CSCI 3330 Assignment 2

Deadline: September 25, 2025, 10:00pm

1 Background

In this assignment, we will be further exploring privilege escalation, command line injection vulnerabilities, and directory traversal vulnerabilities. Specifically, by completing this assignment, you will:

- Explain how file system permissions establish security boundaries and evaluate designs against the principle of least privilege.
- Diagnose insecure system/web patterns by executing attacks and justifying mitigations.
- Translate abstract security concepts into practice by planning, executing, and validating end-to-end privilege transitions and access controls.
- Communicate security reasoning succinctly: narrate the attack path, identify root causes, and recommend organization-ready fixes grounded in policy and engineering.

2 AI policy

You may use AI to learn background concepts. You may not use AI to generate, verify, or iterate on solutions for this assignment.

Allowed:

- Concept definitions and explanations: "What is cron?", "What does `chmod u=rwx,g=,o=` do?", "What does `execlp()` do?", "What is a system call?"
- Generic examples not derived from this assignment: "Walk through an example of a command line injection attack," "Walk through an example of a directory traversal attack."
- Non-generative tooling: man pages, official documentation, calculators, programming languages, and trying something for yourself.

Handle with care:

- Tools which improve grammar or clarity.

- IDE autocompletion or documentation lookup that does not inject solution content.

Not allowed:

- Any hint, outline, or solution for these specific questions (including rephrased or "similar" versions), e.g., "Walk me through the attack in Q2," "Write a crontab to get a root shell," or "Check my answers."
- Pasting assignment text or numbers into AI to tailor examples.
- Using AI to generate any text, code, calculations, or any other artifacts that you submit.

Ask me questions instead! When in doubt, don't use AI, because **you won't have access to it on tests**, for which these assignments are your preparation.

3 Questions

1. You would like to share a directory full of secrets with a chosen set of users on a shared server. Simple — just add those users to a group and give that group permission to access the secrets directory, right? Unfortunately, you are not an administrator of the server, so you cannot create groups or add users to them, and so you must implement your own access controls.

   To do this, imagine you set up the following environment:

   ```
   mkdir -p /var/tmp/secrets-gate/secrets
   # set rwx------ to /var/tmp/secrets-gate
   chmod u=rwx,g=,o= /var/tmp/secrets-gate
   # set rwxr-xr-x to /var/tmp/secrets-gate/secrets
   chmod u=rwx,g=rx,o=rx /var/tmp/secrets-gate/secrets
   echo 'My secret file content' > \
       /var/tmp/secrets-gate/secrets/example-file.txt
   echo 'More secret file content' > \
       /var/tmp/secrets-gate/secrets/example-file-2.txt
   # set rwxr--r-- to secret files
   chmod u=rwx,g=r,o=r /var/tmp/secrets-gate/secrets/*
   ```

   At this point, we have created two secret files that no one has permission to access but you. To facilitate access to them, we will create a setuid "you" program `gatekeeper.c` that allows users with the right password to access these files but no others as follows:

   ```
   #include <stdio.h>
   #include <string.h>
   #include <unistd.h>
   ```

```
#define PASSWORD "password1"
#define SYSCALL1 /* determine this in part (a) */
#define SYSCALL2 /* determine this in part (b) */

int main(void) {
    char *password = getpass ("Enter password: ");
    if (password == NULL) {
        perror ("getpass");
        return 1;
    }
    if (strcmp (password, PASSWORD) != 0) {
        fprintf (stderr, "Invalid password!\n");
        return 2;
    }
    if (chdir ("/var/tmp/secrets-gate/secrets") != 0) {
        perror ("chdir");
        return 3;
    }
    uid_t uid = SYSCALL1 ();
    if (SYSCALL2 (uid) != 0) { /* drop privileges to original user */
        perror (#SYSCALL2);
        return 4;
    }

    execlp ("bash", "bash", NULL);
    perror ("execlp");
    return 5;
}
```

However, there are two Unix system calls left as placeholders in the code.

Consider the following Unix system calls:

- `getuid`: gets real ID of current user
- `geteuid`: gets effective ID of current user
- `setuid`: sets real and effective ID of current user
- `seteuid`: sets effective ID of current user

For this exercise, submit answers to the following questions:

(a) To ensure that privileges have dropped back to those of the original executor of the problem, which of the above system calls should be made in place of the `SYSCALL1` placeholder?

(b) And which of the above system calls should be made in place of the `SYSCALL2` placeholder?

