CITS3007 Group Project – Phase 1 Report

Group Name: *TOC/TOUah*
Group Number: 33
Date: 16 Apr 2025

Group Members

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

1. Team Communication & Responsibilities

> Describe how the group will communicate throughout the project: Frequency of meetings (e.g., weekly; fortnightly; as needed); Meeting format (e.g., face-to-face; online via video calls or chat platforms); Preferred communication tools (e.g., Discord, Slack, MS Teams, email)

Our team will hold `weekly face-to-face` meetings at UWA on Wednesday mornings. These will allow us to provide weekly status updates, issues, challenges, etc. As well as our weekly status updates, we will implement ad hoc emergency meetings through MS Teams as required when there are critical bugs and fixes required. Our preferred communication tool will be `Microsoft Teams`, where we will discuss the bulk of the project.

> Define how responsibilities will be allocated in phase 2. Who will be responsible for which tasks?

Tasks will be assigned based on each member's strengths and preferences to encourage motivation and completion. These sections will be divided based on the project description, which is divided into 4 sections. For example:

- **Cooper** - Team leader. In charge of submissions, coordination of group meetings, keeping track of the development of sections of work.
- **Elke** - In charge of `Player authentication`
- **Fin** - In charge of `Session management`
- **Marc** - In charge of `Role-based access control`
- **Mika** - In charge of `Admin and operations access`

As a group we will plan how to implement each task and comprehensively discuss the requirements before we start writing any code. Specifically, we will be

following an Agile framework, which focuses on iterative development, continuous feedback, and adaptive planning (https://agilemanifesto.org/). As some sections are dependent on the ones before them to be completed, we will coordinate between group members on how these requirements will be facilitated and jump between tasks/features as necessary - particularly the team leader.

> How will the group ensure accountability and track progress?

Each week, team members will be accountable for presenting *weekly status reports* in our team meetings. This is an opportunity for each member to give updates on their recent work, as well as highlight any problems encountered. These features and changes will be logged via the `Issues` tab, where other group members can contribute suggestions/improvements to the code.

---

2. Version Control Strategy

> Specify where the project's source code will be hosted (e.g., GitHub, GitLab, Bitbucket). How will you handle merging members' contributions?

We will use `GitHub` for version control, where we can track all changes with a comprehensive history of what changes were made, when they occurred, and who implemented them. We will handle merges by making sure each change is done via a `pull request`, which allows the new feature's code to be reviewed. We will require these `pull requests` to be reviewed by at least two other group members before it ca merged into the main codebase. This process provides a structured way to review, suggest improvements and discuss code changes, ensuring the code quality is maintained and providing a means of collaboration between group members.

> If you're familiar with version control branching strategies, will you adopt any particular strategy or workflow? (e.g., feature branches, main/dev workflow).

We will adopt a `main/dev` workflow, with feature specific branching within each dev branch as necessary.

- The `main` branch will contain production-ready code.
- The `dev` branch will be the integration branch where feature branches are merged after testing
- Each group member will create feature-specific branches based on assigned components (e.g. `feature/operation-access`). This structure helps reduce merge conflicts and allows us to work on the project simultaneously.

We chose *feature branching* within the `dev` branch as we believe it aligns better with the requirements of our clients needs, and means that developerss in the team can jump between features as necessary without interupting the workflow of the project.

Identify any version control policies (e.g., commit message conventions, review/approval process before merging).

All changes will undergo a peer review before merging. This can be done via a `pull request` that must be approved by at least two other group members before merging into `dev` or `main`.

We will also follow consistent commit message convention: - What was changed - Why it was changed - Any remaining issues/future improvements

Good commit messages are important for maintaining a clear and understandable project history, and will allow us (or other developers) to quickly see what changes were made and why.

---

3. Development Tools

List the common tools the group will use for implementation: Code editor or IDE (e.g., VS Code, JetBrains, Vim); Any additional tools for collaboration or efficiency (e.g., linters, debugging tools, CI/CD services). Explain why you made these choices.

