Upgradeble Smart Contracts with Exploit Detection and Prevention

Comprehensive System Documentation

Table of Contents

1. Introduction

This document provides comprehensive documentation for a system comprising an upgradeable smart contract with a known vulnerability, along with microservices for exploit detection, notification, and prevention. The system also includes reporting capabilities.

2. Functional Requirements

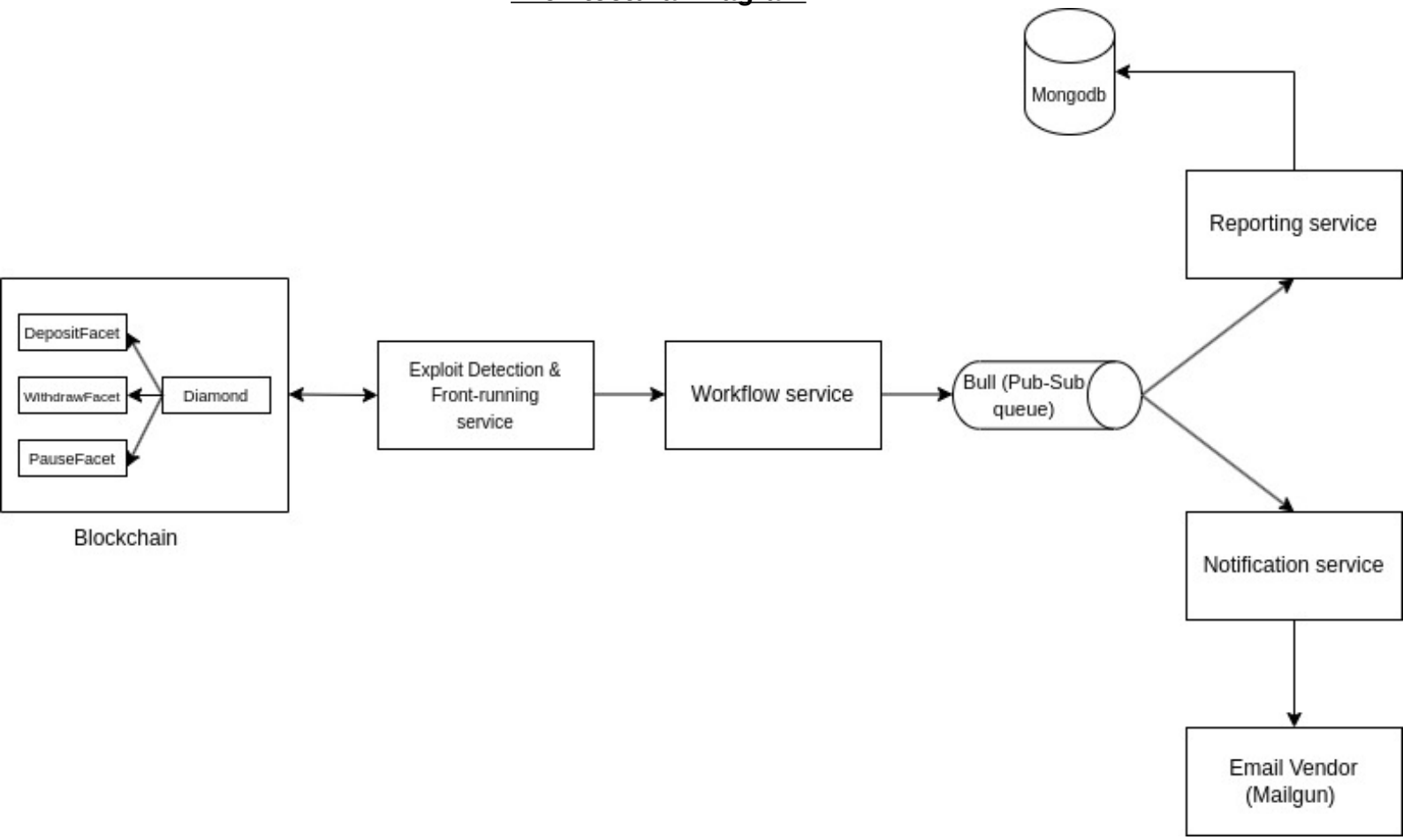
1. Develop an upgradeable Solidity smart contract vulnerable to reentrancy or flash loan attacks.
2. Implement Ownable and Pausable functionalities using OpenZeppelin libraries.
3. Create an exploit detection and notification microservice.
4. Develop a front-running microservice to pause the contract before suspicious transactions occur.
5. Implement a detailed reporting and analytics system.
6. Enable manual upgrade functionality for the smart contract.

