Systems Programming Project 3

May 6, 2025

1 Introduction

CAREFULLY READ THE *ENTIRETY* OF THIS DOCUMENT. DEVIATIONS FROM THE INSTRUCTIONS IN THIS DOCUMENT *WILL* BE PENALIZED.

IF YOU HAVE QUESTIONS OR NEED CLARIFICATION ON SOMETHING IN THIS PROJECT DESCRIPTION, YOU *MUST* INCLUDE THE RELEVANT TEXT FROM THIS PROJECT DESCRIPTION YOU ARE ASKING ABOUT IN YOUR EMAIL, OR WE WILL ASSUME THAT YOU HAVEN'T FULLY READ THE PROJECT DESCRIPTION AND *WILL NOT* RESPOND. LACK OF RESPONSE TO EMAILS THAT DO NOT FOLLOW THIS REQUIREMENT WILL NOT BE CONSIDERED AN EXCUSE FOR DEVIATING FROM THE DIRECTIONS OUTLINED IN THIS DOCUMENT.

2 Hashassin

Now that we have our basic program up and running, we will build a network server around it.

This is a group project with a maximum of *two* students per group.

NB: Group size is non-negotiable. Attempts to negotiate group size will be considered attempting to acquire an unearned grade.

3 Deliverables

Your submission should be made to GitHub classroom.

- GitHub Classroom link: `https://classroom.github.com/a/hH-0wI7C`

- 11:59PM, May 15, 2025. **NB: This is a hard cut off and you will not be able to commit to this repository when the due date passes without a late penalty. You should add at least your `HONESTY.md` file to this repository as soon as you accept it to ensure you can do a pull request if you happen to be late.**

4 Instructions

For this project, you will be extending your solution from Project 2 and building a TCP client and server.

As with Projects 1 and 2, your binary crate should be in a directory called `cli` and it should compile to a binary named `hashassin`.

Your library crate should be in a directory called `core` and it should be available called `hashassin-core` and usable by other crates via `use hashassin_core::Whatever`.

In addition to the above, you will create at least two additional crates: 1) `client`, which should be available to other crates via `use hashassin_client::Whatever` and `server` which should be available to other crates via `use hashassin_server::Whatever`. These crates should be where you put your client and server related code, respectively.

Your CLI will will support two additional commands: 1) `server` and 2) `client`.

`server` launches a network server that responds to requests from clients. This server must accept two commands

`upload` which will accept rainbow table from a client that can be used for cracking passwords. **NB:** Uploaded rainbow tables should persist for at least as long as the server is running. I.e., multiple rainbow tables can be uploaded at the same time and *all* these rainbow tables will be available to any clients that want to `crack` them.

`crack` which will accept a hashes file from a client and attempt to crack them. Output should be in the same format as project 2. When a client requests a `crack`, *all* rainbow tables that have been uploaded since the server was running should be used to attempt to crack the hashes

`server` *must* have several options that can be set:

1. `--bind`, which specifies the ip address your server should bind to. `--bind` should default to `127.0.0.1`.
2. `--port`, which specifies the port youare server should bind to. This should be a non-zero `u16`. `--port` must have a default value of 2025.
3. `--compute-threads`, the *total number of threads* that are available to the server to crack passwords. I.e., there should never be more than this number of threads available to crack passwords, regardless of the number of clients that have requested a `crack` at the same time.

This value *must* be greater than zero and the maximum value should be the maximum length of an array on whatever system the program is being run on. `--compute-threads` must have a default value of 1.

4. `--async-threads` which will set the number of threads that your async runtime uses for tasks. This option must have a7 default of 1.
5. `--cache-size` is an *optional* value that indicates that already cracked passwords should be cached, and that the cache should be no larger than the maximum value of an `i32` bytes.

`client` which has two subcommands: 1) `upload` which will upload a rainbow table to the server and 2) `crack` which will have the server attempt to crack a supplied hashes file.

`upload` must support several options:

- `--server` which will be the address of the server to connect to. E.g., "127.0.0.1:2025"
- `--in-file` which is the rainbow table file to upload to the server.
- `--name` which is a name that can be associated with the rainbow table being uploaded.

`crack` must support several options:

- `--server` which will be the address of the server to connect to. E.g., "127.0.0.1:2025"
- `--in-file` which is input hashes file in the same format as existing hash files.
- `--out-file` which is a *optional* and is the path to a file to save output from the server to. If `--out-file` is not specified, output should be to stdout.

Command formats:

The format for *upload* consists of a header with a few fields and a payload section.

- `MAGIC WORD`: The first n bytes of an upload command should be a utf8 encoded string "upload" (all lower case).
- `VERSION`: The next byte after the end of the magic word *must* be a version number. Unless otherwise specified in a future update, this should always be `1`.
- `NAME LENGTH`: The byte following the version number should be the length of the name of the upload.
- `NAME`: The `NAME LENGTH` bytes following the name of the upload should be the utf8 encoded name of the upload.
- `PAYLOAD SIZE` the next 8 bytes should be the length of the payload in bytes. E.g., a value of 1 here would mean that the payload is one byte total.

