Final Project

Secure Cryptographic Application

Course: MAT364 - Cryptography

Instructor: Adil Akhmetov

University: SDU

Points: 20 points

Team Size: 3 students per group

Press Space for next page →

Project Overview

Objective

Design and implement a **secure cryptographic application** that demonstrates practical understanding of multiple cryptographic concepts covered in this course.

Project Goals

- Apply cryptographic algorithms in a real-world scenario
- Integrate multiple cryptographic primitives
- Demonstrate secure programming practices
- Create a functional, user-friendly application

Key Requirements

- **Working code** with proper documentation
- **Multiple cryptographic techniques** integrated
- **Security analysis** and threat modeling
- **Presentation** demonstrating the system
- **GitHub repository** with clean commit history

Project Options

Option 1: Secure Messaging Application

Core Features

- **End-to-end encryption** for messages
- **User authentication** with secure key management
- **Digital signatures** for message integrity
- **Forward secrecy** (optional bonus)
- **Group messaging** support (optional bonus)

Technical Requirements

- Implement **AES-256** for message encryption
- Use **RSA** or **ECDH** for key exchange
- Implement **HMAC** or **digital signatures** for authentication
- Secure password hashing with **bcrypt/Argon2**
- **TLS/HTTPS** for transport security (if web-based)

Implementation Suggestions

```python
# Example structure
class SecureMessaging:
    def __init__(self):
        self.rsa_key = RSA.generate(2048)
        self.aes_key = None

    def encrypt_message(self, message, recipient_pubkey):
        # Generate ephemeral AES key
        # Encrypt message with AES
        # Encrypt AES key with RSA
        # Sign with digital signature
        pass
```

Deliverables

- Command-line or web-based interface
- Key exchange protocol implementation
- Message encryption/decryption system
- User authentication system

Option 2: Secure File Encryption System

Core Features

- **File encryption/decryption** with multiple algorithms
- **Key derivation** from passwords (PBKDF2/Argon2)
- **Digital signatures** for file integrity verification
- **Key management** system
- **Metadata protection** (file names, sizes)

Technical Requirements

- Support **AES-256** in CBC/GCM mode
- Implement **RSA** for key encryption
- **SHA-256** for file hashing and integrity
- **Digital signatures** (RSA or ECDSA)
- Secure **key storage** mechanism

Implementation Suggestions

```
class SecureFileEncryption:
    def encrypt_file(self, filepath, password):
        # Derive key from password
        key = PBKDF2(password, salt, iterations=100000)

        # Generate file encryption key
        file_key = os.urandom(32)

        # Encrypt file with AES
        encrypted = AES_GCM_encrypt(file_key, file_data)

        # Encrypt file key with password-derived key
        encrypted_key = AES_encrypt(key, file_key)
```

Deliverables

- File encryption/decryption tool
- Key management interface
- Integrity verification system
- Password-based key derivation

Option 3: Secure Authentication System

Core Features

- **Multi-factor authentication** (password + TOTP)
- **JWT token** generation and validation
- **Session management** with secure cookies
- **Password reset** with secure tokens
- **Account recovery** mechanisms

Technical Requirements

- **bcrypt/Argon2** for password hashing
- **HMAC-SHA256** for JWT signing
- **TOTP** (Time-based One-Time Password) implementation
- **RSA** or **ECDSA** for token signing
- **Secure session** storage

Implementation Suggestions

```python
class SecureAuthSystem:
    def register_user(self, username, password):
        # Hash password with bcrypt
        password_hash = bcrypt.hashpw(password, salt)

        # Generate TOTP secret
        totp_secret = pyotp.random_base32()

        # Store user credentials securely
        pass

    def login(self, username, password, totp_code):
        # Verify password
```

Deliverables

- User registration/login system
- JWT token management
- TOTP implementation
- Session management

Option 4: Blockchain-Based Voting System

Core Features

- **Cryptographic voting** with privacy preservation
- **Blockchain** for vote immutability
- **Digital signatures** for voter authentication
- **Zero-knowledge proofs** (optional bonus)
- **Vote verification** without revealing choices

