Smart Contracts Exercise 09: Vulnerabilities Detection

1 Introduction

This exercise focuses on general vulnerability detection in smart contracts. We will explore the static analysis tool 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 (this will be 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 GitHub 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.
- **Tip**: If you already have Node Package Manager installed and use it regularly, you can use it instead of Bun.

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 commands to verify the Rust and Bun installations:

```
$ 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 GitHub 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`. (Alternatively with NPM, you can use `npm 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 first. Students using the Docker container already have these tools available in the container.

Slither

Slither is a static analysis framework 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>
```

Now let's try using Slither to identify some vulnerabilities from our exercises. First, we'll analyze the `src/CatCharity` contract from Exercise 04 – Re-Entrancy. Run:

```
$ slither src/CatCharity.sol
```

Expected output:

```
INFO:Detectors:
Reentrancy in CatCharity.claimRefund() (src/CatCharity.sol#98-114):
        External calls:
        - (success,None) = address(msg.sender).call{value: donated}()
        (src/CatCharity.sol#107)
        State variables written after the call(s):
        - donations[msg.sender] = 0 (src/CatCharity.sol#112)
        CatCharity.donations (src/CatCharity.sol#24) can be used in
        cross function reentrancies:
        - CatCharity.claimRefund() (src/CatCharity.sol#98-114)
        - CatCharity.donate() (src/CatCharity.sol#78-80)
        - CatCharity.donations (src/CatCharity.sol#24)
```

As you can see, Slither successfully identified a re-entrancy vulnerability in the `CatCharity` contract. It also provided a link to documentation about this vulnerability type.

Task 1: Analyze Contracts with Slither

Analyze all other contracts in the `src` directory using Slither (`SimpleDEX`, `USDCToken`, `PiggyBank`) and review Slither's output. Determine whether the identified issues are actual problems or false positives. Examine the linked documentation for each reported vulnerability.

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. We won't cover all Foundry functions in detail here, but we will focus primarily on writing tests. For more information about working with Foundry, visit the project documentation Foundry Book. Let's now introduce the concept of smart contract testing with a simple example that you'll work with in this exercise. The PiggyBank contract is a simple savings contract where anyone can deposit ETH, but only the owner can withdraw.

Note: At the time of writing this exercise, Hardhat is adding support for Solidity-written tests and scripts in an alpha release. If you're interested, check it out: Hardhat 3 Alpha.

Note: Hardhat and Foundry are mutually compatible. Learn more about this topic: Integrating Hardhat with Foundry and Integrating Foundry with Hardhat.

```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);
    }
}
```

PiggyBank in Remix IDE

3.1 Unit Testing

Unit tests verify that individual functions or components of your contract work as expected in isolation. Documentation on how to write basic tests in Foundry is available here. Let's look at an example using our PiggyBank contract. We want to test whether after calling the `deposit()` function, the value of `totalDeposits` will match our deposit.

```
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"
    );
}
```

Another test could verify that if we try to withdraw more money than has been deposited into the contract, we receive the appropriate error – `InsufficientFunds`.

```
function test_RevertWhen_WithdrawExceedsBalance() public {
   // First make a deposit to the piggy bank
   vm.prank(alice);
   piggyBank.deposit{value: 1 ether}();

   // Owner tries to withdraw more than available (should fail)
   vm.prank(owner);
   vm.expectRevert(PiggyBank.InsufficientFunds.selector);
   piggyBank.withdraw(2 ether);
}
```

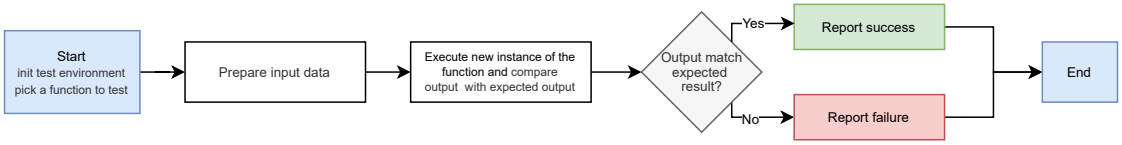

Figure 1: Unit Testing Flow

Task 2: Write Unit Tests for PiggyBank

In the file `test/PiggyBankUnitTest.t.sol`, complete the missing unit tests for the PiggyBank contract. Find the functions marked with `TODO` comments and implement them according to the instructions in the code. The file contains four completed unit tests and four unimplemented ones that you need to complete. Run the tests using the following command:

```
$ forge test --mp test/PiggyBankUnitTest.t.sol -v
```

