UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 453/698 Computer Security and Privacy — Assignment 1 Winter 2025

Assignment due date: **January 31**

Total marks: 60
TA: Andre Kassis
TA Office Hours: Fridays 1:00pm to 2.00pm (online). To attend office hours, access the following URL and use the corresponding access code when prompted: https://bbb.crysp.org/rooms/i01-wb5-pvq-hc9/join - Access code 6gx9wq

What to hand in
All assignment submissions take place on the Dropbox created on the Learn Course Page.

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

- **sploit1.c, sploit2.c,sploit3.c:** The exploit programs that satisfy the rules in §Exploit Guidelines.
- **a1_responses.pdf:** (*Optional*) A PDF file containing a description for each of the submitted sploits. While submitting this file is optional, we highly encourage you to do so, as if you fail to write successful exploits, you may still be awarded part marks based on your written descriptions.

Important: If you choose to submit **a1_responses.pdf**, this file must contain, at the top of the first page, your: name, WatIAM user ID and student number. **We will not be accepting handwritten 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.)

Assignment Description

i. 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 *two 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.

ii. 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. The goal is to exploit **two** 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 by default you have sudo access in the virtual environment. However, to get any marks, you will need to exploit vulnerabilities in the application, and NOT simply use sudo access to open a root shell.)

In order to let you learn responsibly about security flaws that can be exploited, we have set up virtual environment on the ugster machines for you. We have also provided a few scripts along with the vulnerable pwgen.c file at `http://ugster72a.student.cs.uwaterloo.ca/a1/`. You can use the `portal.sh` script found at this URL to create a new VM, destroy an existing VM or ssh into an existing VM. To learn more about the `portal.sh` script and how to use it, please visit `http://ugster72a.student.cs.uwaterloo.ca:8000/`. Once inside the VM, you should run the following command from the `/home/vagrant` directory to intialize the VM for assignment1:

```
wget http://ugster72a.student.cs.uwaterloo.ca/a1/a1_setup.sh && chmod
+x a1_setup.sh && ./a1_setup.sh
```

This will ensure that the pwgen.c file is available for you in the `/home/vagrant` directory, and an executable for the pwgen.c with appropriate privileges will be available for you in the `/usr/local/bin` directory. You can then mount your attacks within the VM.

iii. Exploit Guidelines

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

A) **Memory vulnerabilities** that corrupt the program memory (i.e., stack or heap) in order to enable a malicious attacker to execute arbitrary code. These pertain to buffer overflows and format string vulnerabilities.

B) **Non-memory vulnerabilities** that *do not* corrupt the program memory in order to execute arbitrary code. Examples that have been presented in class include TOCTTOU vulnerabilities and similar techniques.

Task: You must submit a total of **three** exploit programs. One program must target **a memory vulnerability**, and two must target **non-memory vulnerabilities**. Each program must exploit a **different** vulnerability to gain a root shell.

Requirements:

1. **The two classes are exclusive. This means that you must submit two non-memory vulnerabilities and one memory vulnerability. If you cannot identify and exploit the required number of vulnerabilities belonging to each class and instead submit additional exploits for vulnerabilities belonging to the other category, these additional programs will not be graded (you will be awarded 0 marks for them).**
2. Memory vulnerabilities must manipulate the return address on the stack and force the program to execute the attacker's provided shellcode to obtain a root shell.
3. **You cannot reuse the same vulnerability in different exploit programs!**
4. 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.

iv. Grading

As a first step, you must submit the public and private SSH keys generated when using 'register' in `portal.sh`. Keeping these SSH keys safe is very important, as they provide a way to authenticate you when managing your VMs on the ugster machines. **Next**, a total of **three** exploits (**sploit1.c, sploit2.c,sploit3.c**) must be submitted.

The memory vulnerability exploit is worth 30 marks, while each non-memory exploit is worth 15 marks. Unless you choose to submit written descriptions as explained above, you will be awarded marks on an all-or-nothing basis for each of the exploits (i.e., an exploit that fails to obtain a root shell will be given 0 marks). However, if you submit a written response for such a failed exploit that demonstrates a valid approach you might have failed to implement perfectly, you will still receive partial marks.

v. Testing Setup

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

1. We will compile your exploit programs from a clean `/home/user` directory in a virtual machine through these commands:
   `gcc -Wall -ggdb sploitX.c -o /home/user/sploitX`.
2. Sploits will then be run from a clean home directory (`/home/user`) in the virtual environment, as follows: `./sploitX` (where X $\in \{1, 2, 3\}$) 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.
3. Sploits must not require any command line parameters.
4. Sploits must not expect any user input.
5. Sploits must not take longer than 120 seconds to complete. Please do not run any CPU-intensive processes for a long time on the ugster machines (see below).
6. Running each sploit 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 exploit code might look something like the following:

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

To help the grader test your exploit easily:

1. 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.
2. 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).

vi. Other Useful Information

Useful Information For Programming Sploits

The first step in writing your exploit programs will be to identify memory and non-memory vulnerabilities in the original `pwgen.c` source code.

Most of the exploit programs do not need writing much code. Nonetheless, we advise you to start early since you will likely have to read additional information to learn how to find and exploit a vulnerability. We suggest that you take a closer look at the following items:

- Module -Program Security (Course Slides)
- Smashing the Stack for Fun and Profit (https://www.eecs.umich.edu/courses/eecs588/static/stack_smashing.pdf)
- Exploiting Format String Vulnerabilities (Sections 1-3 only) http://julianor.tripod.com/bc/formatstring-1.2.pdf
- The manpages for execve (man execve), system (man system), popen (man popen), setenv (man setenv), passwd (man 5 passwd), shadow (man 5 shadow), symlink (man symlink), expect (man expect), umask (man umask)
- Environment variables (e.g., https://en.wikipedia.org/wiki/Environment_variable)

You are allowed to use code from any of the above webpages as starting points for your sploits, and do not need to cite them. However, the system with which you are experimenting is a 64-bit system, and thus, the memory layout differs from the 32-bit one discussed in the reading material. As a result, your memory vulnerability exploits must be designed to address this different memory layout.

Advice from the TAs — IMPORTANT

1. 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 -nx -readnow -fullname -quiet -args /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)

2. Since the target program contains a number of interactive prompts and you are asked to have your exploits run without requiring any user intervention, you **might** need tools for automating user input. Specifically, you are encouraged to work with the program `expect`. To simulate Ctrl+D (EOF) with `expect` you should execute `send \x04`. Additionally, keep in mind that `expect` tracks the input and output channels of the process it spawns by its process ID. Hence, running a script (program) which invokes `expect` as a sub-process from within your code will not work (e.g., by using the library function `system()`). Instead, use `execv()` or similar functions that replace your current process's code with that of the desired `expect` program. Refer to the man pages for further details.

GDB: GNU Debugger

The GNU debugger will be useful for writing some of the exploit programs. It is available in the virtual machine after you run the `a1_setup.sh` script. 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 -nx -readnow -fullname -quiet -args sploitX` ($X \in \{1, 2, 3\}$, this will start sploitX in the debugger and later let you set breakpoints inside the pwgen binary.)
2. `catch exec` (This will make the debugger stop as soon as the `execve()` function is reached— i.e., pwgen is loaded)
3. `run` (Run the exploit program)
4. `break main` (We are now in the `pwgen` application, so set a breakpoint anywhere we want)
5. `cont` (Run to breakpoint)

You can store commands 2-5 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., http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf) for the various formatting options for the `print` and `x` commands.

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

Questions

For questions about the assignment, ugsters, virtual environment, 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.