Fortephy Audit Report

Date 1/11/23

I have recently conducted a thorough security assessment of the Fortephy auditing tool, with a focus on its ability to identify and detect various vulnerabilities in smart contracts and blockchain networks. In this report, I have documented my findings and provided recommendations for improvement.

Introduction

Fortephy is a blockchain auditing tool designed to detect and identify vulnerabilities in smart contracts and other blockchain-related codebases. The assessment was conducted by running multiple codebases through the tool and evaluating its effectiveness in detecting known vulnerabilities.

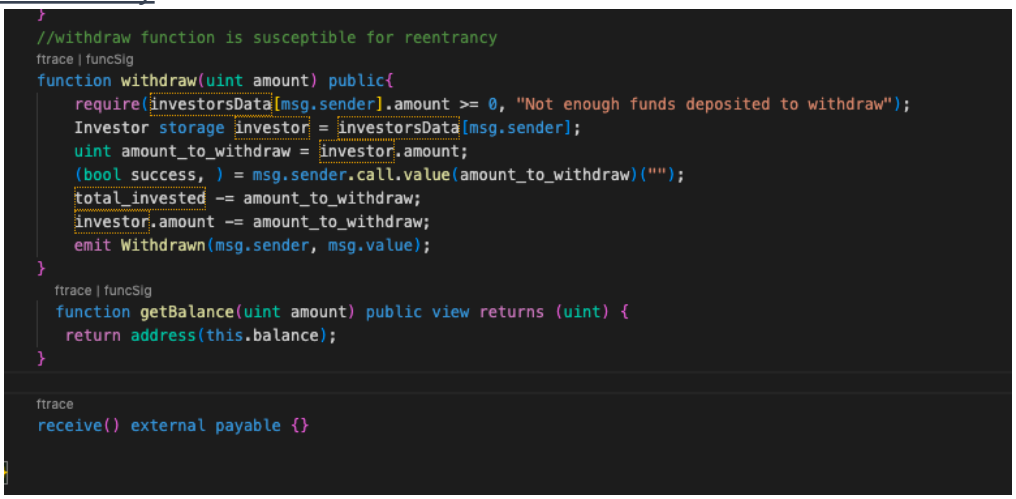
Methodology

Evaluation process:

a. Testing the tool's ability to detect reentrancy vulnerabilities.
b. Assessing the detection of the ERC20 approve race condition.
c. Analyzing the tool's effectiveness in identifying instances of checking for transfer success after the return statement.
d. Evaluating the detection of token deflationary exploits.
e. Conducting a hypothetical stress test to assess the robustness of the Fortephy API.

Findings:
During the assessment, it was observed that the Fortephy auditing tool did not successfully detect the following vulnerabilities:

Reentrancy



```
}
//withdraw function is susceptible for reentrancy
ftrace | funcSig
function withdraw(uint amount) public{
    require(investorsData[msg.sender].amount >= 0, "Not enough funds deposited to withdraw");
    Investor storage investor = investorsData[msg.sender];
    uint amount_to_withdraw = investor.amount;
    (bool success, ) = msg.sender.call.value(amount_to_withdraw)("");
    total_invested -= amount_to_withdraw;
    investor.amount -= amount_to_withdraw;
    emit Withdrawn(msg.sender, msg.value);
}
  ftrace | funcSig
  function getBalance(uint amount) public view returns (uint) {
   return address(this.balance);
}

ftrace
receive() external payable {}
```

The reentrancy attack can occur due to the improper ordering of the contract state changes and the external call to `msg.sender`. Before the contract state is updated, the external call is made. This allows the attacker to re-enter the `withdraw` function before the state changes, potentially draining more funds than they should be able to.

An attacker can exploit this vulnerability by implementing a fallback function in their contract that calls the `withdraw` function again. When the attacker calls the `withdraw` function, the funds are transferred to their contract, and the fallback function is triggered, which calls the `withdraw` function again before the state updates.

