UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Fall 2021**

Urs Hengartner
Adithya Vadapalli

ASSIGNMENT 1

Milestone due date: **Friday, October 1st, 2021 3:00pm (no extensions)**
Assignment due date: **Friday, October 8th, 2021 3:00 pm**

Total marks: 69
Written Response Questions TA: Setareh Ghorshi
Programming Question TA's: Andre Kassis & Adam Campbell

Please use Piazza for all communication. Ask a private question if necessary. The TAs' office hours are Thursdays, 9-10am in the A1 Programming and Written Office Hour channels of the CS 458/CS 658 - Fall 21 Microsoft Team (see TA and Professor Office Hours in Learn).

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

Written Response Questions [29 marks]

Question 1 [Total: 19 marks]

A silent auction is a popular type of auctioning especially for nonprofit organizations. One way that a silent auction can be held is by asking the participants to fill registration forms at the beginning of the event with their personal information, such as contact information for later contact if they were the winner, and then assigning each bidder an id number. The bidders can then visit the items and anonymously write their bid for each item on a specific bid sheet along with their id numbers. The winner will later be notified using the contact information and will be provided with a payment and shipping method. This type of auction allows for more privacy since some people might not feel comfortable with others knowing about their bidding amount or personal information.

- (8 marks) There could be various intentions for attacking an auction. Do you think the silent auction is susceptible to interception, interruption, modification, or fabrication attacks? If so, provide a specific, realistic attack scenario for each category of attack you believe is possible. If not, provide an explanation for why

you believe a specific attack is not possible. If necessary, state any additional assumptions made about the system.

- (8 marks) The silent auction can also be held online through a website that allows bidders to sign up, view the items, and bid on them during a specific time using their ids without their identity being revealed to others on the website. The website is connected to a server keeping all the data about the bidders and auctions in a database. The winner will be notified through an email, including a payment link on the auction website and instructions for receiving their item. Do you believe that this process is susceptible to interception, interruption, modification, or fabrication attacks? If so, provide a specific, realistic attack scenario for each category of attack you believe is possible. If necessary, state any additional assumptions made about the system.
- (3 marks) Assume an online silent auction is being held as described before. In each of the below cases, which of the confidentiality, privacy, availability, or integrity is compromised? (Can be more then one, but only one correct answer is sufficient.) Justify your answer briefly.
  (a) An attacker gains access to a bidder's contact information through an interception attack and sends them an email claiming that they have won the auction. This email also includes a link to a fake payment page, identical to the actual website, which then sends the input data to the attacker.
  (b) The attacker accesses the database used by the auction website using malware and erases all the data.
  (c) The attacker publishes the contact information of users on the internet after accessing them through the previously described attack.

Question 2 [Total: 10 marks]

COVID-19 vaccine cards are nowadays being used in order to relax rules such as mandatory quarantine for travelers who enter Canada if they are vaccinated. However, this has provided an opportunity for scammers to create and sell fake vaccination documents to unvaccinated people. For each of the five classes of methods to defend against faking vaccine cards and passports (prevent, deter, deflect, detect, and recover), give a clear example and briefly explain (one sentence) why this is an example of that class.

Programming Question [40 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*. 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 with the purpose of backing up and restoring files. There are at least two ways to invoke it:

- `backup backup foo password` : this will copy file *foo* from the current working directory into the backup space if *password* is correct.
- `backup restore foo password` : this will copy file *foo* from the backup space into the current working directory if *password* is correct.

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 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` application has 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 `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

The `backup.c` source code contains vulnerabilities that belong to two classes:

- Memory execution vulnerabilities that corrupt the program memory (i.e., stack or heap) in order to enable a malicious attacker to **inject** and then execute arbitrary code. Note that in this case the attacker provides the code to be executed by placing it on the stack (heap). Specifically, the code contains two vulnerabilities belonging to this class — a format string vulnerability and a buffer overflow vulnerability.
- Non-memory execution vulnerabilities that *do not* corrupt the program memory in order to execute arbitrary code. Examples that have been presented in class include TOCTTOU vulnerabilities, integer overflows to corrupt variables and similar techniques.

Task: You must submit a total of **three** exploit programs such that **one** exploit addresses a memory vulnerability (class A), and **two** exploits address non-memory vulnerabilities (class B).

- **IMPORTANT** - A class A exploit may involve intermediate steps that require non-memory execution vulnerabilities. However, for a program to be considered a valid class A exploit, it must obtain a root shell by executing code from the stack (heap). Class B exploits on the other hand must not exploit any memory execution vulnerabilities at all!
- Please note that the two classes are exclusive. That is, an exploit from Class B may not replace one from Class A, and vice versa.
- You *may* use multiple vulnerabilities of the **same class** in one exploit.
- You may exploit the same section of code in multiple exploit programs as long as they each use *different* vulnerabilities. If you are unsure whether two vulnerabilities are different, please ask a private question on Piazza.
- Running each exploit program should result in a shell with root privileges.
- A single exploit program can exploit more than one vulnerability. At least one of the vulnerabilities exploited by an exploit program must be unique, that is, none of your other exploit programs can exploit the same 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.
  - Execution will be from a clean `/home/user` directory on the virtual environment as follows: `./sploitX` (where X=1..4).
    You can assume that `shellcode.h` is available in the `/share` directory.
  - 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: `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 are designed to take more than 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:/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.

