UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

**CS 458/658** **Computer Security and Privacy** **Spring 2014**
**Urs Hengartner**

# ASSIGNMENT 1

Blog Task signup due date: **Wednesday, May 14, 2014 3:00pm (no extension)**
Milestone due date: **Wednesday, May 21, 2014 3:00pm**
Assignment due date: **Monday, June 2, 2014 3:00 pm**

**Total marks:** 64
**Written Response Questions TA:** Aaron Atwater
**Programming Question TA:** Yihang (Frank) Song

Please use Piazza for all communication. Ask a private question if necessary. The TAs' office hours are also posted to Piazza.

## Blog Task

0. [0 marks, but -2 if you do not sign up by the due date] Sign up for a blog task timeslot by the due data above. The 48 hour late policy, as described in the course syllabus, does not apply to this signup due date. Look at the blog task in the Course Materials, Content section of the course website to learn how to sign up.

## Written Response Questions [24 marks]

Note: For written questions, please be sure to use complete, grammatically correct sentences. You will be marked on the presentation and clarity of your answers as well as the content.

1. In this question, you are asked to compare traditional voting to Internet voting.

   In traditional voting, voters go to the polling station, where officials ensure that a voter is eligible to vote and has not voted yet. If this vote registration step is successful, the voter will be given a paper ballot. The voter fills in the paper ballot in secret and puts the ballot into a locked ballot box that is in plain view of some officials. At the end of the day, the officials jointly open the ballot box, count the votes, and phone headquarters to submit the results. At a later stage, a signed report is sent to headquarters.

In Internet voting, voters use their home computer or smartphone to place their vote. Each eligible voter is physically mailed a letter with the URL of the voting website and a unique authorization code. The voter uses his/her computer/smartphone to go to the voting website and enter his/her vote and the authorization code. This information is transmitted to a centralized server, which ensures that the authorization code is valid and has not been used. Then the server aggregates the votes.

- (8 marks) Explain and discuss whether paper-based voting is susceptible to interception, interruption, modification, and fabrication attacks. If you believe the system is susceptible to a given attack, provide a specific, realistic attack scenario. If necessary, state any additional assumptions made about the system.
- (8 marks) Explain and discuss whether electronic voting voting is susceptible to interception, interruption, modification, and fabrication attacks. If you believe the system is susceptible to a given attack, provide a specific, realistic attack scenario. If necessary, state any additional assumptions made about the system.
- (2 marks) Based on your discussion, which scheme is more secure?

2. Alice uses her credit card to pay for purchases at a grocery store. Assume a traditional credit card, that is, a card with neither RFID capabilities nor a chip.

   - (4 marks) Briefly explain how the cashier is expected to both identify and authenticate Alice. Clearly separate the two steps.
   - (2 marks) Explain how this identification and authentication model breaks down for transactions across the Internet.

## Programming Response Questions [40 marks]

### Background

You are tasked with testing the security of a custom-developed *file submission application* for your organization. It is known that the application was *very poorly written*, and that in the past, this application had been exploited by some users with the malicious intent of *gaining root privileges*. There is some talk of the application having *four or more vulnerabilities*! As you are the only person in your organization to have a background in computer security, only you can *demonstrate how these vulnerabilities can be exploited* and *document/describe your exploits* so a fix can be made in the future.

## Application Description

The application is a very simple program to submit files. It is invoked in the following way:

- `submit <path to file> [message]` : this will copy the file from the current working directory into the submission directory, and append the string "message" to a file called `submit.log` in the user's home directory.

There may be other ways to invoke the program that you are unaware of. Luckily, you have been provided with the source code of the application, `submit.c`, for further analysis

The executable `submit` is *setuid root*, meaning that whenever `submit` is executed (even by a normal user), it will have the full privileges of *root* instead of the privileges of the normal user. Therefore, if a normal user can exploit a vulnerability in a setuid root target, he or she can cause the target to execute arbitrary code (such as shellcode) with the full permissions of root. If you are successful, running your exploit program will execute the setuid submit, which will perform some privileged operations, which will result in a shell with root privileges. You can verify that the resulting shell has root privileges by running the `whoami` command within that shell. The shell can be exited with `exit` command.

## Testing Environment

To help with your testing, you have been provided with a virtual *user-mode linux* (uml) environment where you can log in and test your exploits. These are located on one of the *ugster* machines. You will receive an email with your account credentials for your designated ugster machine.

Once you have logged into your ugster account, you can run `uml` to start your virtual linux environment. The following logins are useful to keep handy as reference:

- `user` (no password): main login for virtual environment
- `halt` (no password): halts the virtual environment, and returns you to the ugster prompt

The executable `submit` application has been installed to `/usr/local/bin` in the virtual environment, while `/usr/local/src` on the same environment contains `submit.c`. Conveniently, someone seems to have left some shellcode in `shellcode.h` in the same directory.

It is important to note all changes made to the virtual environment will be lost when you halt it. Thus it is important to remember to keep your working files in `/share` on the virtual environment, which maps to `~/uml/share` on the ugster environment.

**Rules for exploit execution**

- You have to submit four exploit programs to be considered for full credit. Two of your submitted exploit programs must exploit specific vulnerabilities. Namely, one must target a buffer overflow vulnerability, and another must target a format string vulnerability. Your other submitted exploit programs can address other vulnerabilities.
- Each vulnerability may be exploited only in a single exploit program. A single exploit program may exploit more than one vulnerability. If unsure whether two vulnerabilities are different, please contact the Programming Question TA.
- There is a specific execution procedure for your exploit programs ("*sploits*") when they are tested (i.e. graded) in the virtual environment:
  - Sploits will be run in a **pristine** virtual environment, i.e. you should not expect the presence of any additional files that are not already available
  - Execution will be from a clean `/share` directory on the virtual environment as follows: `./sploitX` (where X=1..4)
  - Sploits must not require any command line parameters
  - Sploits must not expect any user input
  - If your sploit requires additional files, it has to create them itself
- For marking, we will compile your exploit programs in the /share directory in a virtual machine in the following way: `gcc -Wall -ggdb sploitX.c -o sploitX`. You can assume that shellcode.h is available in the /share directory.
- Be polite. After ending up in a root shell, the user invoking your exploit program must still be able to exit the shell, log out, and terminate the virtual machine by logging in as user `halt`. Also, please do not run any cpu-intensive processes for a long time on the ugster machines (see below). None of the exploits should take more than about a minute to finish.
- Give feedback. In case your exploit program might not succeed instantly, keep the user informed of what is going on.

**Deliverables**

Each sploit is worth 10 marks, divided up as follows:

- 6 marks for a successfully running exploit that gains root
- 4 marks for the description of the identified vulnerability/vulnerabilities, saying how your exploit program exploits it/them, and describing how it/they could be repaired.

A total of four exploits must be submitted to be considered for full credit, including a *buffer overflow* sploit and a *format string* sploit. **Note:** sploit1.c and sploit2.c are due by the milestone date of Wednesday, May 21.

## What to hand in

It is very important that you follow the rules outlined in the Assignments section of the LEARN course site for submitting your assignment. Otherwise we may not be able to mark your assignment and you may lose partial or all marks.

By the **milestone deadline**, you are required to hand in:

**sploit1.c, sploit2.c:** Two completed exploit programs for the programming question. Note that we will build your sploit programs **on the uml virtual machine**

**a1-milestone.pdf:** A PDF file containing exploit descriptions for sploit1 and 2

**Note:** You will not be able to submit sploit1.c, sploit2.c or a1-milestone.pdf after the milestone date.

By the **a1 deadline**, you are required to hand in:

**sploit3.c, sploit4.c:** The remaining exploit programs for the programming question.

**a1.pdf:** A PDF file containing your answers for the written-response questions, and the exploit descriptions for sploit3 and 4. Do not put written answers pertaining to sploit1 and 2 into this file; they will be ignored.

We strongly encourage you to test your exploits in a clean `/share` directory (see earlier).

The 48 hour late policy, as described in the course syllabus, applies to the milestone due date and the final due date. It does not apply to the blog task signup due date.