Recommendation:
To fix this vulnerability, you should use the "Checks-Effects-Interactions" pattern. This pattern recommends performing all checks first, updating the state (effects), and finally making external calls (interactions). This can be done by updating the contract state before making the external call, as shown below:

```
function withdraw(uint amount) public {
    require(investorsData[msg.sender].amount >= amount, "Not enough funds deposited to withdraw");
    Investor storage investor = investorsData[msg.sender];
    uint amount_to_withdraw = amount;

    // Update the state (Effects)
    total_invested -= amount_to_withdraw;
    investor.amount -= amount_to_withdraw;

    // Make external call (Interactions)
    (bool success, ) = msg.sender.call{value: amount_to_withdraw}("");
    require(success, "Transfer failed");

    emit Withdrawn(msg.sender, amount_to_withdraw);
}
```

In this fixed version of the function, the state is updated before the external call is made. This mitigates the risk of reentrancy attacks because the state is updated before an attacker can re-enter the function.

No Boolean Transfer & TransferFrom

```
contract MyTokenFixed {
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;
    uint256 public totalSupply;

    ftrace | funcSig
    function transfer(address to↑, uint256 amount↑) public returns (bool success↑) {
        require(balanceOf[msg.sender] >= amount↑, "Insufficient balance");
        balanceOf[msg.sender] -= amount↑;
        balanceOf[to↑] += amount↑;
        emit Transfer(msg.sender, to↑, amount↑);
        return true;
    }

    ftrace | funcSig
    function transferFrom(address from↑, address to↑, uint256 amount↑) public returns (bool success↑) {
        require(balanceOf[from↑] >= amount↑, "Insufficient balance");
        require(allowance[from↑][msg.sender] >= amount↑, "Insufficient allowance");
        balanceOf[from↑] -= amount↑;
        balanceOf[to↑] += amount↑;
        allowance[from↑][msg.sender] -= amount↑;
        emit Transfer(from↑, to↑, amount↑);
        return true;
    }

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

In this code, the **transfer** and **transferFrom** functions do not return a boolean value to indicate whether the transfer was successful or not. This can cause issues when these functions are called from another smart contract, as the calling contract would not know whether the transfer was successful or not and might fail to handle the error appropriately.

Remediation:
In this fixed code, we have added a **returns (bool success)** statement to both **transfer** and **transferFrom** functions. This will return a boolean value to the calling contract, indicating whether the transfer was successful or not. With this fix in place, the calling contract will be able to detect and handle transfer failures appropriately.

Transfer and transferFrom should not take a fee. Deflationary tokens can lead to unexpected behavior.

```
pragma solidity ^0.8.0;

uml
interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

UnitTest stub | dependencies | uml | draw.io
contract Token {
    string public name = "My Token";
    string public symbol = "MTK";
    uint256 public totalSupply = 1000000 * 10 ** 18;
    uint8 public decimals = 18;
    address public owner;
    uint256 public feePercent = 1; // 1%

    mapping(address => uint256) balances;
    mapping(address => mapping(address => uint256)) allowances;

    ftrace
    constructor() {
        balances[msg.sender] = totalSupply;
        owner = msg.sender;
    }

    ftrace | funcSig
    function transfer(address to↑, uint256 amount↑) external returns (bool) {
        uint256 fee = amount↑ * feePercent / 100;
        balances[msg.sender] -= amount↑;
        balances[to↑] += (amount↑ - fee);
        balances[owner] += fee;
        emit Transfer(msg.sender, to↑, amount↑);
        return true;
    }

    ftrace | funcSig
    function transferFrom(address from↑, address to↑, uint256 amount↑) external returns (bool) {
        uint256 fee = amount↑ * feePercent / 100;
        allowances[from↑][msg.sender] -= amount↑;
        balances[from↑] -= amount↑;
        balances[to↑] += (amount↑ - fee);
        balances[owner] += fee;
        emit Transfer(from↑, to↑, amount↑);
        return true;
    }

    ftrace | funcSig
    function approve(address spender↑, uint256 amount↑) external returns (bool) {
        allowances[msg.sender][spender↑] = amount↑;
        emit Approval(msg.sender, spender↑, amount↑);
        return true;
    }
```

In this example, the **transfer()** and **transferFrom()** functions both implement a fee. When a transfer is made, the fee is deducted from the amount being transferred, and the remainder is sent to the recipient. The fee is sent to the contract owner.

Remediation:
A better way to implement a fee system is to create a separate function for transferring tokens with a fee, and make it clear to users that this function includes a fee. This ensures that users are aware of the fee and are not surprised when they see fewer tokens in their account. In this updated contract, we have removed the fee calculation and transfer code from the **transfer()** function and created a new function called **transferWithFee()** that includes the fee calculation and transfer code. This makes it clear to users that this function includes a fee, and they will not be surprised when they see fewer tokens in their account after using this function.

