Smart Contracts Exercise 09:
Vulnerabilities Detection

1 Introduction

This exercise focuses on general vulnerability detection in smart contracts. We will explore static code analysis tools such as Mythril and Slither to identify vulnerabilities in contracts from previous exercises. The exercise will also cover the implementation of unit tests, stateless fuzzing tests, and stateful fuzzing tests with invariants. We will introduce the Foundry development environment (new only for students who previously used Hardhat). The primary task involves writing tests for a simple toy smart contract and then applying this knowledge to develop more complex tests for the DEX contract from earlier exercise.

Project Setup

You have two options for working with this exercise: using a Docker container or local installation. Choose the option that best fits your preferences. For students who are accustomed to working in the Hardhat environment and using Docker, it's important to note that this exercise uses a different Docker image.

1.1 Using Docker with VS Code

This option uses Docker to create a development environment with all the necessary tools and dependencies pre-installed.

Prerequisites:

- **Docker** - A platform for developing, shipping, and running applications in containers.
- **Visual Studio Code** - A lightweight but powerful source code editor.
- **Dev Containers** - An extension to VS Code that lets you use a Docker container as a full-featured development environment.

Setting Up the Project:

1. Visit the following GitLab repository and clone it to your local machine.
2. Open the repository folder in VS Code.
3. When prompted, click "Reopen in Container" or use the command palette (F1) and run `Dev Containers: Reopen in Container`.

1.2 Local Setup

If you prefer working directly on your machine without Docker, you can set up the development environment locally. Before setting up Foundry, ensure that you have the following installed on your system:

Prerequisites

- **Rust Toolchain** - Since Foundry is built in Rust, you'll need the Rust compiler and Cargo, Rust's package manager. The easiest way to install both is by using rustup.rs.
- **Bun** - JavaScript runtime & toolkit for installing dependencies and running scripts. Install it from bun.sh.

If you don't have Rust installed, you can install it using:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

To install Bun, use the following command:

```
$ curl -fsSL https://bun.sh/install | bash
```

Open your terminal and run the following command to verify the Rust and Bun installation:

```
$ rustc --version
$ cargo --version
$ bun --version
```

Installing Foundry

You can install Foundry using Foundryup, the official installer:

```
$ curl -L https://foundry.paradigm.xyz | bash
$ foundryup
```

This will install the Foundry toolkit, including:

- **Forge** - Testing framework for Ethereum
- **Cast** - Command-line tool for interacting with smart contracts
- **Anvil** - Local Ethereum node for development
- **Chisel** - Solidity REPL

Verify the installation by running:

```
$ forge --version
$ cast --version
$ anvil --version
```

Setting Up the Project

1. Visit the following GitLab repository and clone it to your local machine.
2. Open a terminal and navigate to the project directory.
3. Install the project dependencies by running `bun install`.

2 Static Analysis

Static analysis is a method of examining code without executing it. Unlike dynamic analysis (which examines code during execution), static analysis looks at the source code or bytecode to find patterns that match known vulnerability types. If you're using a local setup, you'll need to install Slither and Mythril first, while those using the Docker container already have these tools available in the container.

Slither

Slither is a static analysis framework from Trail of Bits that runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses.

To install Slither using pip3:

```
$ pip3 install slither-analyzer
```

To run Slither on a Solidity file:

```
$ slither <contract_file.sol>
```

Mythril

Mythril is an open-source security analysis tool for Ethereum smart contracts. It uses symbolic execution, SMT solving, and taint analysis to detect a variety of security vulnerabilities.

To install Mythril using pip3:

```
$ pip3 install mythril
```

To run Mythril on a Solidity file:

```
$ myth analyze <contract_file.sol>
```

Task: Analyze the CatCharity Contract

The CatCharity contract collects donations to help save kittens. Your task is to use static analysis tools to identify potential vulnerabilities in this contract.

1. Open the `src/CatCharity.sol` file and review the code
2. Run Mythril static analysis:

```
$ myth analyze src/CatCharity.sol
```

3 Testing in Foundry

Foundry provides powerful tools for testing smart contracts, with a focus on flexibility and efficiency. Unlike Hardhat (which uses JavaScript), Foundry uses Solidity for writing tests and scripts.

