Contents

CITS3007 Secure Coding - Group Project Phase 1

Supplementary Documentation

1. Team Communication & Responsibilities

1.1 Detailed Communication Strategy

Meeting Schedule

- **Weekly**: Monday 10:00 at UWA (approx. 2 hours, includes Sprint Planning activities)
- **Standups**: Wednesday & Friday (Online via Discord, 5 min check-in), 18:00
- **Emergency**: Discord/Signal

Meeting Format

- **Primary**: Face-to-face sessions in Reid Library study rooms for regular meetings and complex design discussions
- **Secondary**: Video calls via Discord when in-person meetings are not possible
- **Emergency**: Signal group chat for urgent communications

Communication Tools

- **Discord**: Primary platform for team communication
  - Dedicated channels:
    - #general-discussion
    - #technical-issues
    - #code-reviews
    - #meeting-minutes
  - Voice channels for pair programming sessions
- **Signal**: Secure messaging for urgent communications and sensitive information
- **GitHub Projects**: Task tracking and project management
  - Kanban board for task visualization
  - Issue tracking integrated with code
  - Milestone tracking for deliverables
  - Automated project updates via commits and PRs

1.2 Progress Tracking & Accountability

- **Standup Updates**: Quick check-ins during online standups (Wed/Fri)

- **Weekly Reviews**: Progress assessment during weekly meetings
- **Burndown Charts**: Track velocity and remaining work
- **Peer Reviews**: Code review requirements before merging
- **Documentation Updates**: Regular wiki updates tracking decisions and progress

1.3 Meeting & Participation Expectations

Attendance Policy

- **Meeting Notification**
  - Members must confirm attendance 24 hours before scheduled meetings
  - Unavoidable absences must be communicated in a timely manner
  - Emergency absences should be notified as soon as possible via Signal
  - Additional steps may be taken if a group member is uncontactable for an extended period of time

No-Show Protocol

1. **First Instance**
   - Team lead will contact member via Signal
   - Member must provide catch-up plan within 24 hours
   - Missed work to be redistributed if critical path affected
2. **Second Instance**
   - Formal discussion in next team meeting
   - Written explanation required
   - Adjustment of responsibilities if needed
   - Development of catch-up plan with specific deadlines
3. **Persistent Issues**
   - Meeting with unit coordinator
   - Formal documentation of attendance issues
   - Potential reassignment of critical path tasks
   - Review of team member's role and responsibilities

Contribution Monitoring

- **Weekly Contribution Review**
  - Code commits and pull requests
  - Documentation updates
  - Meeting participation
  - Task completion rate
- **Intervention Triggers**
  - Missing two consecutive meetings without notice
  - No code commits for one week without explanation
  - Consistently late or incomplete deliverables
  - Non-responsive for > 24 hours during critical sprints

Support Measures

- Pair programming sessions for struggling members
- Flexible meeting times for legitimate scheduling conflicts
- Recording of important meetings for async review
- Regular 1-on-1 check-ins if performance issues arise

1.4 Detailed Responsibility Allocation

Core Responsibilities

1. **Technical Lead & Infrastructure** (Stephen Beaver)
    - Repository management and version control
    - CI/CD pipeline implementation and maintenance
    - Development environment setup and management
    - Infrastructure security and hardening
    - Technical architecture decisions
    - Code review coordination
    - Development standards enforcement
2. **Authentication & Security** (Kelly Snow)
    - Implementation of secure login mechanisms
    - Password management using libsodium
    - Account recovery procedures
    - Multi-factor authentication support
    - Security testing and validation
    - Security documentation
3. **RBAC & Financial Controls** (Prem Patel)
    - Design and implementation of role hierarchy
    - Privilege management system
    - Financial instrument access control
    - Transaction security
    - Economic model integration
    - Financial security validation
4. **Session Management & Performance** (Muhammad Qureshi)
    - Secure session handling and validation
    - Session timeout management
    - Concurrent session control
    - Performance optimization
    - Resource management
    - Load testing and monitoring
5. **Testing & Quality Assurance** (Kai Fletcher)
    - Test framework implementation
    - Unit and integration testing
    - Security testing automation
    - Quality metrics tracking
    - Documentation standards
    - Code coverage analysis

Shared Responsibilities

- Code reviews
- Security awareness
- Documentation updates
- Sprint planning
- Knowledge sharing
- Flexible roles, opportunity to work on interest areas

Cross-Training Strategy

- Regular knowledge sharing sessions
- Pair programming on critical components
- Documentation of all processes

- Peer review of critical components

2. Version Control Strategy

2.1 Detailed Repository Structure

```
oo-acs/
├── src/
│   ├── auth/      # Authentication system
│   ├── rbac/      # Role-based access control
│   ├── session/   # Session management
│   └── admin/     # Administrative tools
├── tests/         # Test suite
├── docs/          # Documentation
└── tools/         # Development tools
```

2.2 Branching Strategy

- `main`: Production-ready code
- `develop`: Integration branch
- `feature/feature-name`: New features
- `bugfix/bug-name`: Bug fixes
- `security/security-change`: Security-related changes

2.3 Security Measures

- GitHub commit signing with GPG keys
- Branch protection rules
- Required pull request (PR) reviews
- Automated security checks

3. Development Tools

3.1 Development Environment

- **Linode Instances**:
  - Testing: Linode 1GB (Nanode)
  - Staging: Linode 2GB (Standard)
  - Production: Linode 2GB (Standard)
- **Ansible Configuration**:
  - Infrastructure as Code
  - Automated provisioning
  - Security hardening
  - Deployment automation
  - Playbook Structure:
    ```
    ansible/
    ├── inventory/
    │   ├── testing
    │   ├── staging
    │   └── production
    ```

```
├── group_vars/
│   ├── all.yml
│   ├── testing.yml
│   ├── staging.yml
│   └── production.yml
├── roles/
│   ├── common/
│   ├── security/
│   ├── application/
│   └── monitoring/
└── playbooks/
    ├── site.yml
    ├── testing.yml
    ├── staging.yml
    └── production.yml
```

- GitHub Integration:

```yaml
name: Deploy to Environment
on:
  push:
    branches: [ main ]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Deploy to Testing
        if: github.ref == 'refs/heads/main'
        run: |
          ansible-playbook -i inventory/testing playbooks/testing.yml
```

- GitHub Secrets:
  * LINODE_API_TOKEN
  * ANSIBLE_VAULT_PASSWORD
  * SSH_PRIVATE_KEY
- **Repository Access (Deploy Keys)**:
  * To allow automated cloning of the private repository by Ansible, a unique SSH key pair (Ed25519) is generated on each target server for the deployment user (`oo-acs`).
  * The public key must be manually added as a **Deploy Key** (read-only recommended) to the GitHub repository settings.
  * The Ansible playbook includes tasks to generate the key if needed and pause, displaying the public key to be added to GitHub.
  * This ensures the server uses a dedicated key with limited permissions, adhering to the principle of least privilege, rather than using personal SSH keys.

3.2 Development Standards

- **Coding Standard**: SEI CERT C Coding Standard
- **Compiler Flags**:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -std=c11 -pedantic -Wall -Wextra -Werror -Wformat=
```

