Protocol Audit Report

ETH Scorpion

July 7, 2025

Protocol Audit Report

Version 1.0

ETH Scorpion

July 7, 2025

Prepared by: ETH Scorpion

Lead Security Researcher:

- Therock Ani

Table of Contents

- [I-5] Use of magic numbers in `calculateReward` is discouraged
- [I-6] No reward mechanism implemented for `calculateReward`
- [I-7] Use `external` instead of `public` for non-internally used functions
- [I-8] Avoid `require` in loops in `voteAndExecute`
- [I-9] Use custom errors for gas optimization
- [I-10] Missing reentrancy guard in `deposit`

Protocol Summary

The `MultiVulnerableVault` contract is designed to allow users to deposit and withdraw Ether, with a lock period and a maximum user cap. It includes features like whitelisting users, an emergency stop mechanism, administrative withdrawal by the owner, and a signature-based fund recovery function. The contract aims to manage user funds securely but contains several vulnerabilities that undermine its security and functionality.

The protocol should do the following: 1. Allow users to deposit Ether (`deposit`) with a minimum deposit of 1 ETH, subject to a maximum user cap of 100. 2. Allow users to withdraw funds (`withdraw`) after a 7-day lock period. 3. Restrict critical functions like `adminWithdraw` and `voteAndExecute` to the owner. 4. Allow the owner to toggle an emergency stop (`toggleEmergency`) to pause deposits and withdrawals. 5. Provide a signature-based recovery mechanism (`recoverFunds`) for the owner to withdraw all funds. 6. Calculate rewards (`calculateReward`) based on deposited amounts and time periods.

Disclaimer

The ETH Scorpion team has made every effort to identify vulnerabilities in the `MultiVulnerableVault` contract within the given time period. However, this audit is not an endorsement of the underlying business or product. The audit was time-boxed and focused solely on the security aspects of the Solidity implementation. The findings are based on the provided code and test suite, and the team holds no responsibility for unresolved issues or future modifications.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity.

Audit Details

Audit Scope Details

Scope

```
./src/
#-- MultiVulnerableVault.sol
```

Roles

- **Owner**: Deployer of the contract, with the ability to withdraw funds via `adminWithdraw`, execute arbitrary calls via `voteAndExecute`, and recover funds via `recoverFunds`.
- **User**: Participants who can deposit and withdraw Ether, subject to a lock period and user cap.
- **Attacker**: Malicious actor exploiting vulnerabilities such as signature replay or arbitrary call execution.

Executive Summary

This audit report, prepared by ETH Scorpion on July 7, 2025, evaluates the security and functionality of the `MultiVulnerableVault` smart contract. The audit, led by Therock Ani, identified critical vulnerabilities that could lead to fund loss, denial-of-service attacks, and restricted interoperability. The test suite (`MultiVulnerableVaultTest.t.sol`) provided comprehensive evidence of these issues, and the audit notes in the contract highlighted additional concerns. Addressing these vulnerabilities is critical to ensure the contract's security and reliability.

Issues Found

Severity	Number of Issues Found
High	4
Medium	3
Low	2
Gas	2
Info	10
Total	21

Findings

High

[H-1] Owner can drain vault funds via `adminWithdraw`

Description: The `adminWithdraw` function allows the owner to withdraw any amount of Ether from the contract to any address without restrictions. This centralizes control and violates trust assumptions, as the owner can drain user funds at any time.

```
function adminWithdraw(address target, uint256 amount) external onlyOwner {
    (bool sent,) = target.call{ value: amount }("");
    require(sent, "Transfer failed");
}
```

Impact: Users' deposited funds are at risk of being stolen by the owner, undermining the contract's purpose as a secure vault.

Proof of Concept: 1. User deposits 2 ETH into the vault. 2. Owner calls `adminWithdraw(OWNER, 2 ether)`. 3. The vault's entire balance is transferred to the owner, leaving users with no recourse.