You can use different verbosity levels to make debugging easier:

```
-v
    Pass multiple times to increase the verbosity (e.g. -v, -vv, -vvv).
    Verbosity levels:
    - 2: Print logs for all tests
    - 3: Print execution traces for failing tests
    - 4: Print execution traces for all tests, and setup traces for failing tests
    - 5: Print execution and setup traces for all tests
```

3.2 Stateless Fuzz Tests

Fuzzing involves generating random inputs to test your contract's behavior under various conditions. Unlike unit tests where you provide specific inputs manually, fuzzing tests automatically generate random (pseudo-random) inputs. The key advantage is that you can run the same test multiple times with different input data each time. You can generate completely random input data or constrain it to specific value ranges. When you run a fuzzing test, it executes many times in succession, each with different input data, allowing you to discover edge cases you might not have anticipated. Information about fuzz testing in Foundry is available here: Fuzz Testing.

```
function testFuzz_Deposit(uint96 amount) public {
    // Bound the random input 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"
    );
}
```

Note: In the literature, you'll often encounter confusion between different terms. To be precise, the method described above is called stateless fuzz testing. While we run the function multiple times with different inputs, each run uses a fresh contract instance.

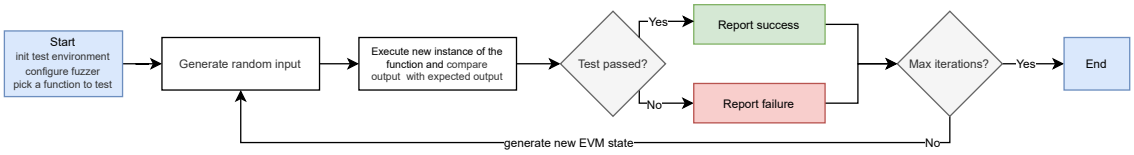

Figure 2: Stateless Fuzz Testing Flow

Task 3: Write Stateless Fuzz Tests for PiggyBank

In the file `test/PiggyBankStatelessFuzzTest.t.sol`, complete the missing fuzz tests for the PiggyBank contract. Find the functions marked with `TODO` comments and implement them according to the instructions in the code. The file contains four completed fuzz

tests and four unimplemented ones that you need to complete. Run the tests using the following command:

```
$ forge test --mp test/PiggyBankStatelessFuzzTest.t.sol -v
```

Note: Test configuration is set in the `foundry.toml` file. The default setting for stateless fuzz testing is 1000 iterations per test

3.3 Invariant Testing

Invariant testing checks that certain properties (invariants) of your contract remain true regardless of the sequence of operations performed. An invariant is a specific condition that must always be satisfied no matter which functions are called on our program, in any order. For example, an invariant for our PiggyBank contract could be that the current balance of the contract must equal the difference between all deposits and all withdrawals. This property should always hold in the system, regardless of how many times the `deposit()` and `withdraw()` functions are called, in what order, and by which addresses. Properly defining invariants is fundamental to this process. After defining the invariants, we can simulate the system's execution using stateful fuzz tests, where different functions will be called in various orders with different input data, and we'll verify that the invariants are always satisfied.

```
/**
* @notice Invariant #1: Contract balance should always equal totalDeposits -
    totalWithdrawals
* @dev This verifies the core accounting of the contract is correct
*/
function invariant_balanceMatchesAccountingDiff() public view {
    assertEq(
        address(piggyBank).balance,
        piggyBank.totalDeposits() - piggyBank.totalWithdrawals(),
        "Contract balance should equal totalDeposits - totalWithdrawals"
    );
}
```

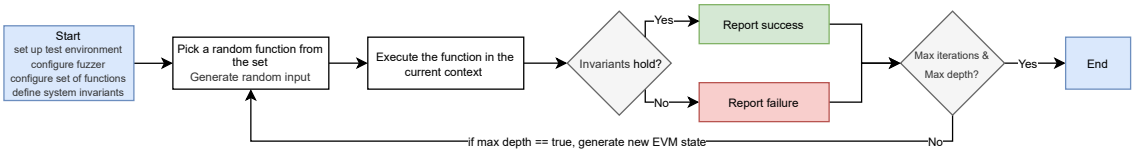

Figure 3: Invariant Testing Flow

Task 4: Write Invariants for PiggyBank

In the file `test/PiggyBankInvariantTest.t.sol`, complete the invariant definitions for the PiggyBank contract. First, carefully review the Foundry documentation on invariant testing. Then examine the structure of the `test/PiggyBankInvariantTest.t.sol` file to understand how the test is constructed. Your task is to define at least two additional invariants that should always hold true for the PiggyBank contract. Run the tests using the following command:

```
$ forge test --mp test/PiggyBankInvariantTest.t.sol -v
```

Warning: After any code change in invariant tests, you need to run the `forge clean` command because Foundry caches completed tests and only replicates them otherwise.
Note: Test configuration is set in the `foundry.toml` file. The default setting for invariant testing is 256 iterations per invariant with a maximum depth of 150. Each invariant is tested separately against a fresh EVM state.