Technical Requirements

- **SHA-256** for blockchain hashing
- **ECDSA** or **RSA** for digital signatures
- **Merkle trees** for vote aggregation
- **Public key infrastructure** for voter identity
- **Consensus mechanism** (Proof of Work/Stake)

Implementation Suggestions

```
class VotingBlockchain:
    def __init__(self):
        self.chain = []
        self.pending_votes = []

    def create_vote(self, voter_id, choice, private_key):
        # Create vote transaction
        vote_data = {
            'voter_id': voter_id,
            'choice': encrypt_choice(choice),  # Encrypted choi
            'timestamp': time.time()
        }
```

Deliverables

- Blockchain implementation
- Vote encryption system
- Digital signature verification
- Vote counting mechanism

Technical Requirements

Mandatory Cryptographic Components

Required Elements (All Projects)

- ✅ **Symmetric encryption** (AES-128/256)
- ✅ **Asymmetric encryption** (RSA or ECC)
- ✅ **Hash functions** (SHA-256 or SHA-3)
- ✅ **Digital signatures** (RSA or ECDSA)
- ✅ **Key exchange protocol** (Diffie-Hellman or RSA)
- ✅ **Password hashing** (bcrypt, Argon2, or PBKDF2)

Additional Requirements

- ✅ **Secure random number generation**
- ✅ **Proper key management**
- ✅ **Error handling** for cryptographic operations
- ✅ **Input validation** and sanitization
- ✅ **Documentation** of security assumptions
- ✅ **Threat model** analysis

Note: You may use cryptographic libraries (e.g., `cryptography` for Python, `crypto` for Node.js), but you must demonstrate understanding of the underlying concepts and implement at least one cryptographic primitive from scratch.

Programming Languages & Libraries

Recommended Languages

- **Python 3.8+** (recommended)
  - `cryptography` library
  - `pycryptodome` library
  - `bcrypt` for password hashing
- **JavaScript/Node.js**
  - `crypto` (built-in)
  - `bcrypt` or `argon2`
  - `jsonwebtoken` for JWT
- **Java**
  - `javax.crypto` package
  - Bouncy Castle library

Allowed Libraries

✅ **Cryptographic libraries** (cryptography, pycryptodome, etc.)

✅ **Web frameworks** (Flask, Express, FastAPI, etc.)

✅ **Database libraries** (SQLite, PostgreSQL, etc.)

✅ **Testing frameworks** (pytest, jest, etc.)

❌ **Pre-built cryptographic applications**

❌ **Copying code without attribution**

Important: While you can use libraries, you must understand and explain how each cryptographic component works. Include comments and documentation explaining the cryptographic operations.

Deliverables & Submission

Required Deliverables

1. GitHub Repository

- **Public repository** with all source code
- **Clean commit history** showing development process
- **README.md** with:
  - Project description
  - Installation instructions
  - Usage examples
  - Team member contributions
- **Code documentation** (comments, docstrings)
- **License** file (MIT, Apache, etc.)

2. Source Code

- **Working implementation** of chosen project
- **Well-structured** and modular code
- **Error handling** and input validation
- **Unit tests** (at least basic test coverage)
- **Configuration files** (requirements.txt, package.json, etc.)

3. Documentation

- **Architecture document** explaining system design
- **Security analysis** document:
  - Threat model
  - Security assumptions
  - Potential vulnerabilities
  - Mitigation strategies
- **User manual** (if applicable)
- **API documentation** (if applicable)

4. Presentation

- **10-15 minute** presentation
- **Live demonstration** of the application
- **Slides** covering:
  - Problem statement
  - Architecture overview
  - Cryptographic components used
  - Security analysis
  - Demo
  - Challenges faced
  - Future improvements

