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 targets 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 virtual machine (VM) we provide that is configured with Debian Lenny that has the address space layout randomization (ASLR) turned off.

- Download the image (hw1-vm.ova). Open it with Virtual Box.

Accounts and locations

There are two accounts on the machine:

- User account username/password: `user/user`
- Superuser account username/password: `root/root`

The VM originally 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`

Keyboard layout

By default, this VM is set to use the Swiss-French keyboard layout. If you want to change the layout inside the VM, use `kbd-config`:

```
su   # now you are "root"
kbd-config
```

Then, choose "select keymap from full list" and pick one. The standard American-English layout, for example, is called "pc / qwerty / US American / Standard / Standard".

Networking

It is convenient to access the VM via SSH (copy-paste works better, and you can transfer files both way using the `scp` command). You can do so with port forwarding.

If you're using *VirtualBox*, follow these steps to enable port forwarding from `host:2222` to `vm:22`:

- Open the settings of your image
- Go to the "Network" panel
- Choose "Advanced" and click on the "Port forwarding" button
- Add a forwarding rule (green "plus" button on the side)
- In the forwarding rule, leave IP addresses empty, set **Host port** to 2222, and **Guest port** to 22 (the default SSH port)
- Restart the virtual machine

Now, you can connect to your virtual machine via ssh:

```
ssh -p 2222 user@127.0.0.1
```

This is how you copy files **to** the VM (note the uppercase `P`):

```
scp -P 2222 <path_to_copy_on_host_OS> user@127.0.0.1:<path_to_copy_to_on_guest_OS>
```

Copy files **from** the VM:

```
scp -P 2222 user@127.0.0.1:<path_to_copy_on_guest_OS> <path_to_copy_to_on_host_OS>
```

Target programs

The `targets` directory in the VM 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.

Instead of exiting with the `exit` command, you can keep a separate terminal or virtual console open with a root login, and run `make setuid` from `/home/user/targets` directory in that terminal or console.

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*?

Then, run the first target in this shell, and keep it open. In another shell, you will write your attack.

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 `targets/shellcode.h`.

Building your exploits

Each exploit, when run in the VM with its target properly installed, 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.

A good way to run gdb is to use the `-e` and `-s` command line flags; for example, the command `gdb -e sploit3 -s target3` in the VM tells gdb to execute `sploit3` and use the symbol file in `target3`. These flags let you trace the execution of the `target3` after the sploit's memory image has been replaced with the target's through the `execve` system call.

When running gdb using these command-line flags, you should use the following procedure for setting breakpoints and debugging memory:

1. Tell gdb to notify you on `exec()`, by issuing the command `catch exec`
2. Run the program using (to your surprise) the command `run`. gdb will execute the sploit until the `execve` syscall, then return control to you
3. Set any breakpoints you want in the target (e.g. `break 15` sets a breakpoint at line 15)
4. Resume execution by telling gdb `continue` (or just `c`)

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.

If you try to set breakpoints before the `exec` boundary, you will get a segfault.

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.