Task 5: Implement SimpleDEX Test Suite

Testing the PiggyBank contract might not have been particularly exciting or challenging. In the second part of this exercise, you will test the SimpleDEX contract that you're familiar with from Exercise 06: Fool The Oracle. To complete this exercise, you must write all the missing unit tests, all the missing fuzz tests, and add meaningful invariants. Implement your solution in the files `test/SimpleDEXUnitTest.t.sol`, `test/SimpleDEXStatelessFuzzTest.t.sol`, and `test/SimpleDEXInvariantTest.t.sol`.

Verify your solution with:

```
$ forge test --mp test/SimpleDEXUnitTest.t.sol -v
$ forge test --mp test/SimpleDEXStatelessFuzzTest.t.sol -v
$ forge test --mp test/SimpleDEXInvariantTest.t.sol -v
```

Note: Foundry follows a specific naming convention for tests and invariants that must be followed; otherwise, the tests won't run:

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

Additional Resources

For more resources on fuzzing and invariant testing, check out these links:

- **Hands-on Learning**: Practical exercises on fuzzing
- **Real-world Examples**: Published professional fuzzing campaigns
- **Case Study**: Invariant Testing WETH With Foundry
- **Video Tutorial**: Fuzzing Workshop by Trail of Bits
- **Even More Resources**: Updated comprehensive list of fuzzing resources

4 Beyond the Course: The End?

This course has equipped you with foundational knowledge in smart contract security. However, the field of blockchain security is vast and constantly evolving. Here are resources to continue your journey:

Interactive Learning Resources – Development Focus

Practical hands-on courses focused on smart contract development.

- **CryptoZombies**: Interactive lessons for learning Solidity
- **SpeedRunEthereum**: Hands-on challenges to build Ethereum apps

Interactive Learning Resources – Security Focus

Practical exercises similar to those you solved in this course. Ethernaut offers shorter and simpler challenges in the style of the Vaults from Exercise 04, while Damn Vulnerable DeFi is more complex and focuses on DeFi applications, often using real copies of smart contracts.

- **Damn Vulnerable DeFi**: Challenges focusing on DeFi vulnerabilities
- **Ethernaut**: OpenZeppelin's collection of smart contract security challenges

University Courses

- **TUM: Blockchain-based Systems Engineering**: Blockchain course at Technical University of Munich
- **Berkeley DeFi Course**: Decentralized Finance course at the University of California, Berkeley
- **FIT ČVUT: NIE-BLO**: Blockchain Course at Czech Technical University in Prague

Cyfrin Courses

Cyfrin is a smart contract security audit company that has published up-to-date practical courses on their website, accessible for free.

- **Security and Auditing Full Course**: Comprehensive guide to smart contract security
- **Foundry Fundamentals**: Master the Foundry development environment
- **Formal Verification**: Learn to use formal methods to verify smart contracts
- **Advanced Foundry**: Take your Foundry skills to the next level
- **SoloDIT**: Cyfrin's searchable database of vulnerabilities found in audits

Competitive Audit Platforms

Participating in competitive audits is an excellent way to apply your skills in real-world scenarios, learn from others, and potentially earn rewards:

- **Code4rena**: Competitive audit platform with regular contests and substantial rewards
- **Sherlock**: Combines security competitions with protocol coverage
- **CodeHawks**: Newer platform with both private and public audit competitions
- **Immunefi**: The largest bug bounty platform in crypto

Security Tools and Technical Resources

- **Manticore**: Symbolic execution tool
- **Echidna**: Fuzzing tool
- **Medusa**: Fuzzing tool

Communities and Networking

- **Cyfrin**: Active community focused on smart contract security
- **Ethereum Research**: Technical discussions about Ethereum

It has been a pleasure guiding you through your first steps in smart contract security. Happy hacking, and best of luck on your smart contract security journey!