Computer Security
Coursework Exercise CW3
Software Security

This document has been updated with remote working guidlines, due to measures taken against Covid-19.

The goal of this coursework is to gain practical experience with attacks that exploit software vulnerabilities, in particular buffer overflows. A buffer overflow occurs when a program attempts to write data beyond the bound of a fixed-length buffer – this can be exploited to alter the flow of the program, and even execute arbitrary code.

You are given five exploitable programs, which are installed with in the virtual machines provided for this coursework. These may be found in the `/task{1-5}/` directories on the virtual machine, with `setuid` for a respective user set. In each case, a secret should be extracted by exploiting the program in some way, by supplying it maliciously crafted inputs, either to spawn a shell, or extract the secret directly.

1 Virtual Machine Layout

In this coursework, you have three virtual machine scripts available to you: `lestrade`, `watson`, and `sherlock`. Each of these scripts can be found in `/afs/inf.ed.ac.uk/group/teaching/compsec/cw3/`, and each serves a different purpose:

- **lestrade**: This is a graphical virtual machine for working in, similar to coursework 2. It provides a basic GNOME environment for doing the coursework in.
- **watson**: This is a terminal-only virtual machine, designed primarily for remote use. As this coursework has no requirement for UI-driven programs, you may find it easier to work this, potentially more responsive environment.
- **sherlock**: Sherlock doesn't like doing as he's told, but is quick to judge and critique the works of others. **Sherlock is an automarker for this coursework.** If your submission is consistently accepted by this automarker, it will be marked correct. On the flip side, as the automarker is available, solutions *not* satisfying it will at best considered partially correct. **Note that only one VM can access your files at a time – to run sherlock, you need to first shut down your work VM!**

Both `lestrade` and `watson` optionally take a port as an argument – the VMs ssh port is forwarded to this port on localhost, allowing you to access it via ssh, including copying files from and to the VM.

In order to allow these attacks to be practically and reproducibly executed, address space layout randomisation (ASLR) has been disabled in the VMs, ensuring that in each execution, the same parts of a program get assigned to a consistent memory location.

For lestrade and watson, your username and password are both `student`, and *you have root access* via `sudo`. Be cautious with how you use it, however, as you will not have root access on sherlock, and we take care that it is not directly exploitable, and for testing purposes only! All work in your home directory is saved in the disk image `cshome-cw3.qcow2`, which you will be asked to submit at the end of the coursework. Anything outside of your home will reset with the VM.

2 Working Remotely

There are two primary ways to work remotely: via SSH, and by copying the virtual machines to your local PC.

2.1 Via SSH

The `watson` VM can be run via SSH on informatics servers, with a few exceptions. Notably, the exceptions are the XRDP server, and the `student.login` and `student.ssh` servers. Servers which do work are `student.compute`, and most lab machines, the names of which can be found at:

`https://mapp.betterinformatics.com/about`

While `lestrade` can also be run via SSH by using X11 forwarding (`ssh -X`), this will be very slow.

2.2 The packaged versions

To run locally on either Linux or Windows, copy one of the following bundles. The first comes with the graphical VM lestrade, while the second does not.

- `/afs/inf.ed.ac.uk/group/teaching/compsec/cw3/full.zip`
  Size: 1.7 GiB
  Sha256: `1089dfc468401969e021b7e40cee1c4edea29c642cdc126c335754a26afb5b82`
- `/afs/inf.ed.ac.uk/group/teaching/compsec/cw3/minimal.zip`
  Size: 540 MiB
  Sha256: `ca025d39ccf120cb82ba77f1790c029c0234420e2e0077b74ac52939a346627b`

2.3 Running locally on Linux

To run on a local linux distribution, ensure that `qemu-kvm` is installed. Further, a SPICE client, such as `spicy` or `vinagre` needs to be installed for `lestrade`. Also ensure your use is in the KVM group (`sudo usermod -aG kvm $(whoami)`), and reboot if necessary.

To get started, run the following commands:

```
$ mkdir cs-cw3
$ cd cs-cw3
$ scp s<student id>@student.ssh.inf.ed.ac.uk:/afs/inf.ed.ac.uk/
group/teaching/compsec/cw3/<version>.zip .
$ sha256sum <version>.zip
$ unzip <version>.zip
```

Next, edit `qemu.env` to point to your local installation of Qemu, and your SPICE client. Unfortunately, the values here differ for each distro, so we cannot give universal defaults. Now, you should be able to run the `watson`, `sherlock`, and (possibly) `lestrade` scripts locally.

2.4 Running locally on Windows 10[1]

Download and install Qemu for windows 10 from `https://qemu.weilnetz.de/w64/`. Install to the default location, or otherwise change the referenced location in the scripts later.

Ensure powershell allows the execution of scripts, by opening a powershell as admin, and running `Set-ExecutionPolicy unrestricted`.

Copy the version you want to run over, then unpack it. To copy, use the command:

```
$ scp.exe s<student id>@student.ssh.inf.ed.ac.uk:/afs/inf.ed.ac.uk/
group/teaching/compsec/cw3/<version>.zip
$ Get-FileHash -Path .\<version>.zip -Algorithm SHA256 .
```

Run the scripts `watson.ps1`, `lestrade.ps1`, and `sherlock.ps1`. If you installed Qemu to a non-standard location, edit the `qemu.env.ps1` file to set `$qemu` to the correct location.

