CITS3007 group project – phase 2

Version:	0.2
Date:	30 Apr 2025

Changes in 0.2

- Clarified that test code should be submitted (sec 3.7)
- Removed report content duplication (duplicated reflection/presentation)
- Code revised to
  - add banned.h
  - add libcheck .ts rules
  - remove spurious `log_fd` from `handle_login`
  - fix Makefile and stubs.c

1. Background

As described in the phase 1 project specification, your team work as software developers for **Enjjin Media**, and you are implementing part of the **Oblivionaire Online (OO)** online game.

1.1 Your tasks

You are working on portions of the Access Control System, which forms a critical security boundary between the outside world and the game's internal infrastructure. You will be responsible for implementing a set of C functions that handle account management, password validation and updates, session management, and user authentication logic.

You are provided with:

- A set of header files (`account.h`, `db.h`, `logging.h`, and `login.h`) defining the data structures and function prototypes relevant to the ACS.
- Stub implementations of several supporting functions (e.g. logging, database lookups).
- A Makefile to build the project.

- A simple packaging system for declaring any additional third-party dependencies you use (see `apt-packages.txt` and `libraries.txt`).

Your main programming task is to implement a number of C functions across multiple files.

In addition to the code, you will submit a (brief) written report. This should:

- Justify your design decisions (e.g. why you chose a particular password hashing algorithm or library).
- Discuss any difficulties you encountered and how you addressed them.

The written report is not expected to be formal or long, but it should demonstrate that you have thought critically about your implementation.

Section 3 of this document lists the specific functions you are required to implement.

2. Project rules and deadlines

- This phase – phase 2, implementation and report – is worth 40 marks and is due on Wednesday 14 May (11:59 pm) 2025 (week 11).

  The marks awarded to individual students will be the raw phase 2 mark multiplied by a student's Group Contribution Factor, calculated as described previously.

- Phase 3 (demo/presentation) is delivered in **weeks 11-12** and is worth 10 marks.

  In phase 3 (final demonstration/presentation for the project) you'll be expected to reflect on your plans from phase 1, the challenges you faced and how you dealt with them.

2.3 Academic conduct

You are expected to have read and understood the University Academic Integrity Policy. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own group's effort.

2.4 Penalties for late submission

For phase 2, you must submit your project before the submission deadline. The standard penalties for late submission apply.

2.5 Revision and effort tracking

All members of a group are expected to contribute equal effort to the project.

As previously discussed, we recommend you maintain:

- a **private** Git repository (hosted on a service such as GitHub or GitLab) containing the code for your project, and
- a shared spreadsheet (e.g. on MS Teams or Google Docs) tracking tasks allocated to and completed by members each week.

Neither the repository nor spreadsheet are submitted for assessment, but the Unit Coordinator may ask to review them.

3. Phase 2 – design and implementation

3.1 Phase 2 deliverables

The phase 2 deliverables are **code** and a brief **report**.

3.2 Code deliverables: required functions

Unless otherwise stated, all functions have the following contracts:

- **String arguments** must be null-terminated strings. It is a **precondition** that callers satisfy this requirement. Functions must not attempt to validate this themselves – doing so in any case is not possible in C.
- **Pointer arguments** must be non-`NULL` unless explicitly stated otherwise in the function description. It is the caller's responsibility to ensure this.

You are encouraged to document these preconditions in your own code as well.

3.3 Account management

```c
account_t *account_create(const char *userid, const char *plaintext_password,
                          const char *email, const char *birthdate);
```

Preconditions:

- All arguments must be valid, null-terminated strings.
- None of the pointers may be `NULL`.

Creates a new user account with the given parameters (and defaults for any other fields). The password must be hashed securely (see Section 3.4, "Password handling"), and the resulting account must be dynamically allocated on the heap. The caller is responsible for freeing any allocated memory with `account_free`. The birth date needs to be checked to ensure it is a valid date in the correct format, and a basic check should be performed on the email: it should consist only of ASCII, printable characters (according to the C standard) and must not contain any spaces. (Other portions of the system will require the user to verify that they can read messages sent to that email address.)

On error, returns `NULL` and logs an appropriate error message using `log_message`.

```
void account_free(account_t *acc);
```

Preconditions:

- `acc` must be a pointer returned by `account_create`, or `NULL`.

Releases any memory associated with an account.

```
void account_set_email(account_t *acc, const char *new_email);
```

Preconditions:

- `acc` and `new_email` must be non-`NULL`.
- `new_email` must be a valid, null-terminated string.

Safely update the account's email address.

```
void account_set_unban_time(account_t *acc, time_t t);
void account_set_expiration_time(account_t *acc, time_t t);
```

Preconditions:

- `acc` must be non-`NULL`.

Set the unban or expiration time of an account; look at the code in `account.h` for details of how these work.

```
bool account_is_banned(const account_t *acc);
bool account_is_expired(const account_t *acc);
```

Preconditions:

- `acc` must be non-`NULL`.

Return true if the account is currently banned or expired, respectively. These functions must compare the current system time with the relevant field in the account structure.

```
void account_record_login_success(account_t *acc, ip4_addr_t ip);
void account_record_login_failure(account_t *acc);
```

Preconditions:

- `acc` must be non-`NULL`.

