UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Fall 2018**
Ian Goldberg
Florian Kerschbaum

ASSIGNMENT 1

Blog Task signup due date: **Fri, Sept 14, 2018 at *4:00 pm (no extension)***
Milestone due date: **Fri, Sept 21, 2018 at *4:00 pm***
Assignment due date: **Wed, Oct 3, 2018 at *4:00 pm***

Total Marks: 80
Blog Task TA: Pierfrancesco Cervellini
Written Response TA: Pierfrancesco Cervellini
Programming TA: Steven Engler

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

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

Written Response Questions [30 marks]

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.

Note: Further, do not copy text directly from any sources, and make sure to cite any sources you do use.

1. (8 marks): During the investigation of a series of robberies in early 2018, the FBI asked Google to provide a list of all users of its services that were in the area of two of the crimes, within half hour of their occurrence.

   Examples like this help us reflect on the consequences of our decisions to liberally share data online and with companies. We will ask some hypothetical questions based on this real-world event. Reading the full details of the case in not required to answer these questions, but available at https://www.forbes.com/sites/thomasbrewster/2018/08/15/to-catch-a-robber-the-fbi-attempted-an-unprecendeted-grab-for-google-location-data/.

For each of the following, please:

- identify the scenario as a compromise of Confidentiality, Integrity, Availability, and/or Privacy,
- briefly explain your choice of compromise.

(Hint: You are only required to give one compromise and corresponding explanation per scenario.)

(a) (2 marks) Google agrees with the FBI request and provides account activity, location history, full name, and address for each user at the specified time and place.

(b) (2 marks) Appalled by Google's actions, a group of citizens sue Google. A sympathetic judge shuts down its feed to the FBI.

(c) (2 marks) Before losing access to Google's data, the FBI notices that quite a few users were present at the right time in the vicinity of the robbery locations. Too many for them all to be the thieves. To shorten their list, they decide to start intercepting their calls.

(d) (2 marks) One of the robbers turns out to be an elite hacker. Upon hearing of Google's collaboration with the FBI, she hacks their database and removes her team's data from it.

2. (8 marks): The FBI is not holding back in its strategies to apprehend the thieves.

   For each of the following, please:

   - answer whether the threat represented in each of the following scenarios is one of interception, modification, interruption, and/or fabrication
   - give a brief explanation for each of your answers.

   (a) (2 marks) The hacker's activity in their database was actually noticed by Google who quickly acted to blacklist the offending IP and inform the FBI.

   (b) (2 marks) With knowledge of what users' data the intruder was attempting to change, the FBI obtains a court order to listen into all of these users' activity.

   (c) (2 marks) The FBI catches wind of a buyer interested in the stolen merchandise and decides to impersonate him. They build a fictional profile to show the thieves and convince them he is legitimate.

   (d) (2 marks) To make it seem even more realistic, the FBI carries out a Man-In-The-Middle attack by intercepting all the incoming traffic by the true buyer and changing the details of the communications to lure the thieves into a trap.

3. (6 marks): For this next part, put yourself in the shoes of the crew's hacker. How would you use these mitigations strategies to shield your team from being captured, and throw the FBI off your trail?

   For each of the following, please:

   - explain how you could use the defense to your advantage
   - try to provide context that fits the narrative of the assignment.

   Example: Preventing.
   Travel to a country where the FBI has no jurisdiction, and there is no extradition to the US.

   (a) (2 marks) Deflecting.

   (b) (2 marks) Detecting.

   (c) (2 marks) Recovering.

4. (8 marks): Identify each of the following pieces of malware as a worm, trojan, ransomware, and/or logic bomb. Then, briefly and concisely, give a description of how it spread, or how a computer is infected, and the resulting effect. Keep in mind that each piece of malware could have multiple classifications.

   Example: CryptoLocker.
   A ransomware trojan targeting MS Windows machines, and disseminated through malicious email attachments and the ZeuS Botnet. Once activated it uses public key cryptography to encrypt files, forcing the victim to pay a ransom for the release of the private key to decrypt their files.

