Protocol Audit Report

Version 1.0

Bhanu Sai Enamala

August 30, 2024

Vault Guardians Protocol Audit Report

Bhanu Sai Enamala

August 30, 2024

Prepared by: Bhanu Sai Enamala

Lead Auditors: Bhanu Sai Enamala

Table of Contents

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.

Github Link to the VaultGuardians Protocol: https://github.com/Cyfrin/8-vault-guardians-audit/tree/main

Disclaimer

The Bhanu Sai Enamala team 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 the team 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

		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

Scope

```
./src/
#-- abstract
|   #-- AStaticTokenData.sol
|   #-- AStaticUSDCData.sol
|   #-- AStaticWethData.sol
#-- dao
|   #-- VaultGuardianGovernor.sol
|   #-- VaultGuardianToken.sol
#-- interfaces
|   #-- IVaultData.sol
|   #-- IVaultGuardians.sol
|   #-- IVaultShares.sol
|   #-- InvestableUniverseAdapter.sol
#-- protocol
|   #-- VaultGuardians.sol
|   #-- VaultGuardiansBase.sol
|   #-- VaultShares.sol
|   #-- investableUniverseAdapters
|       #-- AaveAdapter.sol
|       #-- UniswapAdapter.sol
#-- vendor
    #-- DataTypes.sol
    #-- IPool.sol
    #-- IUniswapV2Factory.sol
    #-- IUniswapV2Router01.sol
```

Roles

There are 4 main roles associated with the system.

- *Vault Guardian DAO*: The org that takes a cut of all profits, controlled by the `VaultGuardianToken`. The DAO that controls a few variables of the protocol, including:
  - `s_guardianStakePrice`
  - `s_guardianAndDaoCut`
  - And takes a cut of the ERC20s made from the protocol
- *DAO Participants*: Holders of the `VaultGuardianToken` who vote and take profits on the protocol
- *Vault Guardians*: Strategists/hedge fund managers who have the ability to move assets in and out of the investable universe. They take a cut of revenue from the protocol.
- *Investors*: The users of the protocol. They deposit assets to gain yield from the investments of the Vault Guardians.
-

Executive Summary

The Vault Guardians project takes novel approaches to work ERC-4626 into a hedge fund of sorts, but makes some large mistakes on tracking balances and profits.

Issues found

Severity	Number of issues found
High	7
Medium	1
Low	2
Info	4
Gas	4
Total	18

Findings

High

[H-1] Missing ethereum handling code to receive `VaultGuardiansBase::GUARDIAN_FEE` resulting in loss of funds from Fee collection from a guardian.

{#h-1}

Description: The `vaultGuardiansBase` contract defines a `GUARDIAN_FEE` equal to `0.1 ether` to be collected when someone calls `VaultGuardiansBase::becomeGuardian` function. The `becomeGuardian` function is not a payable function nor the `VaultGuardiansBase` has any special `receive` function to handle the eth.

Impact: Due to this, someone can become guardian without paying the `GUARDIAN_FEE` resulting in financial loss to the `VaultGuardians` contract.

Proof of Concept:

Add the following code to the `VaultGuardiansBaseTest.t.sol` file.

Code

```
function testBecomeGuardianWithoutFee() public {
            uint256 initialBalance = guardian.balance;
            weth.mint(mintAmount, guardian);
            vm.startPrank(guardian);
            weth.approve(address(vaultGuardians), mintAmount);
            address wethVault = vaultGuardians.becomeGuardian(
                allocationData);
            vm.stopPrank();
            uint256 finalBalance = guardian.balance;
            assertEq(finalBalance, initialBalance);
        }
```

Recommended Mitigation:

Make the `VaultGuardiansBase::becomeGuardian` function payable and add a require statement inside the function as follows.

```
    function becomeGuardian(AllocationData memory wethAllocationData)
        external returns (address)
+   payable
    {
+       require(msg.value==GUARDIAN_FEE,"You need to pay 0.1 ether as guardian fee to become a guardian")
        VaultShares wethVault =
        new VaultShares(IVaultShares.ConstructorData({
            asset: i_weth,
            vaultName: WETH_VAULT_NAME,
            vaultSymbol: WETH_VAULT_SYMBOL,
            guardian: msg.sender,
            allocationData: wethAllocationData,
            aavePool: i_aavePool,
            uniswapRouter: i_uniswapV2Router,
            guardianAndDaoCut: s_guardianAndDaoCut,
            vaultGuardians: address(this),
            weth: address(i_weth),
            usdc: address(i_tokenOne)
        }));
        return _becomeTokenGuardian(i_weth, wethVault);
    }
```

