Phase 2 Report

Group Name: **TOC/TOUah** (Group 33)
Date: Wednesday, May 14, 2025

Name	Student Number
Cooper Thomas	23723986
Elke Ruane	23748615
Fin O'Loughlin	23616047
Marc Labouchardiere	23857377
Mika Li	24386354

GENERAL DESIGN CHOICES

We made the following assumptions:

- **Parameter assumptions**
  Assume inputs such as pointers and strings are valid and conform to the specification (e.g., not null-terminated). This avoids redundant validation and aligns with separation of responsibilities in modular design. Revalidating inputs could introduce undefined behavior or reduce performance unnecessarily.

- **Account data storage**
  `account_t` structs are allocated on the heap and manually freed, as there is no indication of persistent storage. The `db.h` header only provides function declarations, suggesting that in-memory management is expected and sufficient.

- **Logging and DB stubs**
  Since logging and database functions are provided as stubs, our responsibility is limited to calling them correctly. We use `log_message()` to report errors, but we do not handle error recovery internally. This follows the layered approach of the *OO* system design.

- **Client abstraction**
  Treat `client_output_fd` as an abstraction for client communication. It is assumed that file descriptors are valid, and that low-level networking tasks (like buffering and concurrency) are dealt with in higher-levels of the *OO* system design.

- **Time Handling**
  We use `time(NULL)` to obtain the current time for handling bans and expirations. This is the

conventional and portable way to represent "now" in C, and no alternative timing source was specified in the rubric.

- **Thread safety handling**
  Assume that thread safety is managed by the caller or higher-level components. Our functions do not modify shared global state, so internal thread-safety mechanisms would be unnecessary and potentially over-engineered for our low-level implementations.
  `account_t` itself is not necessarily global or shared, but rather how instances of it are accessed (e.g., in `alternate_main.c`). Thus, the caller is responsible for thread safety.

DIFFICULTIES WE ENCOUNTERED

Our group faced the following issues when completing Phase 2, and we addressed them as follows:

- **Ambiguity around thread safety**
  We assumed thread safety was the responsibility of higher-level components and ensured our code avoided shared global state.

- **Ambiguity in the project specs**
  When uncertain, we followed the spec literally and documented assumptions in comments and this report.

- **Choosing field lengths and NULL-terminators**
  We carefully used `strnlen()` and `safe_memcpy()` to avoid overflows and ensured fields did not rely on implicit null termination unless explicitly designed that way.

- **Proper use of stub functions (e.g., logging, DB)**
  We limited our use of stub functions to their documented roles and avoided making assumptions about their internal behavior.

- **Balancing "reasonable assumptions" vs. "project specs"**
  We favored clarity and documented our decisions where specs were ambiguous, aiming for consistency and defensive defaults.

- **Issues over version control robustness**
  We adjusted our workflow to make smaller, well-scoped commits and tested locally before merging, rather than relying solely on pull request reviews.

- **Commenting and code clarity**
  We added concise, meaningful comments where the code wasn't self-explanatory, and avoided over-commenting obvious logic.

- **Not using banned functions**
  We avoided unsafe functions like `strcpy`, `scanf`, and `gets`, and used safer alternatives such as `snprintf`, `strnlen`, and custom memory-safe wrappers.

`account.c` FUNCTIONS

HELPER FUNCTIONS

```
bool validate_email(const char *email, size_t *out_len);
```

- If the input email exceeds `EMAIL_LENGTH`, we assume this was unintended by the caller. In such cases, we log a warning, truncate the email to fit within the allowed length, and proceed with the operation.
- On success, we use `strnlen()` to determine the number of bytes to copy (up to `EMAIL_LENGTH`), ensuring that `memcpy` only copies a valid and bounded range.
- We also encountered ambiguity around what constitutes a "valid" email. Should it include an "@" symbol? Require characters on both sides of it? Allow empty strings? Ultimately, we chose to follow the project specification strictly and only accept ASCII-printable characters with no whitespace. By that definition, even an empty string is considered valid.

```
bool validate_birthdate(const char *bday, size_t *out_len);
```

- Assumes that the caller may reasonably provide `"0000-00-00"` as a default birthdate value, which is explicitly permitted in `account.h`.
- Assumes all dates are permitted. This is strictly following the project specs.
- Assumes the `time.h` library safely validates dates.
- Initially, we encountered an issue with storing the birthdate string: attempting to include a null terminator caused us to overwrite the final byte of the fixed-length `birthdate` field. To resolve this, we chose not to store the null terminator in `account_t->birthdate`, ensuring the full 10-character date format is preserved as specified.

```
bool hash_password(const char *pw, char *out_hash, size_t out_hash_len);
```

- We assume the `pw` string is short enough to be safely passed into `libsodium`'s `crypto_pwhash_str_alg()`. In our testing, the C string length limits kicked in before anything broke on the libsodium side, so we didn't run into practical issues.
- We had to pick a crypto library that was both secure and easy to use. After some trial and error, we settled on `libsodium` because its defaults (e.g., Argon2id with built-in salt) worked well and the resulting hash fit within the `HASH_LENGTH` defined in `account.h`. So we stuck with it.

```
int safe_memcpy(char *dest, const char *src, size_t max_len);
```

- Designed to check for overlapping `src` and `dest` buffers, before calling `memcpy`. Prevents undefined behaviour for overlap in `memcpy` (e.g. partial copying)

- Assumes `dest` and `src` are valid, non-NULL pointers
- We originally used a `safe_strncpy(...)` function to copy strings into the account struct fields. However, we ran into issues because storing a null terminator meant we couldn't use the full fixed-length field (e.g., 10 bytes for the birthdate). Since the struct fields don't allow any extra space, we switched to using `memcpy()` without a null terminator to ensure the entire field could be used.