(c) What would be a better way of checking a password other than by hard-coding the password into the program?

(d) Other than the setuid bit, what file system permissions should the compiled gatekeeper executable be configured with so that users can execute it? Should we give it read permissions? Why or why not?

(e) Why do we have more restrictive permissions on the outer directory (secrets-gate) but more permissive permissions on the inner one (secrets)?

(f) Why is it important for the purposes of security to check the return code of `SYSCALL2` in this code?

2. In this exercise we will be obtaining a root shell on a server. The server is sandboxed and available here. The page shows a Web service front end for uploading files, but it also displays the console of the server backend as experienced by an unprivileged user. Refreshing the page creates a new instance of the server, restoring it to its original state before any meddling, so refresh the page if and only if you desire to reboot the server and lose any changes you made to it.

   The ostensible purpose of this Web page is to provide a Web service converting uploaded images to the WebP format, which typically offers better compression than JPEG. However, this service is running as root and features a command-line injection vulnerability, which we will use to obtain a root shell on the server. For your benefit, the source code for the daemon containing the vulnerability is available here. For your benefit, there is also a form field to override the name of the uploaded file to make it easier to experiment with different uploaded file names. Although there may be multiple ways to get a root shell on the server using a command line injection attack, I recommend performing an attack as follows:

   (a) First, execute a command line injection attack to set the setuid bit on the `/usr/bin/python3.12` binary.

   (b) On the server backend, execute python3.12 as the unprivileged user.

   (c) In your python REPL, call `os.geteuid()` to confirm your root privileges.

   (d) `/bin/sh` has an annoying "feature" where it protects against being run as a setuid root process or the child of one by checking if your real UID is different than your effective UID. What python3 call(s) in the `os` module can we make so that your real UID will match your effective UID?

   (e) Finally, in your python3 REPL, call

   ```
   os.execv('/bin/sh', ['/bin/sh'])
   ```

   to launch the shell. If your attack worked, running `whoami` should print `root`.

For this exercise, after you have obtained a root shell, submit answers to the following questions:

(a) What "filename" string did you use to execute your command line injection attack? **Background:** You can use the `chmod` command to set the setuid bit on a binary by running `chmod u+s <file>`.

(b) What call(s) in the `os` module did you make so that your real UID matched your effective UID? Or, if you obtained a root shell on the server using a command line injection attack another way, tell me about that!

(c) How can we fix the code in `webpd.py` to mitigate the command line injection vulnerability?

3. The previous examples of directory traversal attacks we have looked at show how they can be used to read files on a server. However, in this exercise we will see that they can also be used to write files and that writing a carefully chosen file to a carefully chosen location can, in fact, get us a root shell.

   Using the previous question's server environment, execute a directory traversal attack to obtain a root shell on the server. Although there may be multiple ways to get a root shell on the server via a directory traversal attack, I recommend performing an attack as follows:

   (a) Construct an `/etc/cron.d/` cron job that runs every minute as root. Based on the previous exercise, what would be a good command for the cron job to run?

   (b) Upload it to the `/etc/cron.d/` directory using a directory traversal attack.

   (c) Wait a minute.

   (d) Perform the remainder of the attack necessary to acquire a root shell.

   For this exercise, after you have obtained a root shell, submit answers to the following questions:

   (a) What "filename" string did you use to execute your directory traversal attack?

   (b) What cron entry did you use to execute your directory traversal attack? Or, if you obtained a root shell on the server using a directory traversal attack another way, tell me about that!

   (c) The directory traversal vulnerability does not exist in `webpd.py`, which is only a daemon that handles image conversion. Rather, it exists in the Web service which writes your uploaded files to disk before `webpd.py` processes them. After experimenting with different ways of executing directory traversal attacks (namely, via absolute path or via relative path), which succeed and which do not?

(d) Informed by the above experimentation about which attempts succeeded and which failed, speculate using pseudocode or code in any language what you think the service's code for writing uploaded files to the file system might look like, highlighting the vulnerability.

Background: Cron is the task scheduler on most Unix machines. Among other places, on this system you can schedule tasks to run by placing a specially formatted file in the `/etc/cron.d/` directory. To learn more about the format of `crontab`, which is the file format used in `/etc/cron.d/`, consider reading the official documentation and experimenting with this cron calculator.

4 Submission

Email your answers in a single `*.txt` or `*.pdf` file attachment to my email address in the Syllabus.