L44: Lab – Security Audit

Module F: Advanced Topics

Blockchain & Cryptocurrency Course

December 2025

Lab Overview

- **Objective**: Perform security audit on vulnerable smart contracts
- **Skills Practiced**:
  1. Manual code review (identify vulnerabilities by inspection)
  2. Automated tool usage (Slither, Mythril)
  3. Exploit writing (demonstrate vulnerability)
  4. Fix implementation (apply security patterns)
- **Contracts to Audit**:
  1. VulnerableBank (reentrancy)
  2. InsecureToken (integer overflow, access control)
  3. BadOracle (oracle manipulation)
- **Deliverable**: Audit report with findings, severity, fixes

Lab Setup

1. **Install Tools**:

```
# Hardhat development environment
npm install --save-dev hardhat
npx hardhat init

# Security tools
pip3 install slither-analyzer
pip3 install mythril
```

2. **Clone Vulnerable Contracts Repository**:

```
git clone https://github.com/[course-repo]/vulnerable-contracts-lab.git
cd vulnerable-contracts-lab
npm install
```

3. **Project Structure**:
    - `contracts/vulnerable/` – Contains buggy contracts
    - `contracts/secure/` – Empty (your fixes go here)
    - `test/exploits/` – Write exploit tests

Exercise 1: VulnerableBank Contract

Contract Code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VulnerableBank {
    mapping(address => uint) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");
        (bool success, ) = msg.sender.call{value: _amount}("");
        require(success, "Transfer failed");
        balances[msg.sender] -= _amount;
    }

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Exercise 1: Tasks

1. **Manual Review**:
    - Read the contract code carefully
    - Identify the vulnerability (hint: focus on `withdraw` function)
    - Classify the vulnerability type
    - Assess severity (Critical, High, Medium, Low)
2. **Automated Analysis**:

    ```
    slither contracts/vulnerable/VulnerableBank.sol
    myth analyze contracts/vulnerable/VulnerableBank.sol
    ```

    - Compare tool findings with your manual review
    - Do tools identify the vulnerability?
3. **Write Exploit**: Create `AttackBank.sol` to drain funds
4. **Implement Fix**: Create `SecureBank.sol` with proper mitigations

Exercise 1: Exploit Contract

Task: Write `AttackBank.sol` to exploit reentrancy

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./VulnerableBank.sol";

contract AttackBank {
    VulnerableBank public vulnerableBank;
    uint public attackAmount;

    constructor(address _vulnerableBankAddress) {
        vulnerableBank = VulnerableBank(_vulnerableBankAddress);
    }

    // TODO: Implement attack logic
    // 1. Deposit funds into VulnerableBank
    // 2. Call withdraw to trigger reentrancy
    // 3. Fallback function recursively calls withdraw

    receive() external payable {
        // TODO: Implement reentrancy logic
    }
}
```

Exercise 1: Secure Implementation

Task: Implement `SecureBank.sol` with reentrancy protection **Mitigation Options**:

1. **Checks-Effects-Interactions Pattern**:
   - Update state before external call
2. **Reentrancy Guard**:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SecureBank is ReentrancyGuard {
    function withdraw(uint _amount) public nonReentrant {
        // Protected against reentrancy
    }
}
```

3. **Verify with Slither**: Ensure no reentrancy warnings

Exercise 2: InsecureToken Contract

Contract Code:

```
pragma solidity ^0.7.6;  // Vulnerable version

contract InsecureToken {
    mapping(address => uint256) public balances;
    address public owner;

    function mint(address _to, uint256 _amount) public {
        balances[_to] += _amount;  // No overflow check!
    }

    function transfer(address _to, uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        balances[_to] += _amount;
    }

    function destroyContract() public {
        selfdestruct(payable(owner));  // No access control!
    }
}
```

Multiple vulnerabilities present

Exercise 2: Tasks

1. **Manual Review**: Identify ALL vulnerabilities
   - Hint: Look for integer overflow, access control, uninitialized variables
   - List each vulnerability with severity
2. **Run Slither and Mythril**:

```
slither contracts/vulnerable/InsecureToken.sol --solc-version 0.7.6
```

   - Which vulnerabilities do tools detect?
   - Any false positives or missed issues?
3. **Exploit Scenarios**:
   - Overflow `balances` by minting large amounts
   - Call `destroyContract` as non-owner
4. **Implement SecureToken.sol**:
   - Upgrade to Solidity 0.8.0+ (built-in overflow checks)
   - Add `onlyOwner` modifier
   - Initialize `owner` in constructor

Exercise 2: Secure Implementation

Fix Vulnerabilities:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol";

contract SecureToken is Ownable {
    mapping(address => uint256) public balances;

    // Overflow protection: Solidity 0.8.0+ built-in
    function mint(address _to, uint256 _amount) public onlyOwner {
        balances[_to] += _amount;  // Safe from overflow
    }

    function transfer(address _to, uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient balance");
        balances[msg.sender] -= _amount;  // Safe from underflow
        balances[_to] += _amount;
    }

    // Access control: onlyOwner prevents unauthorized destruction
}
```

Exercise 3: BadOracle Contract

Contract Code:

```
interface IUniswapV2Pair {
    function getReserves() external view returns (uint112, uint112, uint32);
}

contract BadOracle {
    IUniswapV2Pair public pair;

    function getPrice() public view returns (uint) {
        (uint112 reserve0, uint112 reserve1, ) = pair.getReserves();
        return (reserve1 * 1e18) / reserve0;  // Instant price, manipulable!
    }

    function borrow(uint collateralAmount) public {
        uint collateralValue = collateralAmount * getPrice();
        // Lend based on manipulable oracle...
    }
}
```

Vulnerability: Single-block price manipulation via flash loans

Exercise 3: Tasks

1. **Manual Review**:
   - Why is this oracle vulnerable?
   - How can an attacker manipulate `getPrice()`?
   - Design flash loan attack scenario
2. **Write Exploit**:
   - Use Hardhat mainnet fork
   - Flash loan from Aave
   - Manipulate Uniswap pool
   - Exploit `BadOracle.borrow()`
   - Restore pool state, repay flash loan
3. **Implement Secure Oracle**:
   - Option 1: Uniswap V2 TWAP oracle
   - Option 2: Chainlink price feed
   - Option 3: Hybrid (multiple sources)

Exercise 3: Secure Oracle Implementation

Option 1: Uniswap V2 TWAP:

```
import "@uniswap/v2-periphery/contracts/libraries/UniswapV2OracleLibrary.sol";

contract SecureOracle {
    address public pair;
    uint public price0CumulativeLast;
    uint32 public blockTimestampLast;
    uint public constant PERIOD = 1 hours;

    function update() external {
        (uint price0Cumulative, , uint32 blockTimestamp) =
            UniswapV2OracleLibrary.currentCumulativePrices(pair);
        uint timeElapsed = blockTimestamp - blockTimestampLast;
        require(timeElapsed >= PERIOD, "Period not elapsed");
        // Calculate TWAP over PERIOD
        uint priceAverage = (price0Cumulative - price0CumulativeLast) / timeElapsed;
        // Update stored values...
    }
}
```

Attacker must manipulate price for 1 hour (expensive)

Exercise 3: Chainlink Integration

Option 2: Chainlink Price Feed:

```
import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

contract ChainlinkOracle {
    AggregatorV3Interface internal priceFeed;

    constructor(address _priceFeed) {
        priceFeed = AggregatorV3Interface(_priceFeed);
    }

    function getPrice() public view returns (int) {
        (
            , // roundId
            int price,
            , // startedAt
            uint timeStamp,
            // answeredInRound
        ) = priceFeed.latestRoundData();
        require(timeStamp > 0, "Round not complete");
        return price;
    }
}
```

Decentralized oracle network, manipulation-resistant

Slither Deep Dive

Run Slither with Detailed Output:

```
slither . --print human-summary
slither . --detect reentrancy-eth,reentrancy-no-eth
slither . --exclude-informational --exclude-low
```

Detector Categories:

- **High/Critical**: Reentrancy, access control, uninitialized storage
- **Medium**: Timestamp dependence, weak randomness
- **Low**: Solidity version, naming conventions
- **Informational**: Gas optimizations, best practices

Integration: Add to CI/CD pipeline

```
slither . --fail-high --fail-medium
```

Mythril Deep Dive

Run Mythril with Specific Modules:

```
myth analyze contracts/VulnerableBank.sol \
    --execution-timeout 300 \
    --solver-timeout 10000 \
    --max-depth 50
```

Interpretation of Results:

- **SWC-107**: Reentrancy
- **SWC-101**: Integer overflow/underflow
- **SWC-105**: Unprotected ether withdrawal
- **SWC-115**: Authorization through tx.origin

Limitations:

- Path explosion for complex contracts
- False positives possible
- Combine with manual review

Writing Exploit Tests

Hardhat Test Structure:

```
const { expect } = require("chai");

describe("VulnerableBank Exploit", function () {
  it("Should drain bank via reentrancy", async function () {
    const [attacker, victim] = await ethers.getSigners();

    // Deploy VulnerableBank
    const Bank = await ethers.getContractFactory("VulnerableBank");
    const bank = await Bank.deploy();

    // Victim deposits 10 ETH
    await bank.connect(victim).deposit({ value: ethers.utils.parseEther("10") });

    // Deploy AttackBank
    const Attack = await ethers.getContractFactory("AttackBank");
    const attack = await Attack.deploy(bank.address);

    // Execute attack
    await attack.attack({ value: ethers.utils.parseEther("1") });

    // Verify bank is drained
    expect(await ethers.provider.getBalance(bank.address)).to.equal(0);
  });
});
```

Audit Report Template

Professional Audit Report Structure:

1. **Executive Summary**:
    - Scope, methodology, timeline
    - High-level findings summary
2. **For Each Finding**:
    - Title: “Reentrancy in withdraw function”
    - Severity: Critical / High / Medium / Low / Informational
    - Location: Contract name, line numbers
    - Description: Explain the vulnerability
    - Impact: What can an attacker achieve?
    - Proof of Concept: Code or steps to reproduce
    - Recommendation: How to fix (with code example)
3. **Summary Table**: All findings with severity
4. **Conclusion**: Overall security assessment

Sample Finding: Reentrancy

Finding: Reentrancy Vulnerability in VulnerableBank.withdraw()
Severity: Critical
Location: `contracts/VulnerableBank.sol, lines 12-16`
Description: The `withdraw` function performs an external call to `msg.sender` before updating the user's balance. This allows a malicious contract to recursively call `withdraw` and drain funds.
Impact: Complete loss of contract funds. An attacker can drain the entire bank balance in a single transaction.
Proof of Concept: See `test/exploits/test-reentrancy.js`
Recommendation: Apply Checks-Effects-Interactions pattern:

```
balances[msg.sender] -= _amount;  // State update FIRST
(bool success, ) = msg.sender.call{value: _amount}("");  // External call LAST
```

Alternatively, use OpenZeppelin's ReentrancyGuard.

Lab Deliverable Checklist

Submit the Following:

1. **Audit Report (PDF)**:
   - Findings for all three contracts
   - Severity classifications
   - Recommendations with code
2. **Exploit Contracts**:
   - `AttackBank.sol`
   - `ExploitToken.sol`
   - `OracleAttack.sol`
3. **Secure Implementations**:
   - `SecureBank.sol`
   - `SecureToken.sol`
   - `SecureOracle.sol`
4. **Test Suite**:
   - Tests demonstrating exploits
   - Tests proving secure versions are not exploitable
5. **Tool Output**: Slither and Mythril reports

Bonus Challenge: Advanced Vulnerabilities

Optional Advanced Exercises:

1. **Flashloan + Oracle Attack**:
   - Combine flash loan with BadOracle exploit
   - Maximize profit extraction
2. **Signature Replay Attack**:
   - Contract accepts signed messages for token transfers
   - Exploit lack of nonce/chainId
3. **Delegate Call Vulnerability**:
   - Proxy contract with storage collision
   - Overwrite critical state variables
4. **Gas Griefing DoS**:
   - Unbounded loop in distribution function
   - Make contract unusable by adding many recipients

Summary

- **Security audit workflow**: Manual review → Automated tools → Exploit → Fix
- **Exercise 1**: Reentrancy in VulnerableBank (Checks-Effects-Interactions fix)
- **Exercise 2**: Integer overflow + access control in InsecureToken (Solidity 0.8.0+ fix)
- **Exercise 3**: Oracle manipulation in BadOracle (TWAP or Chainlink fix)
- **Tools**: Slither (fast static analysis), Mythril (symbolic execution)
- **Exploit writing**: Demonstrate real attack scenarios in tests
- **Deliverable**: Professional audit report with findings and recommendations
- **Real-world skills**: Manual code review remains critical, tools complement but don't replace human analysis