[H-2] Incorrect calculation inside `UniswapAdapter::_uniswapInvest` function of the tokens to be deposited in the Uniswap pool leading to allocation of funds between uniswap, Aave and holding different from the `VaultShares::s_allocationData`

{#h-2}

Description: In the `UniswapAdapter::_uniswapInvest` function, after half of the input tokens

are swapped for the counterparty token to facilitate investment on both sides of the Uniswap pool, the `UniswapV2Router01::addLiquidity` function is called to add liquidity. However, once the input tokens are swapped, only half of the initially supplied tokens remain available. Despite this, the `UniswapV2Router01` contract is mistakenly approved to spend an amount equivalent to the full initial supply of tokens, rather than the remaining half. This same incorrect amount is then used in the subsequent `UniswapV2Router01::addLiquidity` call.

As a result, the function inadvertently adds more tokens to the Uniswap protocol than intended. The excess tokens are taken from the holding portion of the allocation data, leading to an unintended reduction in the holding allocation and an over-allocation to the Uniswap pool.

Impact: This issue could lead to an imbalance in the allocation strategy, where more tokens are invested in Uniswap than planned, potentially impacting the overall performance and risk profile of the investment strategy. Additionally, the `VaultShares::s_allocationData` will reflect one strategy on paper, but in reality, a different allocation is being followed without the knowledge of those depositing into the contract. This discrepancy could mislead depositors and undermine the integrity of the investment strategy.

Proof of Concept: Add the following code to the `VaultGuardiansBaseTest.t.sol` file.

Code

```
function testWrongAllocation() public {
        weth.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        weth.approve(address(vaultGuardians), mintAmount);
        address wethVault = vaultGuardians.becomeGuardian(
            allocationData);
        uint256 hold = weth.balanceOf(wethVault);
        vm.stopPrank();
        assert(hold!=(allocationData.holdAllocation)*(mintAmount)
            /(1000));
    }
```

Recommended Mitigation:

Consider updating the `UniswapAdapter::_uniswapInvest` function as follows:

```
 function _uniswapInvest(IERC20 token, uint256 amount) internal {
        IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
            i_weth;
        // We will do half in WETH and half in the token
        uint256 amountOfTokenToSwap = amount / 2;
        // the path array is supplied to the Uniswap router, which
            allows us to create swap paths
        // in case a pool does not exist for the input token and the
            output token
```

```
7           // however, in this case, we are sure that a swap path exists
                for all pair permutations of WETH, USDC and LINK
8           // (excluding pair permutations including the same token type)
9           // the element at index 0 is the address of the input token
10          // the element at index 1 is the address of the output token
11          s_pathArray = [address(token), address(counterPartyToken)];
12
13          bool succ = token.approve(address(i_uniswapRouter),
                amountOfTokenToSwap);
14          if (!succ) {
15              revert UniswapAdapter__TransferFailed();
16          }
17          uint256[] memory amounts = i_uniswapRouter.
                swapExactTokensForTokens({
18              amountIn: amountOfTokenToSwap,
19              amountOutMin: 0,
20              path: s_pathArray,
21              to: address(this),
22              deadline: block.timestamp
23          });
24
25          succ = counterPartyToken.approve(address(i_uniswapRouter),
                amounts[1]);
26          if (!succ) {
27              revert UniswapAdapter__TransferFailed();
28          }
29  -       succ = token.approve(address(i_uniswapRouter),
    amountOfTokenToSwap + amounts[0]);
30  +       succ = token.approve(address(i_uniswapRouter),
    amountOfTokenToSwap);
31          if (!succ) {
32              revert UniswapAdapter__TransferFailed();
33          }
34
35          // amounts[1] should be the WETH amount we got back
36          (uint256 tokenAmount, uint256 counterPartyTokenAmount, uint256
                liquidity) = i_uniswapRouter.addLiquidity({
37              tokenA: address(token),
38              tokenB: address(counterPartyToken),
39  -           amountADesired: amountOfTokenToSwap + amounts[0],
40  +           amountADesired: amountOfTokenToSwap,
41              amountBDesired: amounts[1],
42              amountAMin: 0,
43              amountBMin: 0,
44              to: address(this),
45              deadline: block.timestamp
46          });
47          emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
                liquidity);
48      }
```