Proof of Code:

```
function testOwnerCanDrainFunds() public {
    vm.prank(user1);
    vault.deposit{ value: 2 ether }();
    uint256 before = OWNER.balance;
    vm.prank(OWNER);
    vault.adminWithdraw(OWNER, 2 ether);
    assertEq(OWNER.balance, before + 2 ether, "Owner balance should increase by 2 ETH");
}
```

Recommended Mitigation: - Remove the `adminWithdraw` function to prevent centralized control. - Alternatively, implement a multi-signature or timelock mechanism to restrict owner withdrawals and ensure transparency.

[H-2] Signature replay vulnerability in `recoverFunds` allows unauthorized fund drainage

Description: The `recoverFunds` function allows anyone with a valid owner signature to withdraw the entire contract balance. The signature is reusable, enabling replay attacks. Additionally, the hardcoded message hash (`"RECOVER"`) makes it easy for attackers to obtain a valid signature.

```
function recoverFunds(bytes memory signature) external {
    bytes32 message = keccak256(abi.encodePacked("RECOVER"));
    address signer = _recoverSigner(message, signature);
    if (signer == owner) {
        payable(msg.sender).transfer(address(this).balance);
    }
}
```

Impact: An attacker can reuse a valid signature to drain the vault multiple times, stealing all user funds. Also, Owner can drain vault with it.

Proof of Concept: 1. User deposits 5 ETH. 2. Attacker obtains a valid owner signature for the `"RECOVER"` message. 3. Attacker calls `recoverFunds` with the signature, draining 5 ETH. 4. Another user deposits 3 ETH. 5. Attacker reuses the same signature to drain 3 ETH.

Proof of Code:

```
function testSignatureReplayAttack() public {
    vm.prank(user1);
    vault.deposit{ value: 5 ether }();
    vm.prank(attacker);
    vault.recoverFunds(signature);
    assertEq(attacker.balance, 15 ether, "Attacker balance should increase by 5 ETH");
    vm.prank(user2);
    vault.deposit{ value: 3 ether }();
    vm.prank(attacker);
    vault.recoverFunds(signature);
    assertEq(attacker.balance, 18 ether, "Attacker balance should increase by 3 ETH");
}
```

Recommended Mitigation: - Implement a nonce or timestamp in the signed message to prevent replay attacks. - Use OpenZeppelin's ECDSA library with a unique message hash per transaction. - Restrict `recoverFunds` to the owner using the `onlyOwner` modifier. - Better remove it because it poses a centralization risk to users.

[H-3] `voteAndExecute` allows owner to execute arbitrary calls, enabling malicious actions

Description: The `voteAndExecute` function allows the owner to execute arbitrary calls to any contract, including malicious ones, without proper validation or voting mechanisms.

```
function voteAndExecute(address[] memory targets, bytes[] memory data) external onlyOwner {
    require(targets.length == data.length, "Invalid input");
    for (uint256 i = 0; i < targets.length; i++) {
        (bool success,) = targets[i].call(data[i]);
        require(success, "Call failed");
    }
}
```

Impact: A compromised owner or malicious call can manipulate the vault's state (e.g., toggle `emergencyStop`) or interact with external contracts to drain funds.

Proof of Concept: 1. Deploy a malicious contract with a function to toggle the vault's `emergencyStop`. 2. Owner calls `voteAndExecute` with the malicious contract's address and encoded `attack` function. 3. The vault's `emergencyStop` is toggled, disrupting normal operation.

Proof of Code:

```
function testArbitraryCallViaVoteAndExecute() public {
    MaliciousContract malicious = new MaliciousContract();
    address[] memory targets = new address[](1);
    bytes[] memory data = new bytes[](1);
    targets[0] = address(malicious);
    data[0] = abi.encodeWithSignature("attack(address)", address(vault));
    vm.prank(OWNER);
    vault.voteAndExecute(targets, data);
    assertEq(vault.emergencyStop(), true, "Emergency stop should be toggled");
}
```

Recommended Mitigation: - Remove the `voteAndExecute` function or implement a multi-signature voting mechanism.

