UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Spring 2023**
Diogo Barradas
Adithya Vadapalli

ASSIGNMENT 1

Blog Task signup due date: **Wed, May 17, 2023 at *3:00 pm (no extension, no grace period)***
Milestone due date: **Wed, May 24, 2023 at *3:00 pm***
Assignment due date: **Wed, Jun 7, 2023 at *3:00 pm***

Total Marks: 70

Blog Task TA: Shreya Arun Naik
Written Response TA: Ru Ji **Office hours:** Fridays 1:00pm–2:00pm, DC 3332 (and BBB)
Programming TA: Adrian Cruzat La Rosa **Office hours:** Thursdays 10:00am–11:00am, BBB

Instructors:
Diogo Barradas **Office hours:** Tuesdays 10:00am–11:00am, DC 2631
Adithya vadapalli **Office hours:** By appointment only

Please use Piazza for all communication. Ask a private question if necessary.

Blog Task

1. Sign up for a blog task timeslot by the due date above. As described in the course syllabus, the 48-hour late policy does not apply to this signup due date. Look at the blog task description in the *Content* section of the course LEARN site to read about the blog task and to learn how to sign up.

   Please visit https://crysp.uwaterloo.ca/courses/cs458/infodist/blogtask.php to sign up.

   The blog task signup is worth 0 marks for Assignment 1, but if you do not complete it by the blog task signup due date above, you will be unable to earn the blog task marks associated with your individual blog post.

Note: Please ensure that written questions are answered using complete and grammatically correct sentences. You will be marked on the presentation and clarity of your answers as well as on the correctness of your answers.

Written Response Questions [30 marks]

In the movie *Everything Everywhere All at Once*, people from Alphaverse discovered the secret of parallel universes. Every choice a person makes in life will create a new alternative universe. There exists a network of parallel universes all at once. People from Alphaverse also developed a technology named "verse jumping". The way that verse jumping works is as follows: (a) find and perform a low probability action as a jumping pad, then (b) press a button on a path scanning device. After verse jumping, a person can bring back the abilities they possessed in the target universe back to the original universe. For instance, if the verse jumper happens to be a kung fu master in a target universe, they will be able to fight when they come back to their original universe. The protagonist's (Evelyn) mission is to fight against Jobu, who gained the ability to verse jump and manipulate matter at will.

In the questions below, we consider each parallel universe as assets to be protected, and we consider anyone who tries to verse jump into other people's universe as an attacker.

Based on the above description, please answer the following questions:

1. (6 marks) The path scanner stores all parallel universe information of a person. In order to prevent others from peeking into your parallel universes, it is better to keep it secure.
   (a) Can you point out two potential problems of using a traditional password on the path scanner (one focusing security, another focusing usability)?
   (b) Can you point out two different ways to better secure the path scanner (even if one of them still is a password-based approach)? Briefly explain (a) why these alternative solutions are generally better, and also (b) pinpoint potential problems with them.
2. (10 marks) Evelyn is concerned about the tense relationship within her family and the preparation for an audit from IRS (Internal Revenue Service). Their business – a laudromat – is not taken good care of. Customers complain about clothes and personal belongings getting stolen. In order to keep the business going, Evelyn needs to figure out some defence mechanisms. For each of the five classes of methods to defend against thieves (prevent, deter, deflect, detect and recover), give a clear example and briefly explain why this is an example of that class.
3. (6 marks) After the movie's happy ending, Jobu decided to use her ability to perform research on predicting future events based on her knowledge of every parallel universe. She stores the results of her research locally on her computer, and saves a backup copy on a cloud service. Jobu's research results are an attractive target for malicious parties. For each of the possible attack situations described below, please (a) identify the name of the attack approach, and briefly describe the attack process; and (b) identify which one of the security properties (Confidentiality, Integrity and/or Availability) will be compromised.
   (a) Jobu finds out that her research results were leaked. Jobu remembers to have received (and acted upon) an email reminder to update her cloud account password before uploading the latest batch of data.
   (b) Jobu finds out that she is oftentimes unable to log into her cloud service's account, as the website does not seem to respond. However, the she is able to access other Internet pages just fine.

(c) Jobu finds out that her data has been tampered with. Upon further investigation of her computer's system logs, she finds out that there was a user named "surprise" who modified the data several times during the last month.

4. (8 marks) Identify each of the following pieces of malware as a worm, trojan, ransomware, and/or logic bomb. For each specific malware not mentioned in the lectures, please refer to the links below for additional information. For answering the question, give a brief description of (a) **how it spreads** or **how a computer becomes infected**, and (b) **the resulting effect**. (Each piece of malware could have multiple classifications.)

   - Info on Sobig
   - Info on CIH
   - Info on BlackCat

   (a) Stuxnet

   (b) Sobig

   (c) CIH

   (d) BlackCat

Programming Question [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.* 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.