- Test files end with `.t.sol`
- Test contracts inherit from `forge-std/Test.sol`
- Test functions start with `test_`
- Fuzzing tests start with `testFuzz_`
- Invariant tests start with `invariant_`

Unit Testing

Unit tests verify that individual functions or components of your contract work as expected in isolation. In Foundry, unit tests are written in Solidity and run using the Forge test framework.

```
function test_Deposit() public {
    uint256 depositAmount = 1 ether;

    // Alice makes a deposit
    vm.prank(alice);
    piggyBank.deposit{value: depositAmount}();

    // Check totalDeposits
    assertEq(
        piggyBank.totalDeposits(),
        depositAmount,
        "totalDeposits should match deposit amount"
    );
}
```

Fuzzing Tests

Fuzzing (or property-based testing) involves generating random inputs to test your contract's behavior under various conditions. Foundry provides built-in support for fuzzing through the `forge test` command.

```
function testFuzz_Deposit(uint96 amount) public {
    // Bound the amount to a reasonable range
    uint256 depositAmount = bound(uint256(amount), 0 ether, 99 ether);

    // Alice makes a deposit
    vm.prank(alice);
    piggyBank.deposit{value: depositAmount}();

    // Check totalDeposits
    assertEq(
        piggyBank.totalDeposits(),
        depositAmount,
        "totalDeposits should match deposit amount"
    );
}
```

Invariant Testing

Invariant testing checks that certain properties (invariants) of your contract remain true regardless of the sequence of operations performed. Foundry supports invariant testing through specialized test functions and handlers.

```
function invariant_balanceMatchesAccountingDiff() public view {
    assertEq(
        address(piggyBank).balance,
        piggyBank.totalDeposits() - piggyBank.totalWithdrawals(),
        "Contract balance should equal totalDeposits - totalWithdrawals"
    );
}
```

Task: PiggyBank

The PiggyBank contract is a simple savings contract where anyone can deposit ETH, but only the owner can withdraw.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity =0.8.28;

contract PiggyBank {
    address public immutable owner;
    uint256 public totalDeposits;
    uint256 public totalWithdrawals;

    error NotOwner();
    error InsufficientFunds();

    constructor() {
        owner = msg.sender;
    }

    function deposit() public payable {
        totalDeposits += msg.value;
    }

    function withdraw(uint256 amount) public {
        if (msg.sender != owner) revert NotOwner();
        if (amount > address(this).balance) revert InsufficientFunds();
        totalWithdrawals += amount;
        payable(owner).transfer(amount);
    }
}
```

Task 1: Complete the PiggyBank Unit Tests

Your task is to complete the missing unit tests in the `PiggyBankUnitTest.t.sol` file. Look for the `TODO` comments in the code to identify the test functions you need to implement.

Follow these steps:

1. Open the `test/PiggyBankUnitTest.t.sol` file
2. Locate the test functions marked with `TODO` comments
3. Implement each test according to the provided instructions
4. Run the tests using the following command:

```
$ bun run test:unit
```

Task 2: Implement PiggyBank Stateless Fuzzing Tests

Stateless fuzzing tests help identify edge cases by providing random inputs to your contract functions. Your task is to implement the missing stateless fuzzing tests in the `PiggyBankStatelessFuzzTest.t.sol` file.

Follow these steps:

1. Open the `test/PiggyBankStatelessFuzzTest.t.sol` file
2. Locate the test functions marked with `TODO` comments

3. Implement each fuzzing test according to the provided instructions
4. Run the fuzzing tests using the following command:

```
$ bun run test:fuzz:piggy
```

Task 3: Add PiggyBank Invariant Tests

Invariant tests verify that certain properties of your contract hold true regardless of the sequence of operations. Your task is to implement the missing invariant tests in the `PiggyBankInvarintTest.t.sol` file.

Follow these steps:

1. Open the `test/PiggyBankInvarintTest.t.sol` file
2. Locate the invariant functions marked with `TODO` comments
3. Implement each invariant test according to the provided instructions
4. Run the invariant tests using the following command:

```
$ bun run test:invariant:piggy
```

Task 4: Implement SimpleDEX Fuzzing Tests

Apply your knowledge of fuzzing tests to the more complex SimpleDEX contract. Your task is to implement the missing fuzzing tests in the `SimpleDEXStatelessFuzzTest.t.sol` file.

Follow these steps:

1. Open the `test/SimpleDEXStatelessFuzzTest.t.sol` file
2. Locate the test functions marked with `TODO` comments
3. Implement each fuzzing test according to the provided instructions
4. Run the fuzzing tests using the following command:

```
$ bun run test:fuzz:dex
```

Task 5: Add SimpleDEX Invariant Tests

Finally, implement the missing invariant tests for the SimpleDEX contract in the `SimpleDEXInvariantTes` file. This is the most challenging part of the exercise, as it requires understanding the core invariants of a DEX.

Follow these steps:

1. Open the `test/SimpleDEXInvariantTest.t.sol` file
2. Locate the invariant functions marked with `TODO` comments
3. Implement at least three custom invariants that ensure the DEX operates correctly
4. Run the invariant tests using the following command:

```
$ bun run test:invariant:dex
```