```
Report | graph (this) | graph | inheritance | parse | flatten | funcSigs | dump | draw.io
    function transferWithFee(address to, uint256 amount) external returns (bool) {
        uint256 fee = amount * feePercent / 100;
        balances[msg.sender] -= amount;
        balances[to] += (amount - fee);
        balances[owner] += fee;
        emit Transfer(msg.sender, to, amount);
        return true;
    }
```

2

ERC20 approve race-condition:

In the vulnerable ERC20 contract, the **approve** function allows an address to approve another address to spend a certain number of tokens on its behalf. The **transferFrom** function allows the approved address to transfer tokens from the original address to another address.

However, there is a race condition in the **transferFrom** function. If the approved address calls **approve** again with a higher allowance before calling **transferFrom**, then the original allowance is overwritten and the approved address can transfer more tokens than intended.

```
// Vulnerable ERC20 contract
UnitTest stub | dependencies | uml | draw.io
contract VulnerableERC20 {
  mapping(address => uint) balances;
  mapping(address => mapping(address => uint)) allowed;

  ftrace | funcSig
  function approve(address _spender↑, uint _value↑) public returns (bool) {
    allowed[msg.sender][_spender↑] = _value↑;
    emit Approval(msg.sender, _spender↑, _value↑);
    return true;
  }
  ftrace | funcSig
  function transferFrom(address _from↑, address _to↑, uint _value↑) public returns (bool) {
    require(_value↑ <= balances[_from↑]);
    require(_value↑ <= allowed[_from↑][msg.sender]);
    balances[_from↑] -= _value↑;
    balances[_to↑] += _value↑;
    allowed[_from↑][msg.sender] -= _value↑;
    emit Transfer(_from↑, _to↑, _value↑);
    return true;
  }
}
// Fix for the ERC20 approve race-condition vulnerability
UnitTest stub | dependencies | uml | draw.io
contract FixedERC20 {
  mapping(address => uint) balances;
  mapping(address => mapping(address => uint)) allowed;

  ftrace | funcSig
  function approve(address _spender↑, uint _value↑) public returns (bool) {
    allowed[msg.sender][_spender↑] = _value↑;
    emit Approval(msg.sender, _spender↑, _value↑);
    return true;
  }
  ftrace | funcSig
  function transferFrom(address _from↑, address _to↑, uint _value↑) public returns (bool) {
    require(_value↑ <= balances[_from↑]);
    require(_value↑ <= allowed[_from↑][msg.sender]);
    require(allowed[_from↑][msg.sender] > 0); // Add check for non-zero allowance
    balances[_from↑] -= _value↑;
    balances[_to↑] += _value↑;
    allowed[_from↑][msg.sender] -= _value↑;
    emit Transfer(_from↑, _to↑, _value↑);
    return true;
  }
```

Remediation**:**
To fix this vulnerability, a check for a non-zero allowance should be added to the transferFrom function. This ensures that the approved address cannot transfer tokens if the allowance has been set to zero in a subsequent call to approve.
The fix code checks for the allowance being greater than zero before transferring tokens, preventing any token theft attempts:

require(allowed[_from][msg.sender] > 0); // Add check for non-zero allowance

Conducting a hypothetical stress test to assess the robustness of the Fortephy API.
Additionally, the hypothetical stress test conducted on the Fortephy API resulted in its failure. The stress test involved sending a high volume of simultaneous audit requests, with varying smart contract complexities, to evaluate the API's ability to handle high loads and maintain stability.

Optimize the Fortephy API to handle high traffic loads and ensure stability during stress tests, such as the hypothetical scenario outlined above.