Grading

For the final submission a total of **three** exploits must be submitted such that:

- **One** exploit that targets a memory vulnerability (Class A) — 20 marks total.
- **Two** exploits, each of which target a non-memory vulnerability (Class B) — 10 marks each, 20 marks total.

The marks for any given exploit are assigned as follows:

- 60% for a successfully running exploit that gains a shell with root privileges
  - No part-marks will be given for incomplete exploit programs.
- 40% for the exploit description. Consider answering the following questions in your description:
  - What class does the exploit program address (class A / class B)?

- What is the vulnerability/vulnerabilities (buffer overflow, TOCTTOU, etc.) that your exploit program addresses?
- How does your exploit program exploit it/them (with a step-by-step description, explaining the rationale for each step)?
- How can the vulnerability/vulnerabilities be fixed (by specific changes to the vulnerable program itself, not by system-wide changes like adding ASLR, stack canaries, NX bits, etc.)?

What to hand in

All assignment submissions take place on the `linux.student.cs` machines (not ugster or the virtual environments), using the `submit` utility. Log in to the linux student environment, go to the directory that contains your solution, and submit using the following command:

`submit cs458 1 .` (including the dot at the end)

CS 658 students should also use this command and ignore the warning message. Please consult the course homepage for help with the `submit` utility and accessing the linux student environment.

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

sploit1.c A single completed exploit program 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 description for sploit1 (including repairs, as explained above)

Note: You will not be able to submit sploit1.c and a1-milestone.pdf after the milestone due date.

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

sploit2.c, sploit3.c: The two 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 sploit2 and 3 (including repairs, as explained above). Do not put written answers pertaining to sploit1 into this file; they will be ignored.

Note that feedback will not be offered on your milestone submission. You may modify and *resubmit* your milestone exploit for the assignment deadline. However, your resubmitted exploit

will have to be renamed (from sploit1.c to sploit2.c). Additionally, if you do choose to resubmit, then you will not be receiving any marks on the initial submission (sploit1.c), and only two sploits will count toward your final grade (sploit2.c and sploit3.c). If you resubmit an exploit, you must mention the resubmission in a1.pdf.

In summary, the rules are as follows:

- By the milestone date, you should submit exactly one exploit—sploit1.c—along with the respective pdf file.
- By the assignment deadline, you have to submit sploit2.c and sploit3.c, where sploit2.c can be a resubmission of sploit1.c. In this case however, you will lose all the points you have been awarded (if any) for sploit1.c, submitted prior to the milestone deadline.
- Exactly one of the three submitted exploits should be a class A exploit, and the remaining two should be class B exploits. So, you can submit a class A exploit at the milestone deadline and two class B exploits at the actual deadline, or you can submit a class B exploit at the actual deadline and one class A and one class B exploit at the actual deadline.

The 48 hour late policy, as described in the course syllabus, applies to the assignment due date, but not to the milestone due date. The TAs will not answer questions made on Piazza after the assignment due date (or during the 48 hour extension period).

Your PDF files must contain, at the top of the first page your name, WatIAM user ID and student number. **We will not be accepting hand written solutions**. Be sure to "embed all fonts" into your PDF file so that the grader can view the file as intended. Some students' files were unreadable in the past; if we can't read it, we can't mark it. (Note that renaming the extension of a file to **.pdf** does not make it a PDF file.)

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)
  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 passwd (man 5 passwd), execve (man 2 execve), su (man su), system (man 3 system), ln (man ln), symlink (man symlink), mkdir (man mkdir), chdir (man chdir) and **popen (man popen)**
- SSH public key authentication (e.g., https://cs.uwaterloo.ca/cscf//howto/ssh/public_key/index.shtml, ignore the PuTTY part for this assignment)

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 `backup` application using the `execve()` (or a similar) function, the following statements will allow you to debug the `backup` application:

`gdb sploitX` (X=1..4)

`catch exec` (This will make the debugger stop as soon as the `execve()` function is reached)

`run` (Run the exploit program)

`symbol-file /usr/local/bin/backup` (We are now in the `backup` application, so we need to load its symbol table)

`break main` (Set a breakpoint in the `backup` application)

`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.)

Advice from the TAs — IMPORTANT

When debugging with GDB make sure to unset the environment variables `LINES` and `COLUMNS` set by GDB. Not doing so may result in the stack starting at different addresses with and without GDB, causing your exploits to not work outside of the debugger. The following statements will allow you to unset the GDB environment variables:

`gdb /path/to/program` (To start GDB with the specified program)

```
unset env LINES
```

```
unset env COLUMNS
```

`show env` (To verify that `LINES` and `COLUMNS` are not defined)

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!)

There are a number of `ugster` machines. Each student will have an account for one of these machines. You can retrieve your account credentials and what ugster machine you are assigned, from the Infodist system. Please read the limits placed on student accounts listed on the Infodist webpage. Any further questions about your ugster environment should be asked on Piazza. Please prefix your post title with "Sysadmin TA:".

Use `ssh` to log into your account to the appropriate ugster environment (replace XX with your ugster machine): `ugsterXX.student.cs.uwaterloo.ca`. 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. (ssh -J `linux.student.cs.uwaterloo.ca ugsterXX.student.cs.uwaterloo.ca` is a shortcut.)

- 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.

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.

Questions

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 (with your consent) make your question public if they believe it to be useful for the class.