GitHub Repository Structure

Recommended Structure

```
project-name/
├── README.md              # Project overview
├── LICENSE                # License file
├── requirements.txt       # Python dependencies
├── .gitignore            # Git ignore file
├── src/                   # Source code
│   ├── main.py           # Entry point
│   ├── crypto/           # Cryptographic modules
│   ├── utils/            # Utility functions
│   └── tests/            # Unit tests
├── docs/                  # Documentation
│   ├── architecture.md   # System design
│   ├── security.md       # Security analysis
```

README.md Template

```
# Project Name

## Description
Brief description of your project

## Features
- Feature 1
- Feature 2
- Feature 3

## Installation
    pip install -r requirements.txt
```

Presentation Requirements

Presentation Structure

1. **Introduction** (1-2 min)
   - Team members
   - Project overview
   - Problem statement
2. **Architecture** (3-4 min)
   - System design
   - Component overview
   - Data flow diagrams
3. **Cryptographic Components** (4-5 min)
   - Algorithms used
   - Implementation details
   - Security considerations
4. **Live Demo** (3-4 min)
   - Working application
   - Key features demonstration
   - Security features
5. **Conclusion** (1-2 min)
   - Challenges faced
   - Lessons learned

Presentation Tips

- ✅ **Practice** your demo beforehand
- ✅ **Prepare** backup slides/videos if demo fails
- ✅ **Explain** cryptographic concepts clearly
- ✅ **Show** code snippets of key implementations
- ✅ **Discuss** security trade-offs
- ✅ **Answer** questions confidently

Evaluation Criteria

- **Technical implementation** (40%)
- **Cryptographic correctness** (30%)
- **Code quality** (15%)
- **Presentation** (15%)

Evaluation Criteria

Grading Rubric (20 points)

Code Implementation (8 points)

- **Functionality** (3 pts): Application works as intended
- **Code Quality** (2 pts): Clean, readable, well-structured
- **Documentation** (2 pts): Comments, docstrings, README
- **Testing** (1 pt): Unit tests or manual testing evidence

Cryptographic Implementation (7 points)

- **Correct Usage** (3 pts): Algorithms used correctly
- **Security** (2 pts): Proper key management, secure practices
- **Completeness** (2 pts): All required components implemented

Documentation (3 points)

- **Architecture** (1 pt): Clear system design explanation
- **Security Analysis** (1.5 pts): Threat model, vulnerabilities
- **User Guide** (0.5 pt): Usage instructions

Presentation (2 points)

- **Clarity** (1 pt): Clear explanation of project
- **Demo** (1 pt): Working demonstration

Important: Projects that copy code without understanding, have security vulnerabilities, or don't meet minimum requirements will receive significant point deductions or may fail.

Common Mistakes to Avoid

Security Issues

- ❌ **Hardcoded keys** or passwords
- ❌ **Weak random number generation**
- ❌ **Improper key management**
- ❌ **No input validation**
- ❌ **Insecure password storage**
- ❌ **Missing error handling**

Code Quality Issues

- ❌ **No documentation** or comments
- ❌ **Poor code organization**
- ❌ **No version control** or messy commits
- ❌ **Copy-pasted code** without attribution
- ❌ **No testing** or error handling

Presentation Issues

- ❌ **No live demo** or broken demo
- ❌ **Cannot explain** cryptographic concepts
- ❌ **Missing security analysis**
- ❌ **Unclear architecture** explanation
- ❌ **Poor time management**

Best Practices

- ✅ **Use secure defaults** (AES-256, RSA-2048+)
- ✅ **Implement proper key derivation**
- ✅ **Validate all inputs**
- ✅ **Use established libraries** correctly
- ✅ **Document security assumptions**
- ✅ **Test thoroughly**

Getting Started

Step 1: Form Your Team

- Find 2 other students
- Discuss project interests
- Assign roles (e.g., backend, frontend, crypto)

Step 2: Choose Your Project

- Review all 4 options
- Consider your team's strengths
- Select the most interesting option

Step 3: Submit Proposal

- Write 1-page project proposal
- Include: project choice, team members, basic architecture
- Submit via [method TBD]

Step 4: Set Up Repository

```
# Create GitHub repository
git init
git remote add origin https://github.com/username/project-name.

# Create initial structure
mkdir -p src/crypto src/utils docs tests
touch README.md requirements.txt .gitignore

# Make initial commit
git add .
git commit -m "Initial project setup"
git push -u origin main
```

Step 5: Start Coding!