3. Non-Functional Requirements

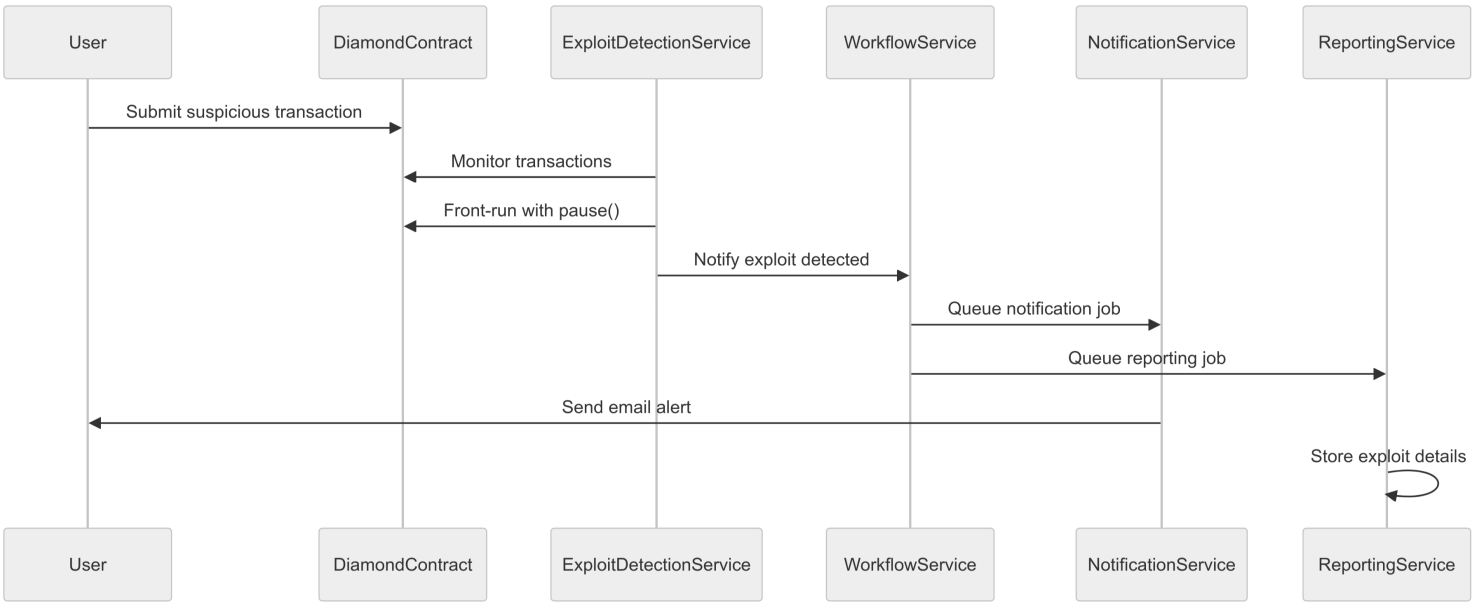
1. Security: Implement robust security measures for microservices and databases.
2. Scalability: Design the system to handle increasing numbers of transactions and users.
3. Performance: Ensure low latency for exploit detection and prevention.
4. Reliability: Implement fault-tolerance and error handling mechanisms.
5. Maintainability: Use modular design and clear documentation for ease of maintenance.
6. Interoperability: Ensure smooth communication between smart contracts and microservices.

4. High-Level Design

Architectural Diagram



Sequence Diagram



The system consists of the following main components:

1. Smart Contract: Upgradeable contract using the Diamond Proxy pattern.
2. Exploit Detection and Front-Running Microservice: Monitors transactions and pauses the contract if suspicious activity is detected.
3. Workflow Service: Manages the flow of information between microservices.
4. Notification Microservice: Sends email notifications about detected exploits.
5. Reporting Microservice: Stores and analyzes exploit-related data.

5. Low-Level Design

5.1 Smart Contract

- **Pattern**: Implements Diamond Proxy pattern for upgradeability
- **Facets**: Includes DepositFacet, WithdrawFacet, and PauseFacet
- **Libraries**: Uses OpenZeppelin for Ownable and Pausable functionalities

5.2 Exploit Detection and Front-Running Microservice

- **Functionality**: Monitors blockchain for suspicious transactions, implements dynamic gas pricing for front-running, and communicates with Workflow Service to trigger notifications and reports
- **Files**:
  - `main.ts`
  - `service.ts`
  - `module.ts`

5.3 Workflow Service

- **Framework**: Built with NestJS
- **Queue Management**: Uses Bull queue for job management
- **Communication**: Coordinates communication between microservices
- **Files**:
  - `main.ts`
  - `service.ts`
  - `controller.ts`
  - `module.ts`

5.4 Notification Microservice

- **Subscription**: Subscribes to Bull queue for notification jobs
- **Email Service**: Sends emails using Mailgun API
- **Files**:
  - `main.ts`
  - `processor.ts`
  - `service.ts`
  - `module.ts`

5.5 Reporting Microservice

- **Subscription**: Subscribes to Bull queue for reporting jobs

- **Database**: Stores exploit data in MongoDB Atlas
- **Files**:
    - `main.ts`
    - `processor.ts`
    - `service.ts`
    - `module.ts`

Smart Contract Architecture

6.1 Diamond Proxy Pattern

The Diamond Proxy pattern is an advanced upgradeability pattern for smart contracts. It allows for modular upgrading of contract functionality and helps overcome the contract size limitation in Ethereum.

Key Components:

- **Diamond**: The main contract that delegates calls to facets
- **Facets**: Separate contracts containing the implementation logic
- **DiamondCut**: Functionality for adding, replacing, or removing facets
- **DiamondLoupe**: Functions for inspecting the diamond's structure

Pros:

- Modular upgradeability
- Overcomes contract size limits
- Allows sharing of functionality across diamonds

Cons:

- Increased complexity
- Potential for centralization if not properly managed

How it works:

1. The Diamond contract uses `delegatecall` to forward function calls to the appropriate facet.
2. The DiamondCut functionality allows for adding, replacing, or removing facets.
3. Function selectors are used to map function calls to the correct facet address.

6.2 Facets

The system implements the following facets:

1. **DepositFacet**: Handles deposit functionality
2. **WithdrawFacet**: Manages withdrawal logic (intentionally vulnerable)
3. **PauseFacet**: Implements pause/unpause functionality
4. **DiamondCutFacet**: Manages diamond upgrades
5. **DiamondLoupeFacet**: Provides introspection functions

Key points:

- Facets are implemented as separate contracts
- Each facet focuses on a specific piece of functionality
- Facets can be upgraded independently

Microservices Architecture

The microservices architecture consists of four key services that communicate through a message queue (Bull) managed by the Workflow Service. This ensures loose coupling and scalability. The following sections describe each service in detail and explain the code flow.

7.1 Exploit Detection and Front-Running Service

Functionality: Monitors blockchain transactions, detects potential exploits, and front-runs malicious transactions to prevent them.

Code Flow:

1. **Transaction Monitoring**:
    - `main.ts`: Initializes the service.
    - `module.ts`: Configures the service module.
    - `service.ts`: Contains the core logic for monitoring transactions. It listens to blockchain events, analyzes transactions for suspicious activities, and uses dynamic gas pricing to front-run suspicious transactions.
2. **Communication with Workflow Service**:
    - Detected exploits are sent to the Workflow Service using HTTP requests. Done in service.ts as well.

Classes Involved:

- `exploit-detection-and-front-running.service`: Manages front-running actions and notifies workflow service

7.2 Workflow Service

Functionality: Manages communication between microservices using Bull queues for job management.

Code Flow:

1. **Initialization and Configuration**:
    - `main.ts`: Bootstraps the application.
    - `module.ts`: Configures the NestJS module.
    - `service.ts`: Contains business logic for job management and coordination between services.
2. **API Endpoints**:

    - `controller.ts`: Defines endpoints for receiving exploit detection data and dispatching jobs to appropriate queues.
3. **Queue Management**:
    - Jobs are added to Bull queues (e.g., notification queue, reporting queue) based on the data received.

Classes Involved:

- `workflow.service.ts`: Coordinates job management and communication between microservices.
- `Workflow.controller.ts`: Handles API requests.

7.3 Notification Service

Functionality: Sends email notifications about detected exploits.

Code Flow:

1. **Queue Subscription**:
    - `main.ts`: Initializes the service.
    - `module.ts`: Configures the service module.
    - `processor.ts`: Listens to the notification queue and processes jobs.
2. **Email Sending**:
    - `service.ts`: Uses the Mailgun API to send emails based on the job data.

Classes Involved:

- `notification.processor.ts`: Processes jobs from the notification queue.
- `Notification.service.ts`: Manages email sending logic.

7.4 Reporting Service

Functionality: Stores and analyzes exploit-related data.

Code Flow:

1. **Queue Subscription**:
    - `main.ts`: Initializes the service.
    - `module.ts`: Configures the service module.
    - `processor.ts`: Listens to the reporting queue and processes jobs.
2. **Data Storage**:
    - `service.ts`: Handles storing data in MongoDB Atlas and performing analytics.

Classes Involved:

- `reporting.processor.ts`: Processes jobs from the reporting queue.
- `Reporting.service.ts`: Manages data storage and analysis.

Conclusion

This microservices architecture demonstrates a robust system for monitoring, detecting, and preventing smart contract exploits. Each microservice has a distinct role, ensuring modularity and scalability. Communication through the Workflow Service and Bull queues provides a flexible and efficient way to manage tasks and maintain system integrity.