[H-4] Anybody can toggle emergency stop, enabling denial-of-service attacks

Description: The `toggleEmergency` function lacks access control, allowing any user to toggle the `emergencyStop` state, which pauses deposits and withdrawals.

```
function toggleEmergency() external {
    emergencyStop = !emergencyStop;
}
```

Impact: An attacker can toggle `emergencyStop` to true, preventing users from depositing or withdrawing funds, effectively causing a denial-of-service (DoS) attack.

Proof of Concept: 1. Attacker calls `toggleEmergency`. 2. `emergencyStop` is set to `true`, blocking all deposits and withdrawals.

Proof of Code:

```
function testAnyoneCanToggleEmergencyStop() public {
    vm.prank(attacker);
    vault.toggleEmergency();
    assertEq(vault.emergencyStop(), true, "Emergency stop should be toggled");
}
```

Recommended Mitigation: - Restrict `toggleEmergency` to the owner using the `onlyOwner` modifier. - Use OpenZeppelin's `Pausable` contract for secure pause functionality.

```
- function toggleEmergency() external {
+ function toggleEmergency() external onlyOwner {
    emergencyStop = !emergencyStop;
}
```

Medium

[M-1] Non-reverting transfer failure in `withdraw` leads to loss of funds

Description: The `withdraw` function does not revert if the Ether transfer fails, updating the user's balance and `totalLocked` without transferring funds. This can trap funds in the contract.

```
function withdraw(uint256 amount) external notEmergency {
    User storage user = users[msg.sender];
    require(user.balance >= amount, "Insufficient balance");
    require(block.timestamp >= user.lastDepositTime + LOCK_PERIOD, "Funds locked");
    user.balance -= amount;
    totalLocked -= amount;
    (bool sent,) = msg.sender.call{ value: amount }("");
    if (!sent) {
        emit TransferFailed(msg.sender, amount);
    }
}
```

Impact: Users may lose funds if their contract lacks a `receive` or `fallback` function, as the state is updated without transferring Ether.

Proof of Concept: 1. A `NoReceiveContract` deposits 2 ETH. 2. After the lock period, it calls `withdraw(1 ether)`. 3. The transfer fails, but `user.balance` and `totalLocked` are reduced, trapping 1 ETH in the vault.

Proof of Code:

```
function testStateChangesNotRevertedOnTransferFailure() public {
    vm.prank(address(newNoReceiveContract));
    vault.deposit{ value: 2 ether }();
    vm.warp(block.timestamp + vault.LOCK_PERIOD() + 1);
    vm.prank(address(newNoReceiveContract));
    vault.withdraw(1 ether);
    (uint256 newNoReceiveBal,,) = vault.users(address(newNoReceiveContract));
    assertEq(newNoReceiveBal, 1 ether, "NoReceiveContract balance should be 1 ETH");
    assertEq(address(vault).balance, 4 ether, "Vault balance should remain 4 ETH");
}
```

Recommended Mitigation: - Revert the transaction if the transfer fails to ensure state consistency.

```
    (bool sent,) = msg.sender.call{ value: amount }("");
-   if (!sent) {
-       emit TransferFailed(msg.sender, amount);
-   }
+   require(sent, "Transfer failed");
```

[M-2] External call in `deposit` restricts contract interoperability

Description: The `deposit` function includes an unnecessary external call to `msg.sender` with zero value, which fails for contracts without a `receive` or `fallback` function, preventing them from depositing.

```
function deposit() external payable notEmergency {
    require(msg.value >= MIN_DEPOSIT, "Deposit too small");
    require(userAddresses.length < MAX_USERS, "Max users reached");
    ...
    (bool success,) = msg.sender.call{ value: 0 }("");
    require(success, "Callback failed");
}
```

Impact: Legitimate contracts (e.g., DeFi protocols or DAOs) without `receive`/`fallback` functions cannot deposit, limiting interoperability.

