UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Fall 2020**
ASSIGNMENT 1

Milestone due date: **Monday, September 21st, 2020 3:00 pm**
Assignment due date: **Friday, October 9th, 2020 3:00 pm**

Total marks: 66
Written Response Questions TA: Sina Faraji
Programming Question TA: Nils Lukas, Rasoul Mahdavi

TA Office Hours: Wednesdays 11:00 - 12:00 EDT

Please use Piazza for questions and clarifications. We will be using Big Blue Button for TA office hours this term; we have separate online rooms for the written and programming parts. To attend office hours, access the corresponding URL for that assignment part and use the corresponding access code when prompted:
Written: https://bbb.crysp.org/b/you-9uu-vgr - Access code: 452674
Programming: https://bbb.crysp.org/b/you-ggq-k6d - Access code: 450936
When asked for your name please enter both your first and last name as they appear in LEARN.

Written Response Questions [26 marks]

For written questions, 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.

Question 1 [Total: 18 marks]

Based on a true story ...

you've been hired by the social media company "ButtonSmash" to analyze a recent security breach on their platform. ButtonSmash employs a username/password authentication system and users are asked to provide an email address and a phone number in case they forget their password (password reset) or want to enable an optional two-factor authentication (2FA).

1. (8 marks) In the following, Identify which of availability, confidentiality, integrity, and/or privacy is violated with a brief explanation.

   (a) (2 marks) Eve, the attacker, was obsessed with the username *@CryptoBob*. Determined to acquire it, she tracked down Bob and followed him for days. Finally, Eve got her shot and caught Bob logging into his ButtonSmash account at a cafe and carefully watched him enter his password.

   (b) (2 marks) Eve quickly tried to log into Bob's account but found out that 2FA was enabled. She then asked Bob if she could use his phone to call her mom and Bob kindly agreed. With Bob's phone number, Eve then did what is known as a *SIM swap* attack. She called the telecommunication company's customer support:

   > **Agent**: "Hello! How can I help you?"
   > **Eve**: "I lost my phone yesterday and I'm waiting for a very important phone call. Could you please port the phone number to my new SIM card?"
   > **Agent**: "Sure, no problem!"

   (c) (2 marks) Bragging about her successful hack on a public forum, Eve got a message from Eva, a vengeful former ButtonSmash employee. Eva knew about an admin page that would give them ultimate power over the platform. She helped Eve to make the following phone call to ButtonSmash's employee office:

   > **Agent**: "ButtonSmash's employee office, This is Bryan speaking."
   > **Eve**: "Hi, I'm Alice and new to the department, Jack told me to contact you for the admin page credentials.
   > **Agent**: "Yes, so the username is *adminsmash* and the password is *button1234*"

   (d) (2 marks) After logging into the admin page, Eve and Eva were surprised by how much damage they could do to the platform. As their first move, they cut out all the other employees from accessing the admin page, making themselves almost unstoppable.

2. (2 marks) It seems like ButtonSmash suffers from a flawed access control design. Propose an access control structure that would properly protect the resource allocation on the admin page.

3. (8 marks) For each of the following scenarios, Answer whether the threat is one of interception, interruption, modification and/or fabrication. Briefly explain your choice of compromise.

   (a) (2 marks) On the admin page, Eve started to change email addresses of multiple accounts to an email address of her own. Then she normally requested a password reset email and changed those accounts' passwords.

   (b) (2 marks) Next, she started reading through private messages of her favorite celebrities' accounts and downloaded all their data onto her hard drive.

   (c) (2 marks) Eve's master plan was to promote a Bitcoin scam. She started posting on behalf of celebrities and political figures that as a COVID-19 charity all bitcoins sent to a specific address would be awarded with double the amount.

   (d) (2 marks) Some of the users noticed the scam and started posting to let other users know that it's not legit. As a counter measure, Eve deactivated those accounts.

Question 2 [Total: 8 marks]

In the aftermath of the attack, FBI started an investigation to arrest Eve. They knew that a few months back, the public forum that Eve was using, got hacked and private messages and email addresses of its users were leaked. Reading through Eve's messages, FBI agents found a bitcoin address that belonged to a cryptocurrency exchange. They then requested from the exchange to provide details of the account associated with that address. It turned out that the email address of the account matched Eve's email address on the forum and even better, Eve had completed the exchange's identity verification. The agents tracked down Eve and realized that she is trying to cash out her gains. Impersonating the buyer, the agents set up a meeting with Eve at a local library and when Eve brought out her laptop to transfer the funds, they connected a flash drive to her laptop that copied all of Eve's data files.

For each of the following methods of defence, explain how Eve could have used it to her advantage to avoid getting caught. Provide context that fits the narrative above.

   (a) (2 marks) Preventing.

   (b) (2 marks) Deflecting.

