Secure Programming Lab 1: Data Corruption

School of Informatics, University of Edinburgh
David Aspinall, Arthur Chan and Rui Li

3pm-6pm, 7th February 2017

This is the first laboratory session out of three in the Secure Programming course. Lab exercises are a key part of the delivery of this course. Your work in labs does not contribute towards the final assessment mark, but you are expected to take part to help your understanding of the material. This will be needed for the upcoming coursework assignment which does contribute towards the final mark, as well develop knowledge that will be required in the exam.

- **Guided introduction.** The lab will include a short introduction, giving some hints about the exercises. This will be delivered in groups, but please arrive on time to make sure you've seen it (or ask the demonstrators).
- **Tools and techniques.** There are some pointers to useful tools in the exercises and in the appendix at the end of this handout.
- **Working together is encouraged**. We want to foster a supportive learning environment. Students who have prior knowledge or expertise are especially encouraged to work with others. Collaborating on the exercises may help you to think more deeply about the problems by discussing different aspects, as well as sharing existing knowledge.
- **Course staff will be on hand**. We will be here to discuss your progress and help with solving the problems. Detailed help will only be available during the timetabled labs.
- **Submit answers**. There are **checkpoint questions** in each exercise which you can answer to measure your progress. You may submit answers using the electronic submission mechanism or simply discuss them in labs. Submitting answers will allow us to give feedback at the next session.

As the work is intended to be completed in the lab and not take more of your time, there will be a short deadline for submissions. For this lab session, submissions will be accepted until **4pm, Monday 13th February**.

To see this handout online, visit http://www.inf.ed.ac.uk/teaching/courses/sp/2016/labs/lab1/

Virtual machinery

For this lab we are providing some code and a virtual machine for you to use. The VM has two users, `user` and `root`. The passwords are the same as their usernames. To install the VM, you should use a virtual disk file stored in the scratch space on your machine, and let Virtual Box use it from there.

To do this you will need to change some settings in VirtualBox. First set Settings→General→Default Machine Folder to

`/tmp/virtualbox`

Then import the appliance from the file:

`/afs/inf.ed.ac.uk/group/teaching/module-sp/SecureProgramming-1.ova`

The machine is set to use NAT. Once started you can either use the console window, or SSH in via your local machine over port 2222 (recommended).

```
ssh -p 2222 user@localhost
```

Lab exercises are in the folders in `/home/user/` on the VM. To compile the examples, just type `make`. You may need to use `chmod` or `chown` command to change the binary permissions and privileges for some question after `make` command is executed.

Beware that `/tmp` is local to your current workstation, it is not backed up. So you should save any work that you do inside the virtual machine (edited source files, etc) in your DICE home directory. We also recommend keeping a **lab notebook** with notes of what you have done during the lab exercises – it can be a physical notebook or simply a file containing a record of what you did.

We've supplied some tools to make things easier but feel free to install additional software in the VM. The package manager is called `yaourt`.

Driving without brakes

The machines (and exercises) have been configured to disable some preventative security measures. This makes exploits easier, so you can try out some of the classical exploitation techniques. For example, ASLR is disabled in the Linux kernel of our VM (`/proc/sys/kernel/randomize_va_space = 0`).

On a modern Linux system classic overflow attacks should be hard or impossible. But some cut-down operating systems and embedded systems are either still vulnerable or have only recently implemented protection. In Android, for example, before version 4 ASLR wasn't used and it wasn't fully enabled until 4.1. As another example, Show 93 of Gary McGraw's *Silver Bullet Security Podcast* describes remotely exploiting a car's brake system by using multiple buffer overflows!

Warmup Exercise (optional)

This warmup exercise helps you to understand (or recap) some Unix permission notions and some basics of buffer overflow that will give hints for the main exercises. If you are familiar with Linux/Unix, this part may be easy and you might want to skip to the main exercise. Otherwise you may need to consult some online resources to answer the questions here. There are plenty of pointers; Wikipedia is a good place to start.

Part 1: effective user id

The source and compiled binary for the programs we will look at are in `/home/user/Exercise-0.1`. First, consider the two binaries `root` and `user` which were compiled from the same source code, `source.c`.

Checkpoint 1. Briefly explain what is an effective user id and why the two binaries compiled from the same source code will print a different effective user id when they are executed.