Update account metadata following either a successful or a failed login. This requires that any failure or logon counts be set correctly, that the last login time be set using the current system time, and the last IP address connected from be set correctly.

Whenever a user logs in successfully, their `login_fail_count` is reset to 0; it's thus a measure of the number of *consecutive* login failures.

Likewise, whenever a user fails to log in successfully, their `login_count` is reset to 0.

```
bool account_print_summary(const account_t *acct, int fd);
```

Preconditions:

- `acct` must be non-`NULL`.
- `fd` must be a valid file descriptor, open for writing.

Print a brief, readable summary of the account to the file descriptor. Write a brief human-readable summary of the account's current status to the given file descriptor. The exact format is up to you, but it should include user ID, email, and login-related statistics (dates, counts and IP addresses). Return `true` on success, or `false` if the write fails.

3.4 Password handling

These functions are responsible for securely hashing and validating user passwords.

The structure of the stored hash, and the hashing algorithm used, are up to you to determine. The only *hard* constraint is that the final stored hash should be a printable string, and it must be no longer than `HASH_LENGTH - 1` in length. Desirable features for password functions have been covered in lectures; it's up to you to choose a library and/or algorithm that is in line with best practices.

```
bool account_validate_password(const account_t *acc,
                               const char *plaintext_password
);
```

Preconditions:

- acc and plaintext_password must be non-NULL.
- plaintext_password must be a valid, null-terminated string.

Checks whether the plaintext password matches the stored hash.

```
bool account_update_password(account_t *acc,
                             const char *new_plaintext_password
);
```

Preconditions:

- acc and new_plaintext_password must be non-NULL.
- new_plaintext_password must be a valid, null-terminated string.

Hash a new password and update the account accordingly. This is similar in structure to account_validate_password, but involves generating a new hash instead of comparing one.

3.5 Login handling

```
login_result_t handle_login(const char *username, const char *password,
                            ip4_addr_t client_ip, time_t login_time,
                            int client_output_fd,
                            login_session_data_t *session);
```

Preconditions:

- username, password, and session must be non-NULL.
- username and password must be valid, null-terminated strings.
- client_output_fd and log_fd must be valid file descriptors, open for writing.

This function implements the business logic for user login. You must:

- Look up the user by ID, using account_lookup_by_userid (a stub implementation is provided);
- Check whether the account is banned or expired, based on the current system time;
- Check whether the account already has more than 10 consecutive login failures;
- Validate the supplied password;
- Record success or failure appropriately;
- Populate the session struct if login succeeds;
- Write appropriate messages to the client_output_fd file descriptor, which send output to the client (i.e., the person trying to log on) as well as to the the system logs (using log_message); and
- Return an appropriate login_result_t.

You are **not** required to implement any actual network or socket code. The client_output_fd represents a writable file descriptor that allows you to simulate communication with the client software.

3.6 Coding standards

The code you submit will be assessed for correctness, security, clarity, and maintainability. General guidelines for code are provided on the CITS3007 website in the FAQ, under "Marking rubrics" / " 'Long answer' questions requiring code".

In addition to those, you are expected to

- Use secure coding practices throughout. This includes avoiding undefined behaviour, and ensuring cryptographic operations are handled appropriately.
- Select and correctly use appropriate libraries for password hashing and validation.
- Include *meaningful* comments in your code, where necessary, to explain implementation decisions (but see the warning in the FAQ about over-commented code).
- Test your code incrementally as you develop it, making use of unit tests or test scaffolds where appropriate.

Tips

- If, at the time of submission, some portion of your code is not compiling, you should either comment it out, or surround it with "`#if 0` … `#endif`" preprocessor instructions, so that your code does compile.

  It's better to show your intent and have code that compiles, than to submit code that doesn't compile.

- When designing and implementing your functions, consider where the *system boundaries* are – are there points at which data enters your code from outside sources? As discussed in lectures: validate data thoroughly when it first enters your system; once validated, you can generally treat it as trustworthy within internal modules – don't re-validate unnecessarily. This keeps code clearer and avoids redundant checks.

- Use logging (`log_message(...)`) where helpful for debugging and for error reporting.

- Refer to the provided header files and scaffolding code for further details on data structures and expected behaviour.

3.7 Test code

In addition to the implemented functions, you should submit source files containing any test code you have written, together with instructions (in the `README.md`) on how to run your tests.

3.8 Report content requirements

Your report should address the following:

- What design decisions did you have to make? How and why did you decide on the approach you chose?
- What testing approach did you choose for the project? How did you decide what and how to test?

The report need not be long or formal.

Assessment is based on whether the report addresses the content requirements, and the clarity, coherency, and strength of justifications given for choices made.

3.9 Code and report submission

- Submit your code (as a `.zip` file) and your report (as a PDF or Markdown `.txt` file) for phase 2 via Moodle (one submission per group) by the due date.
- Ensure that the name and student number of all group members are included in the submission.
- Ensure that the group name and number are included in the submission.