   (c) (2 marks) Deterring.

   (d) (2 marks) Detecting.

Programming Question [40 marks]

Problem Description

Background

You are tasked with testing the security of a custom-developed password-generation application for your organization. It is known that the application was *not written with best practices in mind*, 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 generating a random password and optionally writing it to `/etc/shadow`. The usage of `pwgen` is as follows:

```
Usage:  pwgen [options]
Randomly generates a password, optionally writes it to /etc/shadow
Options:
 -s, --salt <salt>  Specify custom salt, default is random
 -e, --seed [file]  Specify custom seed from file, default is from stdin
 -t, --type <type>  Specify different hashing method
 -w, --write        Write the password to /etc/shadow.
 -h, --help         Show this usage message
Hashing algorithm types:
0 - DES (default)
1 - MD5
2 - Blowfish
3 - SHA-256
4 - SHA-512
```

Note that the parameters for the options have to be specified like the following: "`--seed=temp.txt`", not "`--seed temp.txt`". If you use the short form of the option, it must be like "`-etemp.txt`" (no "=" between). 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, `pwgen.c`, for further analysis. You will also be provided with some shellcode. The goal is to exploit four different vulnerabilities

in the `pwgen.c` file to end up in a shell with root privileges.

The executable `pwgen` is *setuid root*, meaning that whenever `pwgen` is executed (by any user), it will have the full privileges of *root* instead of the privileges of the user that invokes it. Therefore, if an outside user can exploit a vulnerability in a setuid root program, he or she can cause the program to execute arbitrary code (such as shellcode) with the full permissions of the root user. If you are successful, running your exploit program will execute the setuid `pwgen`, which will perform some privileged operations, which will result in a shell with root privileges. (Note that 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.)

In order to let you learn responsibly about security flaws that can be exploited, we have set up a virtual "user-mode linux" (uml) environment on the ugster machines for each student. This environment contains the vulnerable `pwgen` executable and sample shellcode. You can log in to the uml environment, run a virtual machine and mount your attacks within the virtual machine.

Ugster and UML environment

To access and use the UML environment, go through the following steps:

1. 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. Any questions about your ugster environment should be asked on Piazza.

2. 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.

    - 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.

3. Once you have logged into your ugster account, you can run "`uml`" to start the user-mode linux to boot up a virtual machine.

    (a) To login to the virtual machine, login under the username `user`, with no password.

(b) To leave the virtual machine, use the `exit` command. Then at the login prompt, login as user `halt`, with no password to halt the machine. This returns you to the ugster prompt.

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

5. **Before making any changes in the uml environment:** note that any changes that you make are lost when you exit (through the `halt` command or the uml crashes).

   (a) The directory `~/uml/share` on the ugster machine is mapped to `/share` on the VM, and so files in `/uml/share` on the ugster machine can be accessed from `/share` on the VM. 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.)

   (b) In light of the above point, it can be helpful to `ssh` twice into the ugster machines to work on your exploits. In one shell, log into ugster, start the uml, and compile and execute your exploits. In the other shell, log into ugster and edit your exploit files directly in `~/uml/share/`, so as to ensure you do not lose any work.

   - If you choose to edit files directly in the virtual environment, note that 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 the second ssh session mentioned above.

   (c) Finally, note that the ugster machines are not backed up. You should copy all your work (from `~/uml/share` and elsewhere on the ugster machine) over to your `student.cs` account regularly.

6. This UML contains outdated software in order to allow your exploits to work. The version of the `gcc` compiler installed in the uml environment is very old; it is the same as described in the article "Smashing the Stack for Fun and Profit". This compiler 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. Note that 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!)

Rules for exploit execution

1. You must submit a total of four (4) exploit programs to be considered for full credit. Keep in mind, one (1) exploit must target a *buffer overflow* vulnerability that overwrites a saved

return address on the stack.

2. Each vulnerability can be exploited only in a single exploit program. A single exploit program can exploit more than one vulnerability. (But remember, if you don't have to use more than one, you probably shouldn't! If you do, you won't be able to use it somewhere else it could be more useful.) 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 you are unsure whether two vulnerabilities are different, please ask a private question on Piazza.

3. We will test your exploit programs ("*sploits*") for grading in the virtual environment as follows:

   (a) We will compile your sploits in a virtual machine from a clean `/share` directory into a clean `/home/user` directory in a virtual machine through these commands: `cd /share && gcc -Wall -ggdb sploitX.c -o /home/user/sploitX`. You can also assume that for compiling your sploit, the `shellcode.h` file is available in the `/share` directory.

   (b) Sploits will then be run from a clean home directory (`/home/user`) in the virtual environment, as follows: `./sploitX` (where X=1..4) That is, you should not expect the presence of any additional files that are not already available. If your sploit requires additional files, it has to create them itself.