Let's print out the long listing details of each binary (`ls -l`)

Checkpoint 2. Spot the difference of the permissions on the two binary files. Which part shows that their effective user id are different?

Examine the `Makefile` to see how the binary is compiled.

Checkpoint 3. Briefly explain the usage of `chown` and `chmod` operation in the `Makefile`.

From the long list of binary `root` you can see that the execution privilege has been set to 's' instead of the normal 'x'. This is named as 'setuid / setgid bit'.

Checkpoint 4. What is a setuid / setgid big and what are the difference of running the same binary with or without setuid / setgid big set?

Checkpoint 5. What is the security concern for a vulnerable program running with a setuid / setgid bit?

There is a binary `open` compiled from `open.c` try to open a file `secret`. You can run the binary and you will receive the "Permission Denied" error message because the file `secret` is only readable with `root` permission and privilege.

Checkpoint 6. Use the setuid / setgid bit technique to allow the `open` binary to read the file `secret`. State the command you have used.

Part 2: simple buffer overflow

The source and compiled binary for the program are in `/home/user/Exercise-0.2`. Take a look of the source code. You can see that the variable password is an array with the size of 10. It is possible to overflow the password variable because the program does not check the length of the user input before copying it to the password variable.

Checkpoint 1. How many user inputted characters are allowed to store in the array `password` without overflowing it?

Checkpoint 2. If the length of the user input exceed the size of `password`, where will strcpy() store the remaining characters?

Now, try to provide an user input to perform an buffer overflow attack. It is possible to corrupt the checking mechanism and let the program prints `Correct Password` even if you are providing a wrong password.

Checkpoint 3. State the user input you have used and briefly describe how it creates an attack.

Exercise 1

The VM contains a simple *Notice Board* system for a shared machine. It runs as the root user and allows any other user on the machine to append notes to a notice board file stored in `/tmp/noticeboard.txt` which has permissions `-rw-r--r--` (why?). The program is written in C. See the Appendix for some tips on some useful tools installed on the VM.

Part 1: classic stack overflow

The source and compiled binary for the program are in `/home/user/Exercise-1.1`. Take a look at the compiled binary: it is *setuid root* (the effective user id of the binary will set to root when it is executed), so it will have super-user permissions when it executes and can do anything on the system.

Useful Note: If you experience permission problems at some stage, try `sudo rm -f /tmp/noticeboard.txt` to remove the file (and try to understand why!).

Checkpoint 1. How can you tell the program will run as root permissions, and how can you make a compiled program run as *setuid root*? You will need this step if you recompile the code.

Let's examine the binary a bit more with a couple of tools.

- The `checksec.sh` script can help you see what protections are enabled (or not!) in the binary. Run this on the `noticeboard` executable.
- The RATS *Rough Auditing Tool for Security* (`rats`) can help find suspicious API calls in code. Run this on the `noticeboard.c` source.

Checkpoint 2. Briefly explain the output of these tools, and how the compiler flags influence the output of `checksec.sh` (look at the `Makefile` to see how `gcc` was invoked). Remember to give `setuid root` permission to the `noticeboard` binary if you recompile it.

The program contains a buffer overflow vulnerability which you should be able to see immediately. The main task of this exercise is to demonstrate this vulnerability by exploiting the program so that it launches a shell with root privileges. Here are the suggested steps:

1. Trigger a segmentation fault in the running program.
2. Use `objdump` and the debugger GDB and to inspect the code and memory layout. For example, launch the program in GDB, and find where the data is stored on the stack, and where the return address from `main` is.
3. Gain control of the instruction pointer. Do that by demonstrating how to overwrite the return address of `main` with any address of your choosing, e.g., `0xBADC0DE5`. Trace the execution of the program with the attack argument to check that this address is jumped to.

4. Direct execution towards some *shellcode* to execute a root shell. The shellcode could be passed in the overflow argument to execute on the stack, or set in some other way (e.g., an environment variable).
5. If your exploit is success, you should able to open a root shell, you can use the command `whoami` to check the shell owner to see if you have successfully spawned a root shell.