You have been provided with the source code and its corresponding executable binary for this application. There is some talk of the application having *three or more vulnerabilities*! In addition, you have also acquired a different *modified version of the submit application* which you suspect has been altered to include a backdoor (*one additional vulnerability*). Unfortunately you don't have the source code for this version, but you know that it is *very similar to the original version.*

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.
- `submit -s` : show the files you have submitted to the submission directory.
- `submit -v` : show the version of the `submit` application.
- `submit -h` : show a usage message.

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 original version of the application is named `submit`, while the modified (backdoored) version is named `submitV2`. The modified version has the same vulnerabilities as the original application, plus one additional vulnerability. The provided source code `submit.c` corresponds to the original version, while the source code for the modified version is *not* provided. In order to be sneaky, the source code for the modified version differs from `submit.c` by a single line, but it is up to you to find this difference.

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, they 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 can retrieve your account credentials from the Infodist system: https://crysp.uwaterloo.ca/courses/cs458/infodist/.

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` and `submitV2` applications have been installed to `/usr/local/bin` in the virtual environment, while `/usr/local/src` on the same environment contains the source code `submit.c`. Conveniently, someone seems to have left some shellcode in the file `shellcode.h` in the same directory.

It is important to note that **all changes made to the virtual uml 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.

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.)

The root password in the virtual environment is a long random string, so there is no use in attempting a brute-force attack on the password. You will need to exploit vulnerabilities in the application.

Rules for exploit execution

- You must submit a total of four (3+1) exploit programs to be considered for full credit.
  - You must submit *three* exploit programs that target the original submit application (`submit`). Two of these submitted exploit programs must exploit specific vulnerabilities. Namely, one must target a *buffer overflow* vulnerability that overwrites a saved return address on the stack, and another must target a *format string* vulnerability. Your other submitted exploit program can address some other vulnerability.
  - You also must submit *one additional* exploit program that targets the modified submit application (`submitV2`). This submitted exploit program must exploit the vulnerability added in this modified application and not any of the original vulnerabilities. Therefore, your exploit must not work on the original submit application.
- Running each exploit program should result in a shell with root privileges.
- Each vulnerability can be exploited only in a single exploit program, but a single exploit program can exploit more than one vulnerability. You can exploit the same *class* of vulnerability (e.g., buffer

overflow, format string, etc.) in multiple exploit programs, but they must exploit different sections of the code. You may also exploit the same section of code in multiple exploit programs as long as they each use a *different* class of vulnerability. If unsure whether two vulnerabilities are different, please ask a *private* question on Piazza.

- 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 compiled and run in a **pristine** virtual environment; i.e., you should not expect the presence of any additional files that are not already available. The virtual environment is restarted between each exploit test.
  - The three exploits for the original application must be named `sploitX` (where X=1..3), and the exploit for the modified application must be named `sploit4`. **Important**: Even if you submit fewer than three exploit programs for the original application, the exploit program for the modified application must still be named `sploit4`.
  - Your exploit programs will be compiled from the `/share` directory of the virtual environment in the following way:
    `cd /share && gcc -Wall -ggdb sploitX.c -o /home/user/sploitX`
    You can assume that `shellcode.h` is available in the `/share` directory.
  - Execution will be from a clean home directory (`/home/user`) on the virtual environment as follows: `./sploitX` (where X=1..4). Make sure to run your exploits in this same manner when developing them since an exploit that works in one directory is not guaranteed to work when running from another.
  - Sploits must not require any command line parameters.
  - Sploits must not expect any user input.
  - Sploits must not take longer than 60 seconds to complete.
  - If your sploit requires additional files, it has to create them itself.
- 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 are designed to take more than a minute to finish.

The goal is to end up in a shell that has root privileges. So you should be able to run your exploit program, and without any user/keyboard input, end up in a root shell. If you as the user then type in `whoami`, the shell should output `root`. Your exploit code itself doesn't need to run whoami, but that's an easy way for you to check if the shell you started has root privileges.

For example, testing your exploit code might look something like the following:

```
user@cs458-uml:/share$ gcc -Wall -ggdb sploit1.c -o /home/user/sploit1
user@cs458-uml:/share$ cd
user@cs458-uml:~$ ./sploit1
sh# whoami
root
sh# exit
user@cs458-uml:~$
```

For questions about the assignment, ugsters, virtual environment, Infodist, etc, please post a question to Piazza. General questions should be posted publicly, but **do not** ask public questions containing (partial) solutions on Piazza. Questions that describe the locations of vulnerabilities, or code to exploit these vulnerabilities, should be posted *privately*. If you are unsure, you can always post your question privately and ask a TA to make your question public.

Warmup

We have provided an additional setuid program, called `warmup`, along with its source code `warmup.c`, in the same directories as `submit` and `submit.c`. The `warmup` program contains a very straightforward buffer overflow that you can try to exploit as a warmup. You can also read a walkthrough that showcases the steps to perform the exploit in the `warmup.pdf` file you can find in the Assignment 1 folder on LEARN. This warmup is only for your own practice; you **do not submit** your sploit for this program.