[H-3] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables frontrunners to profit from the transaction. (This exact issue is associated with removeLiquidity call inside `UniswapAdapter::_uniswapDivest` function too, so please apply the concept accordingly.A separate writeup is not being provided to avoid duplication fo the same concept).

{#h-3}

Description: In `UniswapAdapter::_uniswapvest` the protocol supplies tokens available in the fucntion to both sides of the pool in Uniswap.It calls the `addLiquidity` function of the `UniswapV2Router01` contract, which has three input parameters to note:

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
@>      uint256 amountAMin,
@>      uint256 amountBMin,
        address to,
@>      uint256 deadline
    ) external returns (uint256 amountA, uint256 amountB, uint256
       liquidity);
```

The parameters `amountAMin` and `amountBMin` represents how much of the minimum number of tokens it expects to put inside the Uniswap pool. The `deadline` parameter represents when the transaction should expire.

As seen below, the `UniswapAdapter::_uniswapInvest` function sets those parameters to `0` and `block.timestamp`:

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

Impact: This lack of slippage protection could result in one of the following scenarios:

- A frontrunning bot sees the transaction in the mempool, uses a flash loan to manipulate prices on Uniswap before the protocol's transaction is executed, leading to the protocol adding liquidity

at an unfavorable rate.

- Without an appropriate deadline, the transaction could be delayed by a node until it becomes advantageous for them to execute, potentially resulting in significant financial losses for the protocol.

Proof of Concept:

1. UniswapAdapter::_uniswapInvest gets called from a function within VaultShares. 1. This triggers the router's addLiquidity function. 2. In the mempool, a malicious user could: 1. Hold onto this transaction which adds liquidity to Uniswap. 2. Use a flash loan to manipulate the prices of assets on Uniswap. 3. Execute the held transaction, resulting in an unfavorable liquidity addition for the protocol due to the amountAMin and amountBMin values set to 0.

This could allow MEV attackers and frontrunners to exploit the vault, leading to potential financial losses.

Recommended Mitigation:

For the deadline issue, we recommend the following:

Given the diverse nature of DeFi protocols, it's important to include a custom parameter in the `withdraw` and `redeem` functions. This would enable the Vault Guardians protocol to accommodate specific requirements and customizations of the DeFi projects it integrates with, ensuring better compatibility and functionality.

```
1 - function withdraw(uint256 assets, address receiver, address owner)
      public override(IERC4626, ERC4626) divestThenInvest nonReentrant
      returns (uint256) {
2 + function withdraw(uint256 assets, address receiver, address owner,
      bytes customData) public override(IERC4626, ERC4626)
      divestThenInvest nonReentrant returns (uint256) {
```

```
1 - function redeem(uint256 assets, address receiver, address owner)
      public override(IERC4626, ERC4626) divestThenInvest nonReentrant
      returns (uint256) {
2 + function redeem(uint256 assets, address receiver, address owner,
      bytes customData) public override(IERC4626, ERC4626)
      divestThenInvest nonReentrant returns (uint256) {
```

This way, you could add a `deadline` to the Uniswap addLiquidity fucntion, and also allow for more DeFi custom integrations.

For the `amountAMin` and `amountBMin` issue, we recommend one of the following:

1. Do a price check on something like a Chainlink price feed before adding liquidity, reverting if the rate is too unfavorable.

2. To avoid potential slippage and unfavorable rates, the protocol could be limited to adding liquidity only on one side of the Uniswap pool, bypassing any token swaps. If the pool lacks sufficient liquidity or doesn't exist, the protocol should prevent the addition of liquidity to that pool. This approach minimizes risk and ensures the integrity of the investment strategy.

Note that these recommendation require significant changes to the codebase.

[H-4] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapDivest` enables frontrunners to steal profits

{#h-4}

Description: In `UniswapAdapter::_uniswapDivest` the protocol swaps one of the tokens received by removing liquidity for another token on other side of the Uniswap pool.It calls the `swapExactTokensForTokens` function of the `UniswapV2Router01` contract, which has two input parameters to note:

```
    function swapExactTokensForTokens(
        uint256 amountIn,
@>      uint256 amountOutMin,
        address[] calldata path,
        address to,
@>      uint256 deadline
    )
```

The parameter `amountOutMin` represents how much of the minimum number of tokens it expects to return. The `deadline` parameter represents when the transaction should expire.

As seen below, the `UniswapAdapter::_uniswapDivest` function sets those parameters to `0` and `block.timestamp`:

```
    uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens
        (
        amountOfTokenToSwap,
@>      0,
        s_pathArray,
        address(this),
@>      block.timestamp
    );
```

Impact: This results in either of the following happening: - Anyone (e.g., a frontrunning bot) sees this transaction in the mempool, pulls a flashloan and swaps on Uniswap to tank the price before the swap happens, resulting in the protocol executing the swap at an unfavorable rate. - Due to the lack of a deadline, the node who gets this transaction could hold the transaction until they are able to profit from the guaranteed swap.

Proof of Concept:

1. `UniswapAdapter::_uniswapDivest` gets called from some other fucntion from `VaultShares`.
    1. This call calls the router's `swapExactTokensForTokens` function.
2. In the mempool, a malicious user could:
    1. Hold onto this transaction which makes the Uniswap swap
    2. Take a flashloan out
    3. Make a major swap on Uniswap, greatly changing the price of the assets
    4. Execute the transaction that was being held, giving the protocol as little funds back as possible due to the `amountOutMin` value set to 0.

This could potentially allow malicious MEV users and frontrunners to drain balances.

Recommended Mitigation:

For the deadline issue, we recommend the following:

DeFi is a large landscape. For protocols that have sensitive investing parameters, add a custom parameter to the `deposit` function so the Vault Guardians protocol can account for the customizations of DeFi projects that it integrates with.

In the `deposit` function, consider allowing for custom data.

```
1 - function deposit(uint256 assets, address receiver) public override(
      ERC4626, IERC4626) isActive returns (uint256) {
2 + function deposit(uint256 assets, address receiver, bytes customData)
      public override(ERC4626, IERC4626) isActive returns (uint256) {
```

This way, you could add a `deadline` to the Uniswap swap, and also allow for more DeFi custom integrations.

For the `amountOutMin` issue, we recommend one of the following:

1. Do a price check on something like a Chainlink price feed before making the swap, reverting if the rate is too unfavorable.
2. Only deposit 1 side of a Uniswap pool for liquidity. Don't make the swap at all. If a pool doesn't exist or has too low liquidity for a pair of ERC20s, don't allow investment in that pool.

Note that these recommendation require significant changes to the codebase.

[H-5] Potential Sandwich Attack Vulnerability in `VaultShares::withdraw` Function

{#h-5}

Description: The `VaultShares::withdraw` function in the contract is potentially vulnerable to MEV (Miner Extractable Value) and sandwich attacks. This is because the function calculates the amount of shares to be withdrawn only after the function is called, which allows an attacker to front-run the transaction and manipulate the asset price.

Impact:
1. Attackers could front-run the transaction by either increasing or decreasing the asset price, causing the user to receive fewer or more shares than expected. 2. This could result in significant losses for users, especially in volatile markets.

Proof of Concept: 1. Attacker observes a pending withdraw transaction in the mempool. 2. Attacker submits a transaction that affects the asset price, knowing that the withdraw function will calculate the shares based on the manipulated price. 3. Once the price is manipulated, the attacker allows the original withdraw transaction to go through, resulting in an unfavorable exchange for the user.

Recommended Mitigation: • Implement a slippage protection mechanism to ensure that the withdraw transaction reverts if the asset price moves beyond a certