UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Winter 2023**
ASSIGNMENT 1

Milestone due date: **Friday, January 27th, 2023 3:00 pm – OPTIONAL**
Assignment due date: **Friday, February 10th, 2023 3:00 pm**

Total marks: 70 (+ 4 bonus points)
Written Response Questions TA: Shufan Zhang
Programming Question TA: Vasisht Duddu, Adrian Cruzat La Rosa

TA Office Hours: Fridays 1:00PM - 2:00 EST

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-ec2-aec-tyb - Access code: 821815
Programming: https://bbb.crysp.org/b/you-pqv-8rg-xsq - Access code: 255717
When asked for your name please enter both your first and last name as they appear in LEARN.

Written Response Questions [30 marks]

You have been hired as a security consultant in a newly-founded company, Cryptopia. Due to its past negligence of its information systems and production machines' security, the company has been hit with serious cyber attacks. Now, you have been tasked with investigating and evaluating the recently reported security breaches in order to prevent similar future attacks. To this end, you have to identify the attack methods that the attackers are using, propose security controls and enforce strong defence mechanisms.

1. (6 marks) For each of the reported security breaches, please 1) identify which one(s) of the security CIA properties is (are) compromised, and 2) describe a possible attack approach leading to the breach.

   (**Note:**: There are various security flaws in Cryptopia's cyber infrastructure.)

   (a) Cryptopia's internet-connected and smart production machines *were functioning normally since their deployment*; However, on the day Cryptopia's executives were meeting

with an important investor for checking the facility, the employees found they *lost control* over *all* the machines.

(b) Cryptopia's marketing leader lost a bid for an important transaction to a competing company. An internal investigation shows that the bidding details were leaked when the marketing leader was uploading them to the auction portal.

(c) The company's logs revealed that a user, with username *"surprise"*, had tampered with the configuration of the production system to cause it to slow down. Further investigation revealed that user *"surprise"* had no password.

2. (6 marks) Cryptopia did not have a formalized way to review, test, and maintain the codebases powering its production systems. You have been tasked with formalizing the process to ensure proper security controls. Please state and explain a security control you would recommend during each of the following phases of development: Code Review/Testing/-Maintenance.

3. (8 marks) After identifying the attack methods, you have been asked to reinforce the defence mechanisms. For each defence method studied in class: *prevent*, *deter*, *deflect*, *detect*, *recover*, discuss how they could be used to defend against *any one* of the attack approaches mentioned in question (1). More specifically, explain how the defence method would apply in the narrative context of the attack, and why it would help against it. (Note: You only need to explain the defences for *one* attack).

   Example:

   Prevent. In case of a Man-in-the-middle attack, encrypt bidding data sent to the auction portal. Encrypted bidding data would be of no use to the attacker.

   (a) Deflect

   (b) Detect

   (c) Recover

   (d) Deter

4. (6 marks) The IT department in Cryptopia proposed a list of custom *two-factor* authentication schemes that protect access to the production mobile controllers (company-owned smartphones used to control the machines). You have been asked to review these proposals. Indicate whether you would accept or reject the proposals below and explain the reason(s) behind your decision. *If you reject a proposal, propose an alternative.*

   (a) The scheme unlocks a controller if a correct password and a correct PIN are entered.

(b) The scheme unlocks a controllers if a correct password is entered or a correct number (received via email) is entered.

(c) The scheme unlocks a controller if the user enters a correct password exclusively within the company's premises.

5. (4 marks) Identify the type of the following pieces of malware – i.e., whether the malware is a worm, Trojan, Ransomware, and/or Logic Bomb. Give a brief description of **how it spreads** or **how a computer becomes infected**, and **the resulting effect**. (A malware may be classified into more than one type.)

   (a) WannaCry

   (b) Code Red

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 *a few 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, they 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 responsibly let you learn 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 upon a crash of user-mode linux).

   (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 – Thus it is important to remember to keep your working files in `/share` on the virtual 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.)

   (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, two (2) exploits must target memory vulnerabilities (e.g., *buffer overflow* and *format string* vulnerabilities) that overwrite 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) 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.

   (b) 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 for compiling your sploit, the `shellcode.h` file is available in the `/share` directory.

   (c) 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. 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.

   (d) Sploits must not require any command line parameters

   (e) Sploits must not expect any user input

   (f) 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). None of the exploits are designed to take more than a minute to finish.

   (g) 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. 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`.

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 **optional** **milestone deadline**, you *may choose* to hand in:

- **sploit1_milestone.c**: One (1) exploit program that exploits any vulnerability. Note that we will build your sploit programs **on the uml virtual machine**.
- **a1_milestone.pdf:** A PDF file containing the exploit description for **sploit1_milestone.c**.

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

- **sploit2.c, sploit3.c, sploit4.c:** The remaining completed exploit programs that satisfy the exploit guidelines §.
- **(sploit1.c):**

  - If you choose not submit anything for the milestone, you are required to submit the other remaining exploit program as **sploit1.c**.
  - If you submit the milestone, you may modify and *resubmit* your milestone exploit by the assignment deadline. Note that **feedback will not be offered on your milestone submission**. If you choose to resubmit, rename your exploit from **sploit1_milestone.c** $\rightarrow$ **sploit1.c**. We will take the higher grade of the milestone submission and final submission. If you do not modify your exploit in the milestone submission you are not required to submit **sploit1.c**.

- **a1_responses.pdf:** A PDF file containing your answers for the written-response questions and the exploit descriptions.

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.)

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 two memory vulnerabilities** is mandatory regardless of when it's submitted.

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.).

For the milestone submission: a maximum of 4 bonus marks will be awarded as follows: your milestone exploit code and description will be assigned a grade (e.g., 7/10) and the bonus marks awarded will be proportional to that grade (e.g., a grade of 5/10 is 50%, so the bonus marks awarded is 2). **Your milestone exploit may address a memory vulnerability or a non-memory vulnerability**.

Guidelines

Asking for Help

The TAs are here to help. They unfortunately cannot 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. Also recognize that as we get closer to the due date, access to the TAs becomes more difficult as we have over 120 students taking the course this term. And enjoy – This is a challenging assignment but it is fun.

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
  (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 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).
- 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 be aware 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