Protocol Audit Report

Version 1.0

khal45

November 30, 2025

Vault Guardians Audit Report

khal45

17th November, 2025

Prepared by: khal45 Lead Auditors:

- khal45

Table of Contents

    * [L-1] `VaultShares::deposit` has the `nonReentrant` modifier as the second modifier opening up the deposit function to reentrancy attacks if the first modifier has any state changes
    * [L-2]: Unsafe ERC20 Operation
    * [L-3]: Unchecked Return
- Informational
    * [I-1] `VaultGuardiansBase::becomeGuardian` doesn't collect any fee with the transaction
    * [I-2]: Public Function Not Used Internally
    * [I-3]: Unused Error
    * [I-4]: Unused State Variable
    * [I-5]: Unused Import

Protocol Summary

This protocol allows users to deposit certain ERC20s into an ERC4626 vault managed by a human being, or a `vaultGuardian`. The goal of a `vaultGuardian` is to manage the vault in a way that maximizes the value of the vault for the users who have despoited money into the vault.

Disclaimer

Khal45 makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

```
./src/abstract
./src/dao
./src/interfaces
./src/protocol
./src/vendor
```

Roles

- Vault Guardian: The person who manages user vaults
- Users: People who deposit funds into the protocol

Executive Summary

Over the course of the security review, khal45 engaged with the vault guardians protocol to review it. In this period of time a total of 10 issues were found

Issues found

Severity	Number of Issues Found
High	2
Low	3
Informational	5
Total	10

Findings

High

[H-1] Vault Guardians can call `VaultShares::redeem` and redeem their stake price without quiting

Description: Vault Guardians are only meant to be able to withdraw their stake price when they quit being a guardian according to the README `If a guardian wishes to stop being a guardian, they give out all user deposits and withdraw their guardian stake.` However, when a guardian becomes a guardian they either call `VaultGuardiansBase::becomeGuardian` or `VaultGuardiansBase::becomeTokenGuardian` which both call `VaultGuardiansBase::_becomeTokenGuardian`. `_becomeTokenGuardian` transfers the stake price from the user and deposits it into the guardian's newly created vault

```
1      function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
           private returns (address) {
2
3          s_guardians[msg.sender][token] = IVaultShares(address(
               tokenVault));
4          emit GuardianAdded(msg.sender, token);
5          i_vgToken.mint(msg.sender, s_guardianStakePrice);
6          token.safeTransferFrom(msg.sender, address(this),
               s_guardianStakePrice);
7          bool succ = token.approve(address(tokenVault),
               s_guardianStakePrice);
8          if (!succ) {
9              revert VaultGuardiansBase__TransferFailed();
10         }
11 @>       uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg
       .sender);
12         if (shares == 0) {
13             revert VaultGuardiansBase__TransferFailed();
14         }
15         return address(tokenVault);
16     }
```

This means they in turn get shares of their own vault and since they have shares in their own vault, the can redeem their assets for those shares at any time without quiting

Impact: Vault guardians can redeem their stake price at any time without quiting, thereby still getting rewards without having any stake price locked. A malicious actor could also potentially become a guardian with the same stake price across multiple vaults by redeeming their stake after becoming a guardian and using the same stake to become a guardian for another vault, repeating the process as many times as possible to potentially gain more rewards.

Proof of Concept: Consider the following

1. Guardian calls `becomeGuardian` with the stake price
2. Guardian calls `redeem` on their newly created vault essentially getting back their stake price
3. Users can still deposit since the vault is not closed

Add the following test to `VaultGuardiansBaseTest.t.sol`

Proof of Code

```
    function test_guardian_redeems_stake_price_without_quitting()
        public hasGuardian {
        // we first get the maximum assets the guardian can redeem
        uint256 maxRedeemable = wethVaultShares.maxRedeem(guardian);
        console.log("max redeemable", maxRedeemable);

        uint256 guardianInitialWethBalance = weth.balanceOf(guardian);
        console.log("guardian initial weth balance",
            guardianInitialWethBalance);

        vm.prank(guardian);
        // normally calling `quitGuardian` redeems the `maxRedeemable`
            and closes the vault but since there's no
        // restriction on redeeming, the guardian can redeem without
            closing the vault
        wethVaultShares.redeem(maxRedeemable, guardian, guardian);

        // guardian final balance increases but vault is still active
        uint256 guardianFinalWethBalance = weth.balanceOf(guardian);
        console.log("guardian final weth balance",
            guardianFinalWethBalance);

        assertGt(guardianFinalWethBalance, guardianInitialWethBalance);
        assertTrue(wethVaultShares.getIsActive());

        // users can still deposit into the vault
        weth.mint(mintAmount, user);
        vm.startPrank(user);
        weth.approve(address(wethVaultShares), mintAmount);
        wethVaultShares.deposit(mintAmount, user);
        vm.stopPrank();
    }
```