Visual Studio Code (VSCode) will be the standardised code editor where we will create our C programs. We have selected **VSCode** due to its lightweight nature, cross-platform compatibility, and extensive extension ecosystem that supports C development, including syntax highlighting (via `IntelliSense`), integrated terminal, in-built `Git` integration, and debugging capabilities.

For compiling and debugging, we will use the `gcc` compiler (for type errors) along with the `gdb` debugger (for logic errors). By utilising the **GCC toolchain**, we will perform compiliation using a robust `Makefile` that integrates extensive compiler warning and Google's sanitisers (e.g. AddressSanitizer and UndefinedBehaviourSanitizer).

Compilation will using flags `-Wall`, `-Wextra`, `-Wshadow`, `-Wconversion`, to detect common sources of bugs early. Debugging is further enhanced using the flags `-ggdb3` and `fno-omit-frame-pointer` for meaningful stack traces.

We have chosen to adopt a strict compiler configuration (see below) during the development and testing phases to enforce secure coding practices. This configuration enables a comprehensive set of warning flags and debugging options that align with secure C programming principles taught in the lectures and labs.

```
# Compiler and tools
CC      := gcc
CFLAGS  := -std=c11 -pedantic -Wall -Wextra -Wshadow -Wconversion \
           -Wno-unused-result -Wno-missing-field-initializers \
           -Wformat=2 -Wswitch-default -Wswitch-enum -Wcast-align \
           -Wpointer-arith -Wbad-function-cast -Wstrict-overflow=5 \
```

```
            -Wstrict-prototypes -Winline -Wundef -Wnested-externs \
            -Wcast-qual -Wunreachable-code -Wlogical-op \
            -Wfloat-equal -Wstrict-aliasing=2 -Wredundant-decls \
            -Wold-style-definition -Werror \
            -ggdb3 -O0 -fno-omit-frame-pointer -fno-common -fstrict-aliasing \
            -fsanitize=address,undefined \
            -fno-sanitize-recover=all -lm -pthread

# Programs to build
PROGRAMS := {programA} {programB} {programC} {...}
OBJS     := $(PROGRAMS:%=%.o)

# Default target
all: $(PROGRAMS)

# Compile each source file
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

# Link each program
%: %.o
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

(more omitted)
```

This was adapted from https://stackoverflow.com/questions/154630/recommended-gcc-warning-options-for-c as recommended in Lab 6

Additionally, we will use **GDB (GNU Debugger)** for runtime debugging. GDB has been consistently used in CITS3007 labs, and all team members are familiar with its workflow. Its integration with VS Code allows for seamless breakpoints, step-through debugging, and variable inspection—crucial for diagnosing issues in low-level C code.

Code quality will be maintained via `clang-tidy`, which is a static analyser which is used to identify potential errors, style violations, and best practice violations. This static analyser will be integrated into the `VSCode` environment of all our group members, as it should be already integrated via the `C/C++ extension`.

---

4. Key Secure Coding Practices for Phase 2

> Identify three security-related tools or practices covered in the unit that will be most critical during phase 2. For each, explain: why it is relevant to the project; how it will be applied during development; and how the group will ensure it is effectively used.

1 - Static and Dynamic Code Analysers

The use of **static code analysers** (like `clang-tidy`) will be relevant to our project as they help detect vulnerabilities and bad practices before runtime. It will be applied regularly during development to review code for security issues.

The group will ensure effective use by making `clang-tidy` checks part of our coding workflow and reviewing any flagged warnings together.

2 - GCC Debugger

The **gcc debugger** will be relevant to the project as it is essential to have a comprehensive debugging tool to catch errors and potential mistakes which 'fall through the cracks' of standard compiling. It will be applied within our development process every time we compile any C code.

We will guarantee its effective use by ensuring all members use the same debugger and compiling specifications.

3 - Best Practice for Storing Passwords (Hash, Salt)

Storing passwords is relevant for protecting user accounts in *Oblivionare Online*. The best practice is utilising a strong hashing algorithm (such as `bcrypt`), with a **unique salt** for each password. A random salt will be generated, and added to a user's password before hashing, ensuring identical passwords will still result in different hashes (protecting from brute-force and rainbow table attacks).

The **Access Control System** is responsible for authenticating users, and must prevent the unauthorised access by 3rd parties. We will implement password handling via secure libraries that automatically perform salting and hashing. Our group will enforce this through code reviews (see *pull requests*), and performing password storage checks in testing phases.

---

5. Risk Management & Quality Assurance

> Outline potential risks to the project and how they will be mitigated. (You may wish to think about resourcing risks – e.g. member illness, service outage – as well as technical and operational risks.)

Group Member Illnesses/Service Outages

An important risk to the project are resourcing risks where we may have interuptions to our work or an increased workload due to member unavailability. To mitigate this, we will ensure our code is readable and every group member understands all aspects of the project so there is no tech debt.

Authentication Logic Errors

A critical risk is the incorrect implementation of authentication or session handling which could lead to unauthorised access. To mitigate this, we will design the login flow early and review all authentication related code during peer reviews in our Wednesday meetings.

Insecure Password Handling

Storing or transmitting passwords insecurely is a serious security risk. To mitigate this, we will use functions designed for password hashing (such as `bcrypt`) to automatically handle the salting and hashing of passwords.

Validating user input

Failing to validate user input can lead to serious vulnerabilities like **buffer overflow** and **injection attacks**. To mitigate this, we will sanitise all inputs to avoid unsafe C function such as `gets()`, opting instead for the safer alternatives like `fgets()`, which limits input size and reduces the risk of memory corruption.

Poor Privilege Handling

Improper privilege checks can expose sensitive admin functions to unauthorised users. To prevent this, we will implement a strict **Role-Based Access Control (RBAC)** and implement test cases to cover all access control scenarios, including edge cases and priv esc attempts.

Low C Security Familiarity

Not all team members may be comfortable with secure C programming practices. To mitigate this, we will share resources and get more experienced members to give guidance during development.

Tool Conflicts and Merge Errors

Version control conflicts or tool inconsistencies could disrupt progress. To mitigate this, we'll adopt a **feature-branch workflow**, standardise our build environment using a shared `Makefile`, and conduct merges through **pull requests**.

As an overarching mitigation strategy, we have developed a project timelime with enough buffer room to accomodate for unforseen delays while still meeting the deadline.

Weekly Schedule

Week	Meeting Time	Agenda	Tasks Completed Before Meeting
Week 7	9am Wednesday	- Plan report content	N/A
Week 8	9am Wednesday	- Edit and finalise report	Report Draft
Study Break	9am Wednesday	- Discuss details of phase 2; Plan approach to each component and how they will work together; Brainstorm potential issues/blind spots	- Read through phase 2; Begin brainstorming
Week 9	9am Wednesday	- Discuss any issues with individual assigned components; Assist each other; Plan and start phase 2 report	Started phase 2
Week 10	9am Wednesday	- Edit and finalise phase 2 report; Final checks of implementation; Begin planning reflection (phase 3)	Implementation and report
Week 11	9am Wednesday	- Finalise presentation; Finalise reflection; Practice demo/presentation	Presentation and reflection

Describe how code quality will be maintained: – Will the group follow a specific coding standard? – Will peer reviews, automated testing, or static analysis tools be used?

Code quality will be maintained by following the **CERT C Secure Coding Standard**, which provides guidelines for secure and reliable coding practices in C particularly involving what to do in the case of integer/buffer overflow and uninitialised variables. We will also aim to align our style with **Google's C Style Guide** to ensure consistency and readability of our group's codebase.

To maintain code integrity, **peer reviews** will be done manually from group members whenever changes are made to our codebase before merging into the main branch (using `GitHub` pull requests). This means the knowledge of functionality is shared in the group, and that the code quality is consistent throughout development.

We will also use **automated testing** by creating unit tests using `Unity`, a popular and lightweight framework for C development. Furthermore, **Clang's static analyser** will be implemented in our code as well. Both of these will detect potential bugs and security vulns, helping us maintain reliable and secure code.

6. Group Name

Group Name: TOC/TOUah