and locations

There are two accounts in the container:

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

The container contains a folder `/home/user/targets/` containing four source files `target*.c` that need to be built (and attacked) as well as skeletons of your exploits `sploit*.c` located in `/home/user/sploits/`.

Target programs

The `targets` directory contains the source code for the 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.
To build the targets, change to the `targets/` 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 install the target binaries, run *as user*:

```
make install
```

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

```
su   # now you are "root"
make setuid
exit   # back to "user"
```

Your exploits should assume that the compiled target programs are owned by root.
Ask yourself: what is the setuid bit, and exactly why is it needed here? Make sure to understand who owns and runs the targets. Notably, how is it possible to run a *root* shell if the target is run by the *user*?

Executing

To run and debug the binaries we provide two commands: `run` and `debug`.
Launch `run /tmp/target1` to execute the binary, arguments can be given in the same manner:

```
run /tmp/target1 arg1 arg2 ...
```

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

```
run /tmp/targetX $(./sploitX)
```

after compiling your exploit file.
The same procedure applies for `debug`.
These commands facilitate the execution of the 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.

Other Commands

- To compile your exploits :

  ```
  make
  ```

- To remove the exploit binaries:

  ```
  make clean
  ```

- To remove the target binaries:

  ```
  make uninstall
  ```

Required reading

Before starting to work on the homework, you want to read this article:

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

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` sets a breakpoint at line 15 or `break foo` to set a breakpoint in function `foo`)
2. Resume execution by telling gdb `continue` (or just `c`)
3. Check the content of the stack by doing `x/20x $sp`

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.
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.

Acknowledgements

This assignment is based in part on materials from Prof. Hovav Shacham at UC San Diego, Prof. Dan Boneh at Stanford and Adam Everspaugh at UW Madison. Adapted for COM-402 by Ceyhun Alp and steadily improved by the COM-402 team.