Recommended Mitigation: Don't deposit the stake price into the guardians's vault but leave the stake price in the vault guardians contract and transfer the stake price to the guardian when they call `quitGuardian`

[H-2] `UniswapAdapter::_uniswapInvest` double-counts swapped tokens causing vault operations to fail with aggressive allocations

Description: `UniswapAdapter::_uniswapInvest` attempts to add liquidity using the full original allocation amount plus the swap input amount, effectively double-counting the tokens that were already consumed during the swap. After swapping half of the allocated tokens for the counterparty token, the function tries to provide liquidity using `amountOfTokenToSwap + amounts[0]` where `amounts[0]` equals the amount already spent in the swap (`amountOfTokenToSwap`). This causes the function to request `1.5x` the allocated tokens instead of the remaining `0.5x`.

```
function _uniswapInvest(IERC20 token, uint256 amount) internal {
    IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
    uint256 amountOfTokenToSwap = amount / 2;
    s_pathArray = [address(token), address(counterPartyToken)];

    // Approve and swap half the allocation
    bool succ = token.approve(address(i_uniswapRouter),
        amountOfTokenToSwap);
    if (!succ) {
        revert UniswapAdapter__TransferFailed();
    }

    uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens
        ({
        amountIn: amountOfTokenToSwap,  // Swaps 25 WETH (consumes from
            vault balance)
        amountOutMin: 0,
        path: s_pathArray,
        to: address(this),
        deadline: block.timestamp
    });
    // amounts[0] = 25 WETH (the input amount that was just consumed)

    succ = counterPartyToken.approve(address(i_uniswapRouter), amounts
        [1]);
    if (!succ) {
        revert UniswapAdapter__TransferFailed();
    }

    // BUG: Approves 25 + 25 = 50 WETH (but vault only has 25 WETH
        remaining!)
@>  succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap
    + amounts[0]);
    if (!succ) {
        revert UniswapAdapter__TransferFailed();
    }

    // BUG: Tries to use 50 WETH for liquidity when only 25 WETH
        remains
```

```
33      (uint256 tokenAmount, uint256 counterPartyTokenAmount, uint256
            liquidity) = i_uniswapRouter.addLiquidity({
34          tokenA: address(token),
35          tokenB: address(counterPartyToken),
36  @>      amountADesired: amountOfTokenToSwap + amounts[0],  // 25 + 25 =
        50 WETH requested
37          amountBDesired: amounts[1],
38          amountAMin: 0,
39          amountBMin: 0,
40          to: address(this),
41          deadline: block.timestamp
42      });
43      emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
            liquidity);
44  }
```

The bug was masked in the original tests because they used a 25% Uniswap allocation with a 50% hold buffer, providing enough unused tokens in the vault to cover the double-spend. However, with allocations requiring more than 50% Uniswap investment (or any scenario where other protocols consume tokens first), the vault has insufficient balance to fulfill the inflated liquidity request.

Impact:

1. **Protocol Design Failure**: Vaults cannot be created with aggressive investment strategies (greater than or equal to 50% Uniswap, less than or equal to 50% hold). The `becomeGuardian` function will revert during the initial stake deposit, making it impossible to deploy such vaults.
2. **Permanent Fund Lock**: Guardians who update existing vaults from conservative to aggressive allocations will permanently lock all user funds. Any operation using the `divestThenInvest` modifier (`withdraw`, `redeem`, `rebalanceFunds`) will revert with `ERC20InsufficientBalance`, preventing users from accessing their assets.
3. **Deposit Failures**: Even if a vault is successfully created with aggressive allocation, all subsequent deposits will fail, rendering the vault unusable.
4. **Silent Over-Investment**: For vaults with low Uniswap allocations that don't trigger reverts, the bug causes silent over-investment in Uniswap (using 1.5x the intended amount), breaking the guardian's intended allocation strategy and potentially exposing users to unintended risk profiles.