- Implement one component at a time
- Commit frequently with meaningful messages
- Test as you go
- Document your code

Resources & Help

Course Materials

- **Lecture slides** and notes
- **Lab exercises** and examples
- **Cryptographic libraries** documentation
- **NIST cryptographic standards**

Getting Help

- **Office hours**: [TBD]
- **Email**: adil.akhmetov@sdu.edu.kz
- **GitHub Discussions**: [If available]
- **Course forum**: [If available]

Recommended Reading

- **"Real-World Cryptography"** - David Wong
- **"Serious Cryptography"** - Jean-Philippe Aumasson
- **OWASP Cryptographic Storage Cheat Sheet**
- **NIST Cryptographic Standards**

Useful Tools

- **Cryptography libraries**: Python `cryptography`, Node.js `crypto`
- **Testing**: pytest, jest, unittest
- **Documentation**: Sphinx, JSDoc, Markdown
- **Version control**: Git, GitHub

Example: Project Proposal Template

Project Proposal Format

Project Title: [Your project name]

Team Members:

- [Name 1] (GitHub: @username1) - Role: [Backend/Crypto/Frontend]
- [Name 2] (GitHub: @username2) - Role: [Backend/Crypto/Frontend]
- [Name 3] (GitHub: @username3) - Role: [Backend/Crypto/Frontend]

Project Option: [Option 1/2/3/4]

Brief Description: [2-3 sentences describing your project]

Cryptographic Components:

-
-
-
- …

Architecture Overview: [Brief description of system architecture]

Questions & Support

Frequently Asked Questions

Q: Can we use existing cryptographic libraries?

A: Yes! You're encouraged to use established libraries like `cryptography` (Python) or `crypto` (Node.js). However, you must understand and explain how the algorithms work.

Q: Do we need to implement everything from scratch?

A: No. You can use libraries, but you should implement at least one cryptographic primitive from scratch to demonstrate understanding.

Q: What if our demo doesn't work during presentation?

A: Prepare backup videos or screenshots. Explain what went wrong and how you would fix it. Partial credit may be given.

Q: Can we modify a project option?

A: Yes, with instructor approval. Submit your modified proposal early.

Q: How do we handle team conflicts?

A: Communicate early and often. Use GitHub issues to track tasks. If conflicts arise, contact the instructor.

Q: What if we finish early?

A: Add bonus features! Implement additional security measures, improve UI/UX, add more tests, or explore advanced topics.

Academic Integrity

Allowed

- ✅ **Using cryptographic libraries**
- ✅ **Referencing course materials**
- ✅ **Using online documentation**
- ✅ **Collaborating within your team**
- ✅ **Asking for help** (with attribution)

Not Allowed

- ❌ **Copying code** from other teams
- ❌ **Using code** from online without attribution
- ❌ **Submitting work** you didn't contribute to
- ❌ **Sharing code** with other teams
- ❌ **Plagiarism** of any kind

Attribution

If you use code from:

- **Stack Overflow**: Include link and attribution
- **GitHub repositories**: Include license and attribution
- **Documentation**: Mention source
- **Course materials**: Acknowledge instructor

Consequences

Violations of academic integrity will result in:

- **Zero points** for the project
- **Academic misconduct** report
- **Potential course failure**

Remember: The goal is learning. It's better to submit incomplete work that you understand than perfect code you copied without comprehension.

Good Luck! 🚀

Start early, test often, and document everything!

We're excited to see what you build!

Questions? Contact: adil.akhmetov@sdu.edu.kz 📧