

# Computer Security

## Coursework Exercise CW3

### Binary exploits

The goal of this coursework is to gain practical experience with attacks that exploit software vulnerabilities, in particular buffer overruns. A buffer overrun occurs when a program attempts to read or write data beyond the bounds 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 within the virtual machine provided for this coursework. Each of you will work on slightly different exploitable programs. You will find the five programs you need to work on in the `/task{1-5}/[yourUUN]` directories on the provided 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

For this assignment we are using a Virtual Machine called `B0_2324` inside VirtualBox. The machine comes as an `*.ova` file, which includes the virtual hard drive as well as all necessary settings for VirtualBox. For `B0_2324` your username is `student` and your password is `CSec-cw3#`, and you have root access via `sudo`. Be cautious with how you use it, however, as you will not have root access on our automarker!

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

To further protect against buffer overflow attacks and other attacks that use shell programs, many shell programs automatically drop their privileges when invoked. Therefore, even if you can “fool” a privileged `setuid` program to invoke a shell, you might not be able to retain the privileges within the shell. This protection scheme is implemented in `/bin/bash`. In Ubuntu, `/bin/sh` is actually a symbolic link to `/bin/bash`. For this coursework, and in order to really realise how much damage this sort of attacks can do, we use another shell program (the `zsh`), instead of `/bin/bash`. The preconfigured Ubuntu virtual machines contains a `zsh` installation.

**Templates** On first starting the VM, templates for each of the exercise submissions should be found in the `cw3/` folder in your home directory. For each task you will find a script `task{1-5}.sh`. This script will be executed by our 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 `gdb`, `ghidra`, and `pwntools`.

**Important** You must run your attacks inside the provided VM. It is very important that your attack programs work in the provided `B0_2324` VM. This is because, the compiler version, the operating system and the installed libraries will affect the exact location of code on the stack. The VM provides an identical environment to the one in which we will test your code for marking it.

Some of the template scripts include a part using the `env` command to clear the environment. This is to ensure the environment our 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 our automarker.

## 2 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 effect 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" // cld;
"\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;
```

## 3 The Vulnerable Programs

In each of these four tasks we expect you to steal the secret in `/task{1-5}/secret.txt`. Each task is worth 20% of the total mark.

**Task 1** This is just a warm-up. The program in `/task1/[yourUUN]/vuln` 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. [20 marks]

**Important:** the real password will be changed for marking! The `pidgeon` and `rooster` variables **will not**.

**Task 2** The program in `/task2/[yourUUN]/vuln` 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. [20 marks]

**Task 3** For the program in `/task3/[yourUUN]/vuln`, 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. [20 marks]

**Task 4** The program in `/task4/[yourUUN]/vuln` functions similarly to the previous one, 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 `root` user. [20 marks]

**Task 5** For task 5 you are not given access to the source of the executables. You will find them in `/task5/[yourUUN]/vuln`. You can use Ghidra. Ghidra is a an open source reverse engineering tool developed by the NSA. With this task you will see how it is possible to bypass ASLR, so you will need to enable ASLR. The automarker will test your attack scripts for this task with ASLR enable. To enable ASLR in the current session, use the following command: [20 marks]

```
echo 2 | tee /proc/sys/kernel/randomize_va_space
```

To make change permanent (across reboots), use the following command:

```
echo "kernel.randomize_va_space = 2" >> /etc/sysctl.d/01-disable-aslr.conf
```

**Important Note.** This last question is substantially more challenging than the previous ones, and may require a considerable amount of time, creativity, further reading to complete. It is intended to challenge students who already feel solid on the basic material in the course, and want to go further. **Do not** attempt this question unless you have completed all the previous questions, and are sure that you've done a good job on those. Also, do not attempt this question if you have personally spent 12 or more hours on the assignment already. Unless you really whizzed through the earlier questions, please stop now and spend your time and energy revising other course materials or getting more sleep. Both are likely to have a much bigger impact on your final mark.

## 4 Submission Instructions

Once you have completed your exploits, you should first test them against a clean B0\_2324 VM. To retrieve your attack scripts `task{1-5}` from the VM you just need to change the settings Devices ↔ Shared Folders ↔ Shared Folders Settings. If you are satisfied, and want to submit, you go on Learn. Under Course Content, you will find in the Assignment folder, the “CW3 - Binary exploits” box. Learn does not support `.sh` files. You will need to create a zip archive with your attack scripts (create an archive of your shell script files rather than an directory), and you will name it `UUN.zip` which you will be able to submit. You will find further guidance on how to submit here: <https://edin.ac/47ddcYY>

The submission deadline is 24th November, at 12:00. **Please be aware:** The automarker will generate new random secrets, different for every student. Your secrets must therefore come from executing the

attack on your exploitable programs specifically, and not, for instance, reading them with root and simply echoing them!

## A Running on DICE Machines

**Note** - The setup given here is adapted from a previous iteration of a similar assignment. It is in theory usable (it worked for me), but it might require minor tweaks, that we'll have to discover together :-)

The VM is accessible from all DICE machines here `/disk/scratch/software_security/`, in a compressed virtual appliance format. VirtualBox is already installed on all DICE computers. Once VirtualBox is running, select File  $\leftrightarrow$  Import Appliance and select the file `/disk/scratch/software_security/B0_2324.ova`. The only property in the configuration screen (Appliance settings) you should change is the location of the virtual hard drive. On DICE computers a home directory account does not have enough quota to store the hard drive, so you will need to store the virtual disk locally on the workstation. The disk takes up about 17G once expanded. In the Appliance settings screen change the path from, e.g.

`/afs/inf.ed.ac.uk/user/sXX/sXXXXXX/VirtualBoxVMs/`

to

`/disk/scratch/software_security/tmp/sXXXXXX/VirtualBoxVMs/`

(you may need to create the directory `/disk/scratch/software_security/tmp/sXXXXXX` first). The import operation can take several minutes and need about 17G of disk space per machine.

On DICE machines the virtual disk has to be stored on the local disk. That means that you will not be able to access the virtual machine on any other DICE computer than the one you did this setup on. If you want to use your virtual machine from another computer you will have to log in to the machine using `ssh -X <computername>`, or copy the disk files to another machine with `scp`; so make sure you remember the name of the computer you used!

## B Running on your personal machines

To get the coursework running:

1. Download VirtualBox (version 6.1.38) at <https://www.virtualbox.org/>
2. Download the virtual appliance at <https://edin.ac/3spFNem>.
3. Run VirtualBox.
4. Select File  $\leftrightarrow$  Import Appliance and select the file `B0_2324.ova`.