Deliverables

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

- 6 marks for a successfully running exploit that gains a shell with root privileges
  - No part-marks will be given for incomplete exploit programs.
- 4 marks for a brief/concise description of:
  1. The identified vulnerability/vulnerabilities.
  2. How your exploit program exploits it/them.
  3. How it/they could be fixed. Valid fixes must not reduce or negatively affect the functionality of the program and they must fix the vulnerability directly in the code. For example, enabling ASLR, stack canaries, etc. are **not** acceptable answers.

A total of four exploits (as described above) must be submitted to be considered for full credit. Marks will be docked if you submit no *buffer overflow* sploit that overwrites a saved return address on the stack or no *format string* sploit *for the original application*. **Note:** sploit1.c and sploit2.c are due by the milestone due date given above. You can but *do not need to* submit the buffer overflow sploit or format string sploit by the milestone due date.

What to hand in

All assignment submission takes place on the `linux.student.cs` machines (not ugster or the uml virtual environments), using the `submit` utility. (The one on the `linux.student.cs` machines for actually submitting assignment files, not the program in the uml environment you are exploiting in this assignment!) In particular, log in to the Linux student environment (`linux.student.cs.uwaterloo.ca`), go to the directory that contains your solution, and submit using the following command: `submit cs458 1 .` (dot included). CS 658 students should also use this command and ignore the warning message. If you are submitting late (in the 48-hour grace period), you will require the "`-t`" (tardy) option to `submit`: `submit cs458 1 -t .` (including the dot). You should verify the files you submitted are the ones you intended with the "`-p`" (print) option to `submit`.

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

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

a1-milestone.pdf: A PDF file containing exploit descriptions for sploit{1,2} (including fixes, as explained above).

Note: You will **not** be able to submit `sploit1.c`, `sploit2.c` or `a1-milestone.pdf` after the milestone due date (plus 48 hours).

By the **assignment due date**, you are required to hand in:

sploit3.c, sploit4.c: The remaining exploit program for the original application (sploit3), plus the additional exploit program for the modified application (sploit4).

a1-responses.pdf: A PDF file containing your answers for the written-response questions, and the exploit descriptions for sploit{3,4} (including fixes, as explained above). Do not put written answers pertaining to sploit{1,2} into this file; they will be ignored.

The 48-hour no-penalty late policy, as described in the course syllabus, applies to the milestone due date and the assignment due date. It does not apply to the blog task signup due date.

The instructors and TAs will not answer questions made on Piazza after the assignment due date (including during the 48-hour extension period).

Useful Information For Programming Sploits

The first step in writing your exploit programs will be to identify vulnerabilities in the `submit.c` source code. 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
  (https://insecure.org/stf/smashstack.html)
  Note: The original article above has a few errors; the following link claims to have fixed them
  (https://www.eecs.umich.edu/courses/eecs588/static/stack_smashing.pdf)
- Exploiting Format String Vulnerabilities (v1.2)
  (http://julianor.tripod.com/bc/formatstring-1.2.pdf, Sections 1-3 only)
- The manpages for execve (man 2 execve), system (man 3 system), passwd (man 5 passwd), su (man su), ln (man ln), symlink (man symlink), chdir (man chdir), mkdir (man mkdir), and objdump (man objdump)
- Environment variables
  (e.g., https://en.wikipedia.org/wiki/Environment_variable)
- Simplified objdump
  (https://stackoverflow.com/questions/8541906/objdump-displaying-without-offsets)

You are allowed to use code from any of the previous webpages as starting points for your sploits, and do not need to cite them.

GDB

The gdb debugger will be useful for writing *some* of the exploit programs. It is available in the virtual machine. For the buffer overflow exploit in particular, using gdb will allow you to figure out the exact address of your shellcode, so using NOPs will **not** be necessary.

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 eip" 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., https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf) for the various formatting options for the print and x command, and for other commands.

Note that `submit` will not run any program or command with root privileges while you are debugging it with gdb. (Think about why this limitation exists.)

The Ugster Course Computing Environment

In order to responsibly let students learn about security flaws that can be exploited, 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` environments: `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 can retrieve your account credentials from the Infodist system.

The ugster machines are located behind the university's firewall. While on campus, you should be able to ssh directly to your ugster machine. When off campus, you have the option of using the university's VPN (see these instructions), or you can first ssh into `linux.student.cs.uwaterloo.ca` and then ssh into your ugster machine from there.

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 may be very old and may not fully implement the ANSI C99 standard. You might need to declare variables at the beginning of a function, before any other code. You may also be unable to use single-line comments ("//"). If you encounter compile errors, check for these cases before asking on Piazza.

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 can be helpful to ssh twice into ugster. In one window, log into the ugster and start user-mode linux, and compile and execute your exploits. In the other window, log into the 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 on Piazza.

Miscellaneous

Running your exploits while using ssh in bash on the Windows 10 Subsystem for Linux (WSL) has been known to cause problems. You are free to use ssh in bash on WSL if it works, but if the WSL freezes or crashes, please try PuTTY or a Linux VM instead.

There are bugs when using `vi` to edit files in the `/share` directory in the virtual environment. It is recommended to use `nano` inside the virtual environment, or even better, use `vim` on the ugster machine in a separate ssh session.