   (c) Sploits must not require any command line parameters

   (d) Sploits must not expect any user input

   (e) Sploits must not take longer than 60 seconds to complete. Please do not run any cpu-intensive processes for a long time on the ugster machines (see below).

   (f) Running each sploit, as mentioned in point 2 above, should result in a shell owned by root. While you are testing an exploit that returns a shell, you can verify that the returned shell has root privileges by running the `whoami` command within that shell. 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 sploit might look something like the following:

```
user@cs458-uml:~$ ./sploit1
sh# whoami
root
sh#
```

4. To help the grader test your exploit easily:

   (a) 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`.

(b) Give feedback. In case your exploit program might not succeed instantly, keep the user informed of what is going on. (This helps the human grader know to guess that your program may not be running infinitely if it does take some time).

Deliverables, Deadlines and Grading Policy

All assignment submission takes place on the `student.cs` machines (not ugster or the virtual environments), using the `submit` utility. 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.

There are two deadlines for the programming assignment as specified at the beginning of the document: a milestone deadline and an assignment due date.

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

- **sploit1.c**: one sploit that exploits any vulnerability.

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

- **sploit2.c, sploit3.c, sploit4.c:** Three additional completed exploit programs for the programming question
- **a1.pdf:** A PDF file containing your answers for the written-response questions, and the exploit descriptions.

A total of four exploits must be submitted to be considered for full credit. Late submissions for the milestone will not be accepted. This is to encourage you to start early on this assignment. Note that a **submission of at least one buffer overflow** is mandatory regardless of when it's submitted. Marks may be docked if you do not submit a buffer overflow exploit.

Grading

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

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

- how it/they could be fixed (by specific changes to the vulnerable program itself, not by system-wide changes like adding ASLR, stack canaries, NX bits, etc.).

Guidelines

Asking for Help

The TAs are here to help. They unfortunately cannnot complete the assignment for you. The more complete and thoughtful your question, the easier it is for a TA to provide support. Be sure to have carefully read the assignment directions and the recommended support materials before asking a question that reveals you have not taken these steps.

All questions (about the programming and written parts) should be asked **privately** on Piazza under Assignment 1 forum. Additional questions on using ugsters, virtual environment and infodist may also be posted to the same forum.

The TAs may decide to make responses to your private questions public so that everyone benefits from knowing the same information. Questions where you need to post partial solutions or questions that describe the locations of vulnerabilities or code to exploit those vulnerabilities will be sanitized before the TA provides a public response.

The TAs also hold weekly office hours. Information on how to access TA office hours is on page 1 of this handout.

Probably the biggest hint or suggestion we can offer is start early. This is a substantial assignment. It is not an assignment that can be completed in only a few hours. Do not be fooled into thinking that with almost a month until the due date that you can wait to jump in at a later time. Also recognize that as we get closer to the due date, access to the TAs becomes more difficult as we have over 200 students taking the course this term. And enjoy. This is a challenging assignment but it is fun. When else has a prof told you to have -at-it and see if you can create an exploit!

Useful Information For Programming Sploits

The first step in writing your exploit programs will be to identify vulnerabilities in the original `pwgen.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 lectures and supplementary readings available on LEARN
- Smashing the Stack for Fun and Profit (http://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 execve (man execve), pipe (man pipe), popen (man popen), getenv (man getenv), setenv (man setenv), passwd (man 5 passwd), shadow (man 5 shadow), symlink (man symlink), expect (man expect).
- 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)

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

1. `gdb sploitX` (X=1..4)
2. `catch exec` (This will make the debugger stop as soon as the `exec()` function is reached)
3. `run` (Run the exploit program)
4. `symbol-file /usr/local/bin/pwgen` (We are now in the `pwgen` application, so we need to load its symbol table)
5. `break main` (Set a breakpoint in the `pwgen` 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. For better formatting, use "`x/200w <address>`", which prints out 200 words after the specified address.
- "`p <variable>`" and "`p &<variable>`" will give you the value and address of a variable, respectively. You should use the variable name as it appears in the C source code file. Ensure that the variable is in scope.
- "`disas <function>`" gives you the disassembled function and the addresses of its instructions, which is useful to set breakpoints.
- "`break *<address>`" sets a breakpoint at the given instruction's address.
- "`set {int} <address> = <value>`" writes a value to the specified address.
- See one of the various gdb cheat sheets (e.g., http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf) for the various formatting options for the print and x command and for other commands.

Please bear witness to the fact that the version of GDB used in the UML has a bug. Attempting to "`print optarg`" will print the wrong value. See the following posts for details:

- https://stackoverflow.com/questions/9736264/using-getopt-with-gdb
- https://stackoverflow.com/questions/35787697/why-does-gdb-get-wrong-optind-variable-value