PRIMARY FUNCTIONS

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

- Assume that `acc->userid` should fit into the size `USER_ID_LENGTH` , or else the caller will get a warning via system log, and the `userid` value will be truncated. This seems reasonable and is documented to the caller.
- Uses helper functions as described above
- Explicitly sets all other values in the `account_t` struct to 0 for clarity and readability.
- Had issues with whether we needed helper functions or not. What they would return? ended up creating helper functions that return true on valid, and false on invalid fields. This also follows DRY, where these validating functions are reused throughout our code, especially when updating an account field.

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

- Assumes a user would not try to free a NULL account pointer, so will warn in a system log.
- Sets all data in `account_t` to 0 using a `libsodium` provided function `sodium_memzero()` that is guaranteed to not be optimised away. This is done to prevent sensitive user info left in the heap (for unauthorised eyes to see).
- This was changed from initial implementation using `memcpy()`

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

- Passes off responsibility to `libsodium` with a password comparator `crypto_pwhash_str_verify` . Assumes the `plaintext_password` is small enough to work with the library.
- When updating the password, we explicitly clear the old hash using `sodium_memzero()` to prevent any part of it from lingering in memory. This ensures that sensitive data doesn't remain in the `account_t` struct after it's no longer needed.

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

- If `acc->login_count` reaches `UINT_MAX`, we stop incrementing it to avoid unsigned integer overflow, which would wrap it back to zero. We chose not to log a warning in this case to avoid log spam from repeated login attempts — especially if used as part of a denial-of-service strategy.
- We use `time(NULL)` to capture the current time for login-related timestamps. This is standard in C, and we didn't encounter any issues with its behavior.
- We originally overlooked the risk of overflow in `login_count` and `login_fail_count`, but added an explicit check against `UINT_MAX` to prevent it.

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

- Compares `unban_time` and `expiration_time` against current time via `time(NULL)`.
- Skips check if fields are zero.

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

- We assume the caller understands what the `t` value represents and that it aligns with the expectations of the higher-level *OO* system. For example, we don't check for `t < 0`, since negative values might have a defined meaning elsewhere in the system. Our priority was to follow the project specification as written, which does not restrict the range of `t`.
- There was some uncertainty around what qualifies as a valid `t` value, but since the spec didn't provide any constraints, we chose not to make assumptions or enforce arbitrary limits.

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

- This function wraps the existing `validate_email(...)` helper, following the DRY principle by reusing the same logic used in `account_create()` to ensure consistency in validation.
- Its behavior closely mirrors the email handling in `account_create()`, so there were no additional edge cases or design considerations specific to this function.

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

- Assuming that all the data in the `account_t` are not properly NULL-terminated, so must be manually added. This allows us to properly store the full value of bytes as defined in `account.h`

without having to worry about storing the NULL-terminator. These "pseudo-strings" are thus converted into strings at the start of this function.

- `time` and `arpa/inet` libraries are concerned with safely converting our `account_t` fields into printable formats. Using safe library calls.
- Uses an out buffer `outbuf` with a max length of 512 to write the message to be written into. This size is larger than the largest possible message (with maxed out fields) can be to avoid truncation.
- No sensitive data is displayed, as if higher levels of the *OO* system can use this function, that sensitive data like passwords hashes and the like are not accessible (and thus not displayed).

`login.c` FUNCTIONS

HELPER FUNCTIONS

```
void safe_fd_message(int fd, const char *fmt, ...);
```

- Safely writes a formatted string to a file descriptor using a fixed-size buffer (buf[256]).
- Uses vsnprintf() to avoid format string vulnerabilities and buffer overflows, then writes the resulting string using write().
- The output is truncated if the formatted string exceeds the buffer size, but this prevents accidental memory corruption or leaking uninitialized memory.
- We chose not to add error handling for write() failures here, as the calling function is responsible for interpreting success or failure and responding appropriately.
- We avoid using `dprintf()` directly because it lacks bounds checking, which could lead to buffer overflows or format string vulnerabilities if misused.

PRIMARY FUNCTIONS

```
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);
```

- The login process checks user existence, then whether the account is expired, banned, or has exceeded failed login thresholds, and only then checks the password. This ordering prevents unnecessary computation and reduces the risk of leaking sensitive state through timing side-channels, aligning with STRIDE principles like *Elevation of Privilege* and *Information Disclosure*.
- Different login failure codes are returned to distinguish causes during auditing and client feedback. This supports layered architecture assumptions and improves debugging and logging clarity.

- A helper function `safe_fd_message()` is used for all formatted output to file descriptors. This avoids format string vulnerabilities and ensures writes are safely bounded and consistent.
- Before casting `acc->account_id` from 64-bit int to a 32-bit int, we check for a conversion overflow. This prevents session state corruption or TOCTOU-style attacks involving unsafe type conversions.
- Login success and failure counters are reset as specified in the project spec to avoid stale values affecting future authentication attempts.
- To securely output messages to clients, we developed a reusable bounded formatter helper that encapsulates the write logic and avoids dangerous patterns like `printf(fd, user_input)`.
- We ensured session integrity by only modifying `session_start`, `account_id`, and `expiration_time` if the login attempt succeeded, avoiding inconsistent or partially-updated session data.
- Since the project specification did not define a separate session expiry time, we chose to align it with the account's expiration_time. This allows higher-level *OO* system users to control session behavior by updating the account state. We considered this a reasonable and coherent design choice.