Proof of Concept: 1. A `NoReceiveContract` attempts to deposit 2 ETH. 2. The transaction reverts due to the failed callback. 3. A `ReceiveContract` with a `receive` function succeeds.

Proof of Code:

```
function testDepositCallbackEffect() public {
    vm.prank(address(noReceiveContract));
    vm.expectRevert("Callback failed");
    vault.deposit{ value: 2 ether }();
    vm.prank(address(receiveContract));
    vault.deposit{ value: 2 ether }();
}
```

Recommended Mitigation: - Remove the unnecessary callback.

```
- (bool success,) = msg.sender.call{ value: 0 }("");
- require(success, "Callback failed");
```

[M-3] Maximum user cap prevents whitelisted users from depositing

Description: The `deposit` function enforces a `MAX_USERS` limit of 100, preventing even whitelisted users from depositing once the cap is reached.

```
require(userAddresses.length < MAX_USERS, "Max users reached");
```

Impact: Whitelisted users cannot deposit after the cap is reached, limiting the contract's usability and fairness.

Proof of Concept: 1. 100 users deposit, reaching the `MAX_USERS` limit. 2. A whitelisted user attempts to deposit and is reverted.

Proof of Code:

```
function testMaxUserDepositFailsAfterCap() public {
    for (uint256 i = 0; i < vault.MAX_USERS(); i++) {
        address newUser = vm.addr(i + 4);
        vm.deal(newUser, 2 ether);
        vm.prank(newUser);
        vault.deposit{ value: 1 ether }();
    }
    vm.prank(user1);
    vm.expectRevert("Max users reached");
    vault.deposit{ value: 1 ether }();
}
```

Recommended Mitigation: - Create a separate function for whitelisted users to bypass the `MAX_USERS` check. - Use a counter instead of an array to track users, reducing gas costs and allowing flexibility.

```
+ function depositWhitelisted() external payable notEmergency {
+     require(users[msg.sender].isWhitelisted, "Not whitelisted");
+     require(msg.value >= MIN_DEPOSIT, "Deposit too small");
+     ...
+ }
```

Low

[L-1] Precision loss in `calculateReward` due to division before multiplication

Description: The `calculateReward` function performs division before multiplication, leading to precision loss due to Solidity's integer division truncation.

```
function calculateReward(uint256 amount, uint256 periods) public pure returns (uint256) {
    uint256 reward = amount / 1000;
    return reward * periods;
}
```

Impact: Small deposits result in zero or incorrect rewards, affecting user expectations.

Proof of Concept: 1. Call `calculateReward(999, 10)`; returns 0 due to `999 / 1000 = 0`. 2. Expected result: `(999 * 10) / 1000 = 9`.

Proof of Code:

```
function testCalculateRewardPrecisionLoss() public view {
    uint256 reward = vault.calculateReward(999, 10);
    assertEq(reward, 0, "Reward should be 0 due to truncation");
    uint256 expectedReward = (999 * 10) / 1000;
    assertEq(expectedReward, 9, "Expected reward should be 9");
}
```

Recommended Mitigation: - Perform multiplication before division.

```
- uint256 reward = amount / 1000;
- return reward * periods;
+ return (amount * periods) / 1000;
```

[L-2] Incorrect re-whitelisting after withdrawals in `deposit`

Description: The `deposit` function re-whitelists users if their balance is zero, even if they were previously whitelisted and withdrew funds.

```
if (user.balance == 0) {
    user.isWhitelisted = true;
    userAddresses.push(msg.sender);
}
```

Impact: Users who withdraw all funds are incorrectly re-whitelisted, potentially bypassing intended restrictions.

Recommended Mitigation: - Check `isWhitelisted` instead of `balance`.

```
- if (user.balance == 0) {
+ if (!user.isWhitelisted) {
    user.isWhitelisted = true;
    userAddresses.push(msg.sender);
}
```

Gas

[G-1] Unchanged state variables should be declared constant or immutable

Description: State variable `owner` is not modified after initialization and should be declared `immutable` to save gas.