Hardware virtualisation is not enabled for Windows 10, so the VMs will start quite slowly.

3 Templates

On first starting the VM, templates for each of the exercise submissions should be found in your home directory. Each task has it's own sub-directory for you to work in, and contains a script `attack.sh`. This script will be executed by the automarker, and should output the secret to its `stdout`. Beyond this, you are free to use what you want – provided the tools are preinstalled on the VM. This includes using your own compiled binaries if you wish, although you will not get partial marks if we can't see the source! Notable tools installed include netcat (`nc`), and `gdb`.

Some of the template scripts include a part using the `env` command to clear the environment. This is to ensure the environment the automarker runs the attack in, and the environment you test it in are the same. Take care to also make this consistent with your gdb environment! If you remove this part, you do so at your own peril, and you may experience an attack that works fine for you being turned down by the automarker.

[1] Special thanks to Vidminas Mikucionis for their help on adapting the coursework to Windows 10.

4 Shellcode

In task 3 you will have to deploy some shellcode, and while we do not ask you to come up with this yourself, we will briefly elaborate what this code does. These are x86 instructions, that, when executed, have the same affect as executing the following C function:

```
int main() {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  setreuid(geteuid(), geteuid());
  execve(name[0], name, NULL);
}
```

This program does two things: First, it sets the current user id the the effective user ID. This ensures that the shell does not drop privileges when it is started, i.e. reverting to an unprivileged user. In particular bash does this, which supplies `/bin/sh` in our VM. Second, it executes `/bin/sh` directly.

Good shellcode does not contain `NULL` bytes, as these will not typically be copied by C programs – for this reason just compiling the above is usually not sufficient. We provide the below shellcode, and in task 3, this is already available in the task template.

```
/* geteuid() */
"\x6a\x31"     // push $0x31;
"\x58"         // pop %eax;
"\x99"         // cltd;
"\xcd\x80"     // int $0x80;

/* setreuid() */
"\x89\xc3"     // mov %eax, %ebx;
"\x89\xc1"     // mov %eax, %ecx;
"\x6a\x46"     // push $0x46;
"\x58"         // pop %eax;
"\xcd\x80"     // int $0x80;

/* execve("/bin/sh", 0, 0) */
"\xb0\x0b"     // mov $0x0b, %al;
"\x52"         // push %edx;
"\x68n/sh"     // push $0x68732f6e;
"\x68//bi"     // push $0x69622f2f;
"\x89\xe3"     // mov %esp, %ebx;
"\x89\xd1"     // mov %edx, %ecx
"\xcd\x80"     // int $0x80;
```

5 The Vulnerable Programs

Task 1 This is just for warming up. This program is fund in `/task1/vuln`, and waits for a password (stored in `/task1/password.txt`), and if the correct

password is entered by the user, it prints the secret. The program has a buffer overflow vulnerability, making it possible – without knowing the password – to allow it to log in. **Important:** the real password will be changed for marking! The `pidgeon` and `rooster` various **will not**.

Task 2 This program takes the user's name as a command line argument, copies it to a buffer, and then welcomes the user. This program is vulnerable to a buffer overflow attack. Your attack script should call this program with a carefully crafted argument, such that it overwrites the return address on the stack, and returns to the `read_secret` function, instead of back to `main`. Stack canaries are not enabled in this task.

Task 3 For this program, there is no helpful function for extracting information, meaning you must inject your own shellcode, jump to this on-stack instead of a pre-existing function. Furthermore, from this task on, the program is compiled with GCC's stack protection enabled, which inserts stack canaries after the return address on stack. Your attack script will input a malicious argument, such that the program loads and jumps into your shellcode, while avoiding these canaries. The template provided already contains some boilerplate code to then read the secret from this shell.

Task 4 This program functions similarly to the previous script, however execution of code on the stack has been disabled! Instead, you should perform a return-to-libc attack. In particular, you should overwrite the stack such that execution returns into the `libc` function `system()`, and make this function believe it received `"/bin/sh"` as an argument. For this purpose, you'll need to find the locations of these in memory, and analyse how calls to `system()` would ordinarily function. Your program should use this return technique to spawn a shell – in this case the program itself forces that the real uid is set, ensuring the shell will be as the `task4` user.

Task 5 This task does not have a binary to directly exploit, but instead runs a small TCP server on the local port 4040. This program accepts requests the read sections of a secret file, but denies requests to some parts. It is however vulnerable to an attack allowing you to read this part too! Your attack script should connect to the server using netcat, and submit a malicious request, extracting the secret string.

6 Submission Instructions

Once you have completed your exploits, you should first test them against the automarker! Run `/afs/inf.ed.ac.uk/group/teaching/compsec/cw3/sherlock`, which will give you a list of whether you passed each respective task. If you are satisfied, and want to submit, run

```
submit cs cw3 ~/cshome-cw3.qcow2
```

If you are working on your own machine, copy this file over to a DICE machine first. On Linux, it is found in the same location. On Windows, it is found in your Documents folder.

The submission deadline is 3rd April, at 16:00, *although a further extension is currently being negotiated due to the impact of the strike.* A further extension is therefore possible, but not certain! **Please be aware:** The actual marking will generate *new random secrets.* Your secrets must therefore come from executing the attack, and not, for instance, reading them with root and simply echoing them!