   (a) (2 marks) Kovter.

   (b) (2 marks) ILOVEYOU.

   (c) (2 marks) Mirai.

   (d) (2 marks) NotPetya.

Programming Question [50 marks]

Background

You are tasked with testing the security of a custom-developed *backup 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 *four or more vulnerabilities*! In addition, you have also acquired a different *modified version of the backup 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's *very similar to the original version.*

Application Description

The application is a very simple program with the purpose of backing up and restoring files. There are at least three ways to invoke it:

- `backup backup foo` : this will copy file *foo* from the current working directory into the backup space.
- `backup restore foo` : this will copy file *foo* from the backup space into the current working directory.
- `backup ls` : this will list the files stored in the backup space.

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, `backup.c`, for further analysis.

The original version of the application is named `backup`, while the modified (backdoored) version is named `backupV2`. The modified version has the same vulnerabilities as the original application, plus one additional vulnerability. The provided source code `backup.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 `backup.c` by a single line, but it is up to you to find this difference.

The executable `backup` is *setuid root*, meaning that whenever `backup` 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 backup, 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 `backup` and `backupV2` applications have been installed to `/usr/local/bin` in the virtual environment, while `/usr/local/src` on the same environment contains the source code `backup.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 five (4+1) exploit programs to be considered for full credit.
  - You must submit *four* exploit programs that target the original backup application (`backup`). 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 programs can address other vulnerabilities.
  - You also must submit *one additional* exploit program that targets the modified backup application (`backupV2`). 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 backup application.
- Running each exploit program should result in a shell owned by root.
- 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 (ex: 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 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 four exploits for the original application must be named `sploitX` (where X=1..4), and the exploit for the modified application must be named `sploit5`. **Important**: Even if you submit fewer than four exploit programs for the original application, the exploit program for the modified application must still be named `sploit5`.
  - Execution will be from a clean home directory (`/home/user`) on the virtual environment as follows: `./sploitX` (where X=1..5)
  - 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
- For marking, we will compile your exploit programs in the `/share` directory in a virtual machine 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.
- 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.

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:~$ ./sploit1
sh# whoami
root
sh#
```

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 a TA can make your question public if they believe it to be useful for the class.

Deliverables

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

- 6 marks for a successfully running exploit that gains a shell owned by root
- 4 marks for the description of:
  - the identified vulnerability/vulnerabilities
  - how your exploit program exploits it/them
  - how it/they could be fixed

A total of five 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 `student.cs` machines (not ugster or the virtual environments), using the `submit` utility. 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.

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, sploit5.c: The two remaining exploit programs for the original application (sploit3 and sploit4), plus the additional exploit program for the modified application (sploit5).

a1.pdf: A PDF file containing your answers for the written-response questions, and the exploit descriptions for sploit{3,4,5} (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.

For the milestone, the TAs will continue to answer questions on Piazza after the milestone due date (and through the 48 hour extension period). However, they will no longer answer questions after the assignment due date (and *not* through 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 `backup.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)
- Exploiting Format String Vulnerabilities (v1.2) (http://julianor.tripod.com/bc/formatstring-1.2.pdf, Sections 1-3 only)
- The manpages for passwd (man 5 passwd), execve (man 2 execve), su (man su), ln (man ln), mkdir (man mkdir), chdir (man chdir), and objdump (man objdump)
- SSH public key authentication (e.g., https://cs.uwaterloo.ca/cscf/howto/ssh/public_key/, Ignore the PuTTY part for this assignment)
- 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. 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 `backup` application using the `execve()` (or a similar) function, the following statements will allow you to debug the `backup` application:

1. `gdb sploitX` (X=1..5)
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/backup` (We are now in the `backup` application, so we need to load its symbol table)
5. `break main` (Set a breakpoint in the `backup` 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 `backup` 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 shell, log into the ugster and start user-mode linux, and compile and execute your exploits. In the other account, 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.