   Hints:

   Environment variables are really useful here. They can be defined in the command line by `VARIABLE_NAME=string`, and they get loaded into every program you run. If you're interested you should go and figure out precisely how they get loaded, but for now you might want to know that there is a secret `environ` global variable in all C programs, that it points to the environment variables, and that its address is predictable. It's a bit easier than using sleds (landing areas of NOPs).

   Setuid programs have their full path expanded when being run so `./` becomes `$PWD/`.

For step 4, you will need to find or write some shellcode. Samples can easily be found online but we encourage you to have a go at writing your own. A partial example was given in the lectures.

Checkpoint 3. Explain what your shellcode does and how you made it.

Checkpoint 4. Explain how your exploit works.

Finally, you should give a patch which fixes the vulnerability in the source program. Take a copy of the program, edit it and re-compile, then verify that it indeed prevents your exploit.

Checkpoint 5. Provide your patch to fix the notice board program (use `diff -c <oldprogram> <newprogram>`).

We have provided an exploit to check your fix. The exploit script is encrpyted and you can decrpyt it by using the gpg command. Just provide the encrpyted package as the first argument. The password needed for the decrpytion will be announced at the lab time. You can also ask us if you finish all the above checkpoints before we announce the password.

Part 2: another vulnerability

Rather than patching the vulnerability, the notice board program can be compiled with some of the standard overflow protection mechanisms provided by the compiler.

Consider next the new version of the program given in `/home/user/Exercise-1.2` on the VM. The programmer of this version read on a programming blog that global variables are dangerous, so the code has been re-written to avoid using them.

Unfortunately, this new version still has vulnerabilities, especially if it runs as setuid root. Take a look again at the binary and source code.

Checkpoint 1. Identify the security flaw in the new version of the compiled `noticeboard` program. Explain what it allows and demonstrate an exploit that compromises the standard system security.

In fact, the code even be exploited to give an attacker a shell. There is more than one way to do it, but it is a bit intricate. If you want to give it a go, feel free to try (we'll try and help!)

> *Hints*
>
> Just because you used shell code last time, doesn't mean you need it this time. Figure out what the vulnerability lets you do, and then think about how you could abuse that.

Checkpoint 2 (optional). Briefly, explain how your root shell exploit works.

Checkpoint 3. Give a patch which fixes the second version of `noticeboard.c`.

After finishing the above checkpoints, you can test with the exploit script. Wait for the announcement of password or ask us if you finish far earlier.

Exercise 2

To avoid low-level vulnerabilities that lurk when C programming, the notice board program has now been re-written in Java. It takes the form of a *server* (which writes the messages) and a *client* program (which sends notices to the board). The client and server use a very simple message format to communicate.

Unfortunately, there is a vulnerability in this Java version of the program. A malicious user can cause the server program to crash. Since the program works over the network, this attack can be conducted remotely.

You should find this vulnerability, create an exploit to demonstrate it, patch the server program and then verify your patch prevents your exploit.

Checkpoint 1. Explain the format of the messages sent by the client.

Checkpoint 2. Provide a program (or shell script) which crashes the server remotely.

Checkpoint 3. Give a patch to fix the problem(s).

After finishing the above checkpoints, you can test your patch with the encrypted exploit script. Wait for the announcement of the password or ask us if you finish far earlier.

Note
You can compile the server / client code by the `make` command.
You can run the server by `java Server <port number>`.
You can run the client by `java Client <host> <port number>`.

Exercise 3 (Advanced, Optional)

This exercise is a C program with another stack overflow vulnerability but this time there is a catch: writable memory is no longer executable. To get a shell this time you will need to utilise a technique called *return-to-libc*.

Return-to-libc attacks are a simplified form of return-oriented-programming. The idea is as follows. Programs and their libraries are typically large; often there is enough code already in memory from which an attacker could chain together fragments to make a malicious program. This means that using shellcode is unnecessary, and (in general) that CPU memory protection separating executable memory from data memory may not be effective.

In return-to-libc attacks we modify the stack in such a way so that after the buffer overflow the stack is set up to make a number of calls to libc functions when the overflown function returns.

The source and ready-compiled binary for the program are in `/home/user/Exercise-3`. By following the checkpoint questions you should be able to construct a return-to-libc exploit.

Checkpoint 1. How do you get the address of a `/bin/sh` string, and if you can't find one in memory how can you inject one?

Checkpoint 2. Where does the `system` function exist in libc? Where is it loaded in your program?