## Useful Information For Programming Sploits

Most of the exploit programs do not require much code to be written. Nonetheless, we advise you to start early since you will likely have to read additional information to acquire the necessary knowledge for finding and exploiting a vulnerability. Namely, we suggest that you take a closer look at the following items:

- Module 2
- Smashing the Stack for Fun and Profit (http://insecure.org/stf/smashstack.html)
- Format String Attacks (http://www.thenewsh.com/ newsham/format-string-attacks.pdf)
- Intel x86 Function-call Conventions - Assembly View (http://www.unixwiz.net/techtips/win32-callconv-asm.html) The article does not explicitly mention that %esp is set to %ebp at the end of a function (before restoring the old value of %ebp). As hinted at by the article, keep in mind that gcc sometimes allocates space on the stack in addition to what you would expect just from looking at the source code (e.g., for storing register values). You might have to consider this additional space in your design of an exploit. You can learn about the size of this space with the help of gdb.
- The manpages for passwd (man 5 passwd), execve (man 2 execve), su (man su), mkdir (man mkdir), and chdir (man chdir). In general, if you don't understand a particular function used by the submit program, consult its manpage.
- Environment variables (e.g., http://en.wikipedia.org/wiki/Environment_variable).

Note that in the virtual machine you cannot create files that are owned by root in the `/share` directory. Similarly, you cannot run chown on files in this directory. (Think about why these limitations exist.)

### GDB

The gdb debugger will be useful for writing some of the exploit programs. It is available in the virtual machine. In case you have never used gdb, you are encouraged to look at a tutorial (e.g.,http://www.unknownroad.com/rtfm/gdbtut/).

Assuming your exploit program invokes the `submit` application using the `execve()` (or a similar) function, the following statements will allow you to debug the `submit` application:

1. `gdb sploitX` (X=1..4)
2. `catch exec` (This will make the debugger stop as soon as the `execve()` function is reached)
3. `run` (Run the exploit program)
4. `symbol-file /usr/local/bin/submit` (We are now in the submit application, so we need to load its symbol table)
5. `break main` (Set a breakpoint in the submit application)
6. `cont` (Run to breakpoint)

You can store commands 2-6 in a file and use the "`source`" command to execute them. Some other useful gdb commands are:

- "`info frame`" displays information about the current stackframe. Namely, "saved ip" gives you the current return address, as stored on the stack. Under saved registers, eip tells you where on the stack the return address is stored.
- "`info reg esp`" gives you the current value of the stack pointer.
- "`x <address>`" can be used to examine a memory location.
- "`print <variable>`" and "`print &<variable>`" will give you the value and address of a variable, respectively.
- See one of the various gdb cheat sheets (e.g., this one) for the various formatting options for the print and x command and for other commands.

## The Ugster Course Computing Environment

In order to responsibly let students learn about security flaws that can be exploited in order to become "root", we have set up a virtual "user-mode linux" (uml) environment where you can log in and mount your attacks. The gcc version for this environment is the same as described in the article "Smashing the Stack for Fun and Profit"; we have also disabled the stack randomization feature of the 2.6 Linux kernel so as to make your life easier. (But if you'd like an extra challenge, ask us how to turn it back on!)

To access this system, you will need to use ssh to log into your account on one of the `ugster` environment: `ugsterXX.student.cs.uwaterloo.ca`. There are a number of ugster machines, and each student will have an account for one of these machines. You will get an e-mail with your password and telling you which ugster you are to use. If you do not receive a password please check your spam folder. When logged into your ugster account, you can run "`uml`" to start the user-mode linux to boot up a virtual machine.

The gcc compiler installed in the uml environment is very old and does not fully implement the ANSI C99 standard. You must declare variables at the beginning of a function, before any other code. You cannot use single-line comments ("//").

Any changes that you make in the uml environment are lost when you exit (or upon a crash of user-mode linux). **Lost Forever**. Anything you want to keep must be put in `/share` in the virtual machine. This directory maps to `~/uml/share` on the ugster machines, which is how you can copy files in and out of the virtual machine. It is wisest to ssh twice into ugster. In one shell, start user-mode linux, and compile and execute your exploits. In the other account, log into ugster and edit your files directly in `~/uml/share/` so as to ensure you do not lose any work. The ugster machines are not backed up. You should copy all your work over to your student.cs account regularly.

When you want to exit the virtual machine, use exit. Then at the login prompt, login as user "halt" and no password to halt the machine.

Any questions about your ugster environment should be asked in Piazza.