- `PAYLOAD`: the next `PAYLOAD SIZE` bytes should be a rainbow table, in the same format as specified for project 2.

The format for *crack* consists of a header with a few fields and a payload section.

- `MAGIC WORD`: The first n bytes of an crack command should be a utf8 encoded string "crack" (all lower case).
- `VERSION`: The next byte after the end of the magic word *must* be a version number. Unless otherwise specified in a future update, this should always be `1`.
- `PAYLOAD SIZE` the next 8 bytes should be the length of the payload.
- `PAYLOAD`: the next `PAYLOAD SIZE` bytes should be a hashes file, in the same format as specified for project 1 and 2.

NB: You can add additional functionality that can do other things, but the above commands and formats *must* work by default.

5 Grading

You will receive points according to the following list. Please note that there are more than 100 points available.

-1,000 points: If you do not update `CREDITS.md` with the names of your group members, as well as an honest break down of the work each group member did, you will receive **NEGATIVE ONE THOUSAND** points. I.e., it would be next to impossible to get anything higher than a zero on this project.

-1,000 points: If you do not include an `HONESTY.md` with the academic honesty statement from the syllabus, you will receive **NEGATIVE ONE THOUSAND** points.

-1,000 points: If you do not include a `README.md` with information about your project, libraries used, instructions on how run things, etc., you will get **NEGATIVE ONE THOUSAND** points. Your `README.md` should also include the items in the grading rubric that you attempted. Only items you note in your `README.md` file will be graded!

-1,000 points: If running `cargo fmt` results in any change to your repository, you will get **NEGATIVE ONE THOUSAND** points. Be sure to run `cargo fmt`!!!!!!!!!

-1,000 points: Program does not compile to correct binary/library name.

25 points: Program compiles.

35 points: All required functionality of `server` is implemented (e.g., setting `--compute-threads` actually uses multiple threads to do cracking, etc.).

- 5 points: `--bind`.
- 5 points: `--port`.
- 5 points: `--compute-threads`.
- 10 points: `--async-threads`.
- 10 points: `--cache-size`.

30 points: All required functionality of `client` is implemented.

- `upload`.
  - 5 points: `--server`
  - 5 points: `--in-file`
  - 5 points: `--name`
- `crack`
  - 5 points: `--server`
  - 5 points: `--in-file`
  - 5 points: `--out-file`

5 points: Comprehensive documentation. Points will be determined by looking at the documentation built when running `cargo doc --document-private-items --no-deps`.

2.5 points: No warnings from `cargo check` with no use of directives that would suppress default warnings (e.g., `#[allow(dead_code)]`)

2.5 points: No warnings from `cargo clippy` with no use of directives that would suppress default warnings (e.g., `#[allow(dead_code)]`)

2.5 points: Proper error handling. Create and use proper error types in library code and appropriate handling in any user code (e.g., CLI).

2.5 points: No unwraps or expects. Add `#![deny(clippy::unwrap_used, clippy::expect_used)]` to the top of your `main.rs` and `lib.rs`. If running `cargo clippy` doesn't result in any errors, then you get the 5 points.

2.5 points: Add proper logging. Using the `tracing` crate to generate logs. Points will be determined on comprehensiveness of logs, as well as the usage of different log levels.

10 points: Performance report. In your repository, create a file `PERFORMANCE.md` that shows the performance characteristics of your program. E.g., how does it perform with respect to the number of clients that are connected vs. the number of async threads and the number of compute threads?

Plots and tables would be pretty cool. The report is not required to conform to any particular formatting (besides markdown), but please try to make it look decent.

Unlimited points: Cool factor. Points are determined subjectively by me, but if I find anything about your project to be particularly unique, difficult, clever, etc., you can get some extra points. Be sure to point out anything we should pay special attention to in your `README.md` file.

In addition to the above, you must have a README.md to summarize what you did and give any and all instructions on how to run things! We're going to go off what's in the README.md to grade, so make sure you tell us everything you did and test all the instructions, etc.

5.1 Allowed Crates

Generally speaking, you are free to use any crates listed on `https://blessed.rs`.

You can also use the `hex` crate (`https://docs.rs/hex/latest/hex/index.html`) as well as the `num` crate (`https://docs.rs/num/latest/num/`).

It is also ok to use the `bon` crate (`https://docs.rs/bon/latest/bon/`) to help you build more ergonomic APIs (especially using the builder pattern).

If you are going to implement caching, we suggest using the `stretto` (`https://github.com/al8n/stretto`). It is not exactly a simple crate, but if you read through the documentation and examples, it's relatively straight forwards for use in this project. One thing in particular to note: you do not need to wrap a stretto cache in an `Arc<Mutex<T>>` since it does all that internally.

All tokio project crates are allowed as well. I.e., if the repository of the crate is part of the `tokio-rs` org, it's fine to use even if it's not explicitly listed on blessed.rs.

Any additional crates not on Blessed.rs need to be approved by Jeremy.

Be sure to note what crates you use in your `README.md`!