Checkpoint 3. How do you call `system` as you return from the overflown function with your string as its argument?

Checkpoint 4. A program which crashes may leave a log file somewhere. You should also ensure your program exits cleanly. How do you do this?

Note: this is a more advanced technique in buffer overflow and the checkpoint hints do not form a complete tutorial, intentionally, so that you can figure out some of the mechanisms yourself. If you find yourself spending too long on some of the steps or would like to look at the solution first, you can ask for the password to decrypt the exploit script and have a look at our solution.

Exercise 4 (Optional)

In this exercise we look at a real flaw in an old version of the OpenSSL library. OpenSSL is the main open source implementation of SSL/TLS, and is a ubiquitous part of many applications that use secure communication on the Internet.

The flaw might have been noticed by attempted attacks leaving evidence in system log files, in this scenario:

- A company's web servers seem to be crashing a lot.
- The logs indicate that something is causing memory to be corrupted; it seems like it happens whenever certain X.509 certificates are presented to the server.

We have provided the source code for the relevant version of OpenSSL on the virtual machine in the directory `/home/user/Exercise-4`, for you to look at. The vulnerability is inside the C function `asn1_d2i_read_bio`.

Checkpoint 1. Identify the security flaw in the code, and provide the relevant CVE number.

Checkpoint 2. Briefly summarise the problem and explain and why it is a security flaw.

Checkpoint 3. Give a recommendation for a way to repair the problem.

Checkpoint 4 (very optional). Build a *proof-of-concept* to demonstrate the security flaw and explain how it might be exploited; check that your repair (or the current released version) prevents your attack.

Appendix: Useful commands

General linux

make Builds your code
gcc <source-code> C compiler
  -o <program> to name the executable, else it will be called a.out
  -O 0 disables optimisations, giving simpler code
objdump <program> Lets you view information about a program
  -x to see the headers
  -d to disassemble
strace <program> Runs the program and shows systemcalls as they are made (usefull for debugging shellcode)
nc <host> <port> The netcat program. Useful for sending and listening to data going between ports. By default writes.
  -l to listen

ip addr See what ip address the virtual machine has

GDB

GDB is the go-to command line debugger. For some primitive GUI support, you can try running it inside Emacs (M-x gdb) or with a curses UI, gdb --tui.

gdb <program> Runs GDB on your program
  -x <gdbinit> lets you run a script when GDB is first loaded

Inside GDB, the following commmands are useful:

run [<args>] [< <input>] Run your program with optional args and input
  Can use backticks in the args to run an external command (such as ‘perl -e ‘print “A”x9001’‘)
set args [<args>] [< <input>] Specifies the args for the run command automatically
awatch <address> Sticks a read/write watch point whenever the memory at the address is accessed
b <break-point> Set a break point at a memory location (which can be a function name, e.g., b main or a de-referenced pointer, as b *0x123456)
c Continue
si Step instruction
x/32x $ebp Prints the memory as hexadecimal ints for the 32 bytes at the memory pointed to by the register $ebp
x/8i $eip Prints the memory as disassembled instructions the 8 bytes at the memory pointed to by the register $eip
p/x $ebp Print the value in the register $ebp

disas Shows the disassembly for wherever the instruction pointer is
list Shows where you are in the source code (if debugging data is on)
<!– **set disassembly-flavor intel** Gives you Intel-style disassembly –>
help The manual!

Radare

Radare is a *really* powerful dissembler framework but the documentation is non-existent. It is worth learning but not required for these labs. Here are a couple of commands which help in constructing shellcode.

rasm2 "nop;nop;nop" Gives you the bytecode for three `nop` instructions
rasm2 -f <file> -a x86.as -C Gives you the bytecode for assembler instructions in `<file>`, avoiding zeros and printing the result as a C-formatted string

Submission instructions

Download a copy of the text file checkpoints.md from

```
http://www.inf.ed.ac.uk/teaching/courses/sp/2016/labs/lab1/checkpoints.md
```

and edit it to insert your answers. Submit it with the command:

```
submit sp lab1 checkpoints.md
```

We will give some feedback about the submissions at the next lab session.

David Aspinall and Arthur Chan, February 2017. *Thanks to Joseph Hallett for some of the exercises.*