Instances: - `owner` should be `immutable`.

Recommended Mitigation:

```
- address public owner;
+ address public immutable owner;
```

[G-2] Use of array for user tracking leads to potential DoS

Description: The `userAddresses` array grows with each new user, increasing gas costs for deposits and risking a DoS attack as the array size approaches `MAX_USERS`.

```
require(userAddresses.length < MAX_USERS, "Max users reached");
userAddresses.push(msg.sender);
```

Impact: High gas costs discourage deposits, and a malicious actor could fill the array to prevent further deposits.

Recommended Mitigation: - Use a counter or mapping to track users.

```
+ uint256 public userCount;
- address[] public userAddresses;
- require(userAddresses.length < MAX_USERS, "Max users reached");
+ require(userCount < MAX_USERS, "Max users reached");
- userAddresses.push(msg.sender);
+ userCount++;
```

Informational/Non-Crits

[I-1] Solidity pragma should be specific, not wide

Description: The pragma `^0.8.17` allows a range of versions, which may introduce vulnerabilities or incompatibilities.

Recommended Mitigation: - Use a specific version, e.g., `pragma solidity 0.8.20`.

[I-2] Using a vulnerable version of Solidity is not recommended

Description: Solidity `^0.8.17` has known issues (e.g., `VerbatimInvalidDeduplication`, `FullInlinerNonExpressionSplitArgumentEvaluationOrder`, `MissingSideEffectsOnSelectorAccess`).

Recommended Mitigation: - Upgrade to `0.8.20` or later, ensuring compatibility with the target EVM.

[I-3] Use of assembly in `_splitSignature` is error-prone

Description: The `_splitSignature` function uses assembly, increasing the risk of errors.

Recommended Mitigation: - Use OpenZeppelin's ECDSA library to parse signatures safely.

[I-4] `ecrecover` is susceptible to signature malleability

Description: The `ecrecover` function in `_recoverSigner` is vulnerable to signature malleability.

Recommended Mitigation: - Use OpenZeppelin's ECDSA library to mitigate malleability risks.

[I-5] Use of magic numbers in `calculateReward` is discouraged

Description: The number `1000` in `calculateReward` lacks context.

Recommended Mitigation:

```
+ uint256 public constant REWARD_DIVISOR = 1000;
- uint256 reward = amount / 1000;
+ uint256 reward = amount / REWARD_DIVISOR;
```

[I-6] No reward mechanism implemented for `calculateReward`

Description: The `calculateReward` function is not used in the contract, making its purpose unclear.

Recommended Mitigation: - Implement a reward mechanism or remove the function.

[I-7] Use `external` instead of `public` for non-internally used functions

Description: Functions like `calculateReward` are `public` but not used internally.

Recommended Mitigation:

```
- function calculateReward(uint256 amount, uint256 periods) public pure returns (uint256) {
+ function calculateReward(uint256 amount, uint256 periods) external pure returns (uint256)
```

[I-8] Avoid `require` in loops in `voteAndExecute`

Description: The `require(success, "Call failed")` in a loop can cause the entire transaction to revert if one call fails.

Recommended Mitigation: - Log failed calls and continue processing.

```
    for (uint256 i = 0; i < targets.length; i++) {
        (bool success,) = targets[i].call(data[i]);
-       require(success, "Call failed");
+       if (!success) emit CallFailed(targets[i], data[i]);
    }
```

[I-9] Use custom errors for gas optimization

Description: Using `require` with strings is gas-intensive compared to custom errors.

Recommended Mitigation:

```
+ error NotOwner();
+ error EmergencyStopActivated();
- require(msg.sender == owner, "Not owner");
+ revert NotOwner();
```

[I-10] Missing reentrancy guard in `deposit`

Description: The `deposit` function lacks a reentrancy guard, risking reentrancy attacks via the external callback.

Recommended Mitigation: - Use OpenZeppelin's `ReentrancyGuard`.