Proof of Concept:

Test Case 1: Cannot Create Vault with Aggressive Allocation

Add the following test to `VaultSharesTest.t.sol`:

```
1  function test_CannotEvenCreateVaultWithHighUniswapAllocation() public {
```

```
2       // ===== SETUP: Try to create vault with allocation that triggers
            the bug =====
3       // Use 50% Uniswap, 50% Aave (no hold buffer to mask the issue)
4       AllocationData memory triggeringAllocation = AllocationData({
5           holdAllocation: 0,      // 0% - no buffer to hide the bug
6           uniswapAllocation: 500, // 50% - high enough to trigger failure
7           aaveAllocation: 500     // 50% - will consume half the balance
                first
8       });
9
10      console.log("\n=== ATTEMPTING TO CREATE VAULT ===");
11      console.log("Allocation: Hold: 0%, Uniswap: 50%, Aave: 50%");
12      console.log("\nbecomeGuardian() automatically deposits guardian
            stake price");
13      console.log("This deposit will trigger _investFunds, which will
            fail due to the bug");
14
15      // Try to create guardian with the triggering allocation
16      weth.mint(mintAmount, guardian);
17      vm.startPrank(guardian);
18      weth.approve(address(vaultGuardians), mintAmount);
19
20      console.log("\n=== EXPECTED FAILURE ===");
21      console.log("Guardian deposits 10 WETH (stake price)");
22      console.log("  1. _aaveInvest uses 5 WETH (vault has 5 WETH left)")
            ;
23      console.log("  2. _uniswapInvest allocated 5 WETH");
24      console.log("     - Swaps 2.5 WETH (vault has 2.5 WETH left)");
25      console.log("     - Tries to add liquidity with 5 WETH (2.5 + 2.5
            double-count)");
26      console.log("     - FAILS: vault only has 2.5 WETH!");
27
28      // becomeGuardian will revert because the initial deposit fails
29      vm.expectRevert(); // ERC20InsufficientBalance error
30      vaultGuardians.becomeGuardian(triggeringAllocation);
31      vm.stopPrank();
32
33      console.log("\n=== IMPACT ===");
34      console.log("CRITICAL: Cannot even CREATE a vault with aggressive
            allocation!");
35      console.log("Any vault requiring >=50% Uniswap with <=50% hold is
            impossible to deploy");
36      console.log("This completely breaks the protocol's investment
            strategy flexibility");
37
38      console.log("\n=== ALLOCATION THAT TRIGGERS BUG ===");
39      console.log("Hold: 0%, Uniswap: 50%, Aave: 50%");
40      console.log("This means no buffer to mask the double-spend bug");
41
42      console.log("\n=== VERIFICATION ===");
43      console.log("Guardian was unable to create the vault");
```

```
        console.log("The protocol CANNOT support aggressive investment
            strategies");
    }
```

Test Case 2: Updating Allocation Permanently Locks User Funds

Add the following test to `VaultSharesTest.t.sol`:

```
function test_RedeemFailsDueToUniswapDoubleSpend() public hasGuardian
    userIsInvested {
    // ===== SETUP: Record initial state =====
    address vaultGuardian = wethVaultShares.getVaultGuardians();
    uint256 userMaxRedeemable = wethVaultShares.maxRedeem(user);
    uint256 userBalanceBeforeRedeem = weth.balanceOf(user);

    AllocationData memory previousAllocation = wethVaultShares.
        getAllocationData();

    console.log("\n=== INITIAL STATE ===");
    console.log("Vault WETH Balance:", weth.balanceOf(address(
        wethVaultShares)));
    console.log("User Redeemable Shares:", userMaxRedeemable);
    console.log(
        "Previous Allocation - Uniswap: %s%, Aave: %s%",
        previousAllocation.uniswapAllocation / 10,
        previousAllocation.aaveAllocation / 10
    );

    // ===== ACTION 1: Guardian updates allocation strategy =====
    // Change from balanced (25/25/50) to aggressive (50/50/0) strategy
    vm.prank(vaultGuardian);
    wethVaultShares.updateHoldingAllocation(newAllocationData);

    AllocationData memory newAllocation = wethVaultShares.
        getAllocationData();

    console.log("\n=== ALLOCATION UPDATED ===");
    console.log(
        "New Allocation - Uniswap: %s%, Aave: %s%",
        newAllocation.uniswapAllocation / 10,
        newAllocation.aaveAllocation / 10
    );

    // Verify allocation actually changed
    assertNotEq(
        newAllocation.uniswapAllocation,
        previousAllocation.uniswapAllocation,
        "Uniswap allocation should have changed"
    );
    assertNotEq(
        newAllocation.aaveAllocation,
```

```
40          previousAllocation.aaveAllocation,
41          "Aave allocation should have changed"
42      );
43
44      // ===== ACTION 2: User attempts redemption =====
45      // The divestThenInvest modifier will:
46      // 1. Divest all positions (Uniswap LP + Aave aTokens)
47      // 2. Process user's redemption (burning shares, transferring
            assets)
48      // 3. Calculate remaining balance and attempt to reinvest per new
            allocation
49
50      console.log("\n=== USER REDEMPTION ATTEMPT ===");
51      console.log("Attempting to redeem:", userMaxRedeemable, "shares");
52
53      vm.prank(user);
54      vm.expectRevert(); // Expect ERC20InsufficientBalance error
55      wethVaultShares.redeem(userMaxRedeemable, user, user);
56
57      // ===== VERIFICATION: User did not receive funds =====
58      uint256 userBalanceAfterFailedRedeem = weth.balanceOf(user);
59
60      console.log("\n=== IMPACT ===");
61      console.log("User balance before redeem:", userBalanceBeforeRedeem)
            ;
62      console.log("User balance after failed redeem:",
            userBalanceAfterFailedRedeem);
63      console.log("Expected to receive ~:", userMaxRedeemable);
64
65      assertEq(
66          userBalanceAfterFailedRedeem,
67          userBalanceBeforeRedeem,
68          "User should not have received any WETH due to failed
                transaction"
69      );
70
71      console.log("\n=== ROOT CAUSE ===");
72      console.log("After redemption, vault had X WETH remaining");
73      console.log("_aaveInvest consumed 50% (X/2) successfully");
74      console.log("_uniswapInvest attempted to:");
75      console.log("  1. Swap 25% (X/4) to counterparty token");
76      console.log("  2. Add liquidity using 50% (X/2) - BUT ONLY X/4
            REMAINING");
77      console.log("Result: ERC20InsufficientBalance error");
78  }
```

Recommended Mitigation:

Remove the double-counting of the swap input amount. The `addLiquidity` call should only use the remaining half of the allocation that wasn't swapped:

```
function _uniswapInvest(IERC20 token, uint256 amount) internal {
    IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
    uint256 amountOfTokenToSwap = amount / 2;
    s_pathArray = [address(token), address(counterPartyToken)];

    bool succ = token.approve(address(i_uniswapRouter),
        amountOfTokenToSwap);
    if (!succ) {
        revert UniswapAdapter__TransferFailed();
    }

    uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens
        ({
        amountIn: amountOfTokenToSwap,
        amountOutMin: 0,
        path: s_pathArray,
        to: address(this),
        deadline: block.timestamp
    });

    succ = counterPartyToken.approve(address(i_uniswapRouter), amounts
        [1]);
    if (!succ) {
        revert UniswapAdapter__TransferFailed();
    }

-   succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap
    + amounts[0]);
+   succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap)
    ;
    if (!succ) {
        revert UniswapAdapter__TransferFailed();
    }

    (uint256 tokenAmount, uint256 counterPartyTokenAmount, uint256
        liquidity) = i_uniswapRouter.addLiquidity({
        tokenA: address(token),
        tokenB: address(counterPartyToken),
-       amountADesired: amountOfTokenToSwap + amounts[0],
+       amountADesired: amountOfTokenToSwap,
        amountBDesired: amounts[1],
        amountAMin: 0,
        amountBMin: 0,
        to: address(this),
        deadline: block.timestamp
    });
    emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
        liquidity);
}
```

This ensures that after swapping half the allocation, only the remaining half is used for liquidity provision, correctly implementing the intended 50/50 split between the token and its counterparty.

Low

[L-1] `VaultShares::deposit` has the `nonReentrant` modifier as the second modifier opening up the deposit function to reentrancy attacks if the first modifier has any state changes

Modifiers in Solidity execute in the order they are written, so when you have multiple modifiers they run from left to right in the order they are written. If the `isActive` modifier performs any state change or the contract is upgraded and the function includes another modifier that updates state beore `nonReentrant` an attacker could potentially exploit this if they can profit from it

[L-2]: Unsafe ERC20 Operation

ERC20 functions may not behave as expected. For example: return values are not always meaningful. It is recommended to use OpenZeppelin's SafeERC20 library.

5 Found Instances

- Found in src/protocol/VaultGuardiansBase.sol Line: 324

```
1           bool succ = token.approve(address(tokenVault),
               s_guardianStakePrice);
```

- Found in src/protocol/investableUniverseAdapters/AaveAdapter.sol Line: 29

```
1           bool succ = asset.approve(address(i_aavePool), amount);
```

- Found in src/protocol/investableUniverseAdapters/UniswapAdapter.sol Line: 59

```
1           bool succ = token.approve(address(i_uniswapRouter),
               amountOfTokenToSwap);
```

- Found in src/protocol/investableUniverseAdapters/UniswapAdapter.sol Line: 75

```
1           succ = counterPartyToken.approve(address(i_uniswapRouter),
                amounts[1]);
```

- Found in src/protocol/investableUniverseAdapters/UniswapAdapter.sol Line: 81

```
1           succ = token.approve(address(i_uniswapRouter),
               amountOfTokenToSwap + amounts[0]);
```

[L-3]: Unchecked Return

Function returns a value but it is ignored. Consider checking the return value.

3 Found Instances

- Found in src/protocol/VaultShares.sol Line: 81

```
1             _uniswapDivest(IERC20(asset()),
                 uniswapLiquidityTokensBalance);
```

- Found in src/protocol/VaultShares.sol Line: 84

```
1             _aaveDivest(IERC20(asset()), aaveAtokensBalance);
```

- Found in src/protocol/investableUniverseAdapters/AaveAdapter.sol Line: 49

```
1         i_aavePool.withdraw({asset: address(token), amount: amount
              , to: address(this)});
```

Informational

[I-1] `VaultGuardiansBase::becomeGuardian` doesn't collect any fee with the transaction

The natspec says `* @notice they have to send an ETH amount equal to the fee, and a WETH amount equal to the stake price *` but the function is not payable and there isn't a require statement that checks that

[I-2]: Public Function Not Used Internally

If a function is marked public but is not used internally, consider marking it as `external`.

2 Found Instances

- Found in src/protocol/VaultShares.sol Line: 127

```
1     function setNotActive() public onlyVaultGuardians isActive {
```

- Found in src/protocol/VaultShares.sol Line: 207

```
1     function rebalanceFunds() public isActive divestThenInvest
          nonReentrant {}
```

[I-3]: Unused Error

Consider using or removing the unused error.

4 Found Instances

- Found in src/protocol/VaultGuardians.sol Line: 46

```
1       error VaultGuardians__TransferFailed();
```

- Found in src/protocol/VaultGuardiansBase.sol Line: 46

```
1       error VaultGuardiansBase__NotEnoughWeth(uint256 amount,
            uint256 amountNeeded);
```

- Found in src/protocol/VaultGuardiansBase.sol Line: 48

```
1       error VaultGuardiansBase__CantQuitGuardianWithNonWethVaults(
            address guardianAddress);
```

- Found in src/protocol/VaultGuardiansBase.sol Line: 52

```
1       error VaultGuardiansBase__FeeTooSmall(uint256 fee, uint256
            requiredFee);
```

[I-4]: Unused State Variable

State variable appears to be unused. No analysis has been performed to see if any inline assembly references it. Consider removing this unused variable.

1 Found Instances

- Found in src/protocol/VaultGuardiansBase.sol Line: 67

```
1       uint256 private constant GUARDIAN_FEE = 0.1 ether;
```

[I-5]: Unused Import

Redundant import statement. Consider removing it.

1 Found Instances

- Found in src/interfaces/InvestableUniverseAdapter.sol Line: 4

```
1  import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.
       sol";
```