```
set(CMAKE_C_FLAGS_DEBUG "${CMAKE_C_FLAGS_DEBUG} -fsanitize=address,undefined")
```

- **Coding Style Rules**:
  - Indentation: 4 spaces (no tabs)
  - Line length: 80 characters max
  - Function names: snake_case
  - Constants: UPPER_CASE
  - Struct names: PascalCase
  - Local variables: snake_case
  - Comments: Doxygen style
  - Braces: K&R style

3.3 Security Standards

1. **CIA Triad Implementation**
   - Confidentiality:
     - libsodium encryption for sensitive data
     - Basic access control implementation
     - Secure session management
   - Integrity:
     - Input validation for all user inputs
     - Basic error checking mechanisms
     - Secure logging practices
   - Availability:
     - Simple error handling
     - Basic logging system
     - Resource management
2. **C11 Standard Compliance**
   - Strict adherence to ISO/IEC 9899:2011 (C11)
   - Compiler flags:
     ```
     set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -std=c11 -pedantic -Wall -Wextra -Werror")
     ```
   - No compiler-specific extensions unless explicitly required
   - Standard library usage:
     - stdint.h for fixed-width integers
     - stdbool.h for boolean types
     - stdatomic.h for atomic operations
     - threads.h for threading support
3. **SEI CERT C Coding Standard**
   - Key rules implemented:
     - INT30-C: Ensure operations on unsigned integers cannot wrap
     - STR31-C: Guarantee null termination for string operations
     - MEM35-C: Allocate sufficient memory for an object
     - ERR33-C: Detect and handle errors
   - Memory safety rules:
     - MEM00-C: Allocate and free memory in the same module
     - MEM01-C: Store a new value in pointers immediately after free
     - MEM02-C: Immediately cast the result of a memory allocation function call
   - String handling rules:
     - STR00-C: Represent characters using an appropriate type
     - STR02-C: Sanitize data passed to complex subsystems
     - STR03-C: Do not inadvertently truncate a string
4. **MISRA C:2012 Guidelines**
   - Mandatory rules compliance required
   - Advisory rules evaluated per case

- Deviations must be documented and reviewed
- Key guidelines:
  - Rule 8.2: Function types shall be in prototype form
  - Rule 8.4: Identifiers shall be distinct
  - Rule 8.7: Objects shall be defined at block scope
  - Rule 8.12: When an array is declared, its size shall be explicitly specified

5. **ISM Guidelines**
   - ISM-1759: Secure development lifecycle
   - ISM-1760: Secure coding practices
   - ISM-1761: Security testing
   - ISM-1762: Vulnerability management
   - Implementation:
     - Development process documentation
     - Security testing procedures
     - Vulnerability management plan
     - Secure coding guidelines
6. **OWASP Security Guidelines**
   - Input Validation:
     - Validate all user inputs
     - Sanitize data before processing
     - Use parameterized queries
   - Authentication:
     - Implement secure password policies
     - Use strong session management
     - Implement proper access control
   - Error Handling:
     - Implement proper error handling
     - Avoid information disclosure
     - Log security events
7. **STRIDE Threat Modeling**
   - Spoofing:
     - Strong authentication
     - Session management
     - Access control
   - Tampering:
     - Input validation
     - Data integrity checks
     - Secure storage
   - Repudiation:
     - Audit logging
     - Transaction tracking
     - User accountability
   - Information Disclosure:
     - Data encryption
     - Access control
     - Secure communication
   - Denial of Service:
     - Resource management
     - Rate limiting
     - Error handling
   - Elevation of Privilege:
     - Principle of least privilege
     - Role-based access control
     - Permission verification

3.4 Security Tools

1. **Static Analysis**
   - gcc -fanalyzer
     - Enable all analyzers: -fanalyzer -fanalyzer-call-summaries
     - Enable verbose output: -fanalyzer-verbose-edges
   - Valgrind
     - Full memory checking: –leak-check=full
     - Show reachable blocks: –show-reachable=yes
     - Track origins: –track-origins=yes
   - AFL++ for fuzzing
     - Timeout: 1000ms
     - Memory limit: 50MB
     - Dictionary: custom.dict
2. **Testing Framework**
   - Unity testing framework
   - Test coverage requirements:
     - Critical components: 80% minimum
     - Non-critical components: 60% minimum
     - Security-critical paths: 100%
   - Test Structure:

```c
#include "unity.h"
#include <valgrind/memcheck.h>

void setUp(void) {
    // Setup code
}

void tearDown(void) {
    // Cleanup code
}

void test_memory_safety(void) {
    // Memory safety tests
    TEST_ASSERT_EQUAL(0, VALGRIND_COUNT_LEAKS());
}

void test_input_validation(void) {
    // Input validation tests
    TEST_ASSERT_EQUAL(0, validate_input("valid input"));
    TEST_ASSERT_EQUAL(-1, validate_input("invalid input"));
}

int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_memory_safety);
    RUN_TEST(test_input_validation);
    return UNITY_END();
}
```

3. **GitHub Security Features**
   - Commit Signing:

```
# Generate GPG key
gpg --full-generate-key
# Export public key
```

```
gpg --armor --export <key-id> > public-key.asc
# Configure Git
git config --global user.signingkey <key-id>
git config --global commit.gpgsign true
```

- Pre-commit Hooks:

```
#!/bin/sh
# Run static analysis
gcc -fanalyzer -c $1
# Check memory safety
valgrind --leak-check=full ./$1
# Verify documentation
doxygen -g
```

- Pull Request Template:

```
## Description
[Description of changes]

## Security Checklist
- [ ] Memory safety verified
- [ ] Input validation complete
- [ ] Error handling implemented
- [ ] Documentation updated
- [ ] Tests added/updated

## Review Checklist
- [ ] Code follows style guide
- [ ] Security requirements met
- [ ] Tests pass
- [ ] Documentation complete
```

4. **Dependency Management**
   - GitHub Dependabot Integration:

```
version: 2
updates:
  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "weekly"
  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "weekly"
```

   - CMake Dependencies:

```
# Core Dependencies
find_package(OpenSSL 3.0.0 REQUIRED)
find_package(Valgrind 3.18.1 REQUIRED)
```

5. **Cryptographic Tools**
   - Password Hashing: Argon2id via libsodium
     - Implementation: crypto_pwhash
     - Parameters:
       * Memory cost: 64MB
       * Time cost: 3
       * Parallelism: 4
       * Salt length: 16 bytes
       * Hash length: 32 bytes
     - Justification:

        * Winner of Password Hashing Competition
        * Memory-hard function resistant to GPU/ASIC attacks
        * Configurable parameters for security/performance balance
        * Built-in salt generation and storage
6. **Security Programs**
    - Basic Security Testing:
        - Automated security scanning:
            * Static analysis: gcc -fanalyzer
            * Memory safety: Valgrind
            * Basic fuzzing: AFL++
        - Input validation testing:
            * Buffer overflow tests
            * Integer overflow tests
            * Format string tests
        - Boundary condition testing:
            * Array bounds
            * String lengths
            * Integer ranges
        - Memory leak detection
    - Security Documentation:
        - Document known vulnerabilities
        - Track security updates
        - Note custom security patches
        - Maintain security checklist
    - Basic Security Review Process:
        - Code review checklist
        - Security-focused testing
        - Regular security audits
7. **CISA Guidelines Implementation**
    - Secure Development Lifecycle:
        - Development Environment:
            * Linode instances
            * Ansible automation
            * GitHub security practices
        - Code Quality:
            * SEI CERT C standard
            * Unity testing
            * Secure coding practices
        - Security Testing:
            * Static analysis
            * Memory safety
            * Input validation
    - Deployment Security:
        - Environment Separation
        - Access Control
        - Monitoring
    - Documentation Requirements:
        - Security Documentation
        - Development Documentation
        - User Documentation
8. **Supply Chain Security**
    - Simple Dependency Documentation:

```
## Dependencies
- OpenSSL 3.0.0: Cryptographic functions
```

```
- Valgrind 3.18.1: Memory analysis
- Unity: Unit testing framework
- libsodium: Password hashing
```

- Basic Library Management:
  - System package manager for core dependencies
  - Document custom modifications
  - Track security updates
- Simple Version Control:
  - Document library versions in CMakeLists.txt
  - Track security updates
  - Note any custom security patches

9. **Security Metrics**
   - Code Quality:
     - Cyclomatic complexity: < 10
     - Function length: < 50 lines
     - Comment density: > 20%
     - Duplication: < 5%
   - Security Metrics:
     - Static analysis warnings: 0
     - Memory leaks: 0
     - Buffer overflows: 0
     - Race conditions: 0
   - Testing Metrics:
     - Unit test coverage: > 80%
     - Security test coverage: > 95%
     - Performance test pass rate: 100%
10. **Session Management**
    - Token Generation:

```c
typedef struct {
    char token[TOKEN_LENGTH];
    time_t issued_at;
    time_t expires_at;
    char session_id[SESSION_ID_LENGTH];
} session_token_t;
```

    - Security Measures:
      - Token rotation every 15 minutes
      - Concurrent session limit: 3
      - Session timeout: 30 minutes
      - IP-based session validation
11. **Authentication System**
    - Password Policies:
      - Minimum length: 12 characters
      - Required character types: 3
      - Maximum age: 90 days
      - History: 5 previous passwords
    - Account Recovery:
      - Security questions: 3 required
      - Recovery email verification
      - Temporary password expiration: 24 hours
      - Failed attempts limit: 5

3.4 Collaboration Tools

- **Discord Configuration**:

    - Dedicated channels:
        * #general-discussion
        * #technical-issues
        * #code-reviews
        * #meeting-minutes
    - Voice channels for pair programming sessions
    - Meeting scheduling and reminders
    - File sharing and code snippets
- **GitHub Projects Setup**:
    - Kanban board configuration
    - Issue tracking templates
    - Milestone tracking
    - Automated project updates
    - Integration with CI/CD pipeline
- **Signal Integration**:
    - Emergency communication protocol
    - Secure messaging for sensitive information
    - Quick response system for critical issues

4. Key Secure Coding Practices

4.1 Memory Safety

- Safe memory management patterns from reference codebases:

```
// Memory management (from curl)
void *safe_malloc(size_t size) {
    void *ptr = malloc(size);
    if (!ptr) {
        log_error("Memory allocation failed");
        return NULL;
    }
    return ptr;
}
```

- Bounds checking for all array operations
- Memory leak detection
- Buffer overflow prevention

4.2 Input Validation

- Comprehensive input validation
- Boundary condition testing:
    - Buffer overflow tests: ±1 byte boundaries
    - Integer overflow tests: INT_MAX, INT_MIN
    - Format string tests: %n, %s, %x
- Format string validation
- Sanitization of user inputs

4.3 Access Control

- RBAC implementation based on FreeRADIUS patterns:

```
typedef struct {
    role_t role;
    permission_t *permissions;
    size_t num_permissions;
} user_roles_t;
```

- Principle of least privilege
- Permission verification
- Audit logging

4.4 Error Handling

- Consistent error handling patterns:

```
// Error handling (from OpenSSH)
int handle_error(int error_code) {
    switch (error_code) {
        case AUTH_ERROR:
            log_auth_error();
            return -1;
        case SESSION_ERROR:
            log_session_error();
            return -1;
        default:
            log_generic_error();
            return -1;
    }
}
```

- Proper error reporting
- Secure logging
- Recovery procedures

5. Risk Management & Quality Assurance

5.1 Risk Management

1. **Technical Risks**
   - Memory safety issues
   - Input validation vulnerabilities
   - Access control weaknesses
   - Performance bottlenecks
2. **Operational Risks**
   - Development environment issues
   - Team collaboration challenges
   - Time management
   - Resource constraints

5.2 Quality Assurance

1. **Code Quality**
   - SEI CERT C compliance
   - Static analysis

    - Code reviews
    - Documentation standards
2. **Security Testing**
    - Unit testing with Unity
    - Memory safety checks
    - Input validation testing
    - Access control verification
3. **Documentation**
    - Code documentation
    - Security documentation
    - User documentation
    - Process documentation

5.3 Implementation Timeline

Phase 1 (Planning - Completed)

- **Objective:** Establish a clear plan for collaboration, tooling, and quality for the project.
- **Key Outcomes Achieved:**
    - Team Communication plan and Responsibilities defined (Sec 1).
    - Version Control Strategy established using Git/GitHub, including branching and security measures (Sec 2).
    - Development Tools selected, including Linode instances and Ansible for automation (Sec 3.1).
    - Development Environment (Testing) provisioned and configured via Ansible, including security hardening and tool installation.
    - Secure Repository Access strategy (Deploy Keys) implemented in Ansible.
    - Development and Security Standards defined (SEI CERT C, C11, MISRA C, etc.) (Sec 3.2, 3.3).
    - Key Secure Coding Practices identified for Phase 2 (Sec 4).
    - Initial Risk Management and Quality Assurance plan outlined (Sec 5.1, 5.2).
    - Git-based tracking procedures defined (Sec 6).

Phase 2 (Implementation - Upcoming)

- **Duration:** Approx. 3 Weeks (Week 9 - Week 11)
- **Objective:** Implement the core ACS functionality based on Phase 1 planning and standards.
- **Week 9: Core Functionality & Initial Security**
    - Implement Authentication system basics (libsodium hashing).
    - Implement RBAC core structure and basic privilege checks.
    - Implement Session management basics (token generation/validation).
    - Begin implementing Memory safety practices and Input validation system.
    - Set up basic Admin tools framework.
    - Start core unit tests for implemented features.
    - Begin code documentation.
- **Week 10: Advanced Features & Core Testing**
    - Refine Access control mechanisms.
    - Implement secure Logging for security events.
    - Develop further Unit tests, aiming for initial coverage targets.
    - Begin security testing procedures (static analysis, manual checks).
    - Continue code and process documentation.
- **Week 11: Comprehensive Testing & Finalization**
    - Complete comprehensive Testing (Security, Performance, Integration).

  - Finalize all Documentation (Code, Security, User, Process).
  - Address findings from testing and perform necessary refactoring.
  - Prepare Phase 2 report and Phase 3 demo/presentation materials.

Phase 3 (Demo/Presentation - Weeks 11-12)

- Deliver final project demonstration and presentation.
- Reflect on Phase 1 plans vs. actual execution.

Buffer Time and Contingency Planning

- **Time Allocation**
  - Contingency time factored into task estimates where possible.
  - Aim to maintain a buffer of ~1-2 days for critical path tasks.
  - Flexible scheduling for unexpected delays.
- **Progress Monitoring**
  - Online Standups (Wed/Fri) for quick updates.
  - Weekly progress reviews and planning during Monday meetings.
  - Regular security audits.
- **Task Prioritization**
  - Critical path tasks identified
  - Dependencies mapped
  - Risk assessment for each task
  - Contingency plans for high-risk items
- **Resource Management**
  - Team member availability tracking
  - Skill development planning
  - Cross-training sessions
  - Knowledge sharing meetings

6. Git-Based Tracking

6.1 Progress Tracking

- **Regular**: Commit-based tracking with conventional commits.
- **Standups (Wed/Fri)**: Quick check on GitHub Projects board status.
- **Weekly (Mon)**: In-depth PR reviews and GitHub Projects updates/planning.

6.2 Effort Tracking

- **Metrics**:
  - Commit frequency and quality
  - PR participation
  - Code review activity
  - Documentation updates
- **Triggers**:
  - No commits for 3 days (indicates potential blocker).
  - Unreviewed PRs > 24 hours (ensures timely reviews).
  - Failed CI/CD checks.
  - Significant documentation gaps identified.

6.3 Quality Control

- **Automated**:

    - Pre-commit hooks
    - Static analysis
    - Test coverage
    - Documentation validation
- **Manual**:
    - Code reviews
    - Security reviews
    - Documentation reviews
    - Weekly retrospectives/reviews (part of Monday meeting).

7. Reference Documentation

7.1 Security Standards

- ISM Guidelines
- CISA Publications
- NIST Guidelines
- OWASP Standards

7.2 Development Standards

- SEI CERT C
- MISRA C
- C11 Standard
- Secure Coding Practices

7.3 Testing Standards

- Unit Testing
- Security Testing
- Performance Testing
- Documentation Standards

7.4 Reference Codebases

- curl: Memory management patterns
- OpenSSH: Authentication and session management
- FreeRADIUS: RBAC implementation

8. Project Rules and Assessment

8.1 Project Rules and Deadlines

- **Project Weighting**: 30% of final mark
- **Total Marks**: 60 marks
- **Phase 1 Due Date**: Wednesday 16 April 2025 (11:59 pm)
- **Phase 1 Marks**: 10 marks
- **Phase 2 Due Date**: Week 11 (40 marks)
- **Phase 3 Due Date**: Weeks 11-12 (10 marks)
- **Peer Evaluation**: Required for Phases 2 and 3
    - Group Contribution Factor (GCF) applied
    - Range: 0 to 1.2
    - Cannot exceed 100% for phases 2 and 3

8.2 Academic Conduct

- **University Policy**: Adherence to University Academic Integrity Policy
- **Collaboration Guidelines**:
  - Discussion of general principles permitted
  - All submitted work must be original
  - Clear documentation of external references
  - Proper attribution of all sources
- **Documentation Requirements**:
  - Maintain clear records of all contributions
  - Document all external resources used
  - Track all collaborative decisions

8.3 Report Format

- **Format**: Markdown (preferred)
  - Highly readable in raw form
  - Easily convertible to other formats
  - Excellent version control support
  - Consistent formatting across platforms
- **Submission**: Via Moodle
  - One submission per group
  - Include all group member names and numbers
  - Include group name and number

8.4 Assessment Criteria

- **Content Requirements (5 marks)**:
  - Team communication and responsibilities
  - Version control strategy
  - Development tools
  - Key secure coding practices
  - Risk management and quality assurance
- **Report Quality (5 marks)**:
  - Clarity of presentation
  - Logical organization
  - Strength of justifications
  - Technical accuracy
  - Professional presentation

1. Document Conversion Standards

1.1 Conversion Process

- **Tool**: Pandoc with xelatex engine
- **Command**:

```
pandoc input.md -o output.pdf \
  --pdf-engine=xelatex \
  -V geometry:"margin=1in" \
  -V colorlinks=true \
  -V linkcolor=blue \
  -V toccolor=blue \
  --toc \
```

```
--toc-depth=2 \
-V mainfont="Fira Sans" \
-V monofont="Fira Code"
```

1.2 Typography Standards

- **Main Text**: Fira Sans
  - Sans-serif font optimized for readability
  - Excellent Unicode support
  - Consistent rendering across platforms
- **Code Blocks**: Fira Code
  - Monospace font with programming ligatures
  - Enhanced readability for code snippets
  - Special character support for technical documentation

1.3 Layout Standards

- **Margins**: 1 inch (2.54cm) on all sides
- **Links**: Colored blue for better visibility
- **Table of Contents**:
  - Depth: 2 levels
  - Automatic generation
  - Clickable navigation
- **Code Blocks**:
  - Syntax highlighting
  - Monospace font
  - Preserved indentation
  - Line number support

1.4 Markdown Standards

- **Headings**: ATX-style (# for headings)
- **Lists**:
  - Unordered: * for bullet points
  - Ordered: 1. for numbered lists
- **Code**:
  - Inline: `backticks`
  - Blocks: “‘ with language specification
- **Links**: text format
- **Images**: alt format

1.5 Quality Control

- **Pre-conversion Checks**:
  - Valid Markdown syntax
  - Proper heading hierarchy
  - Consistent list formatting
  - Valid link references
- **Post-conversion Verification**:
  - Table of contents accuracy
  - Link functionality
  - Code block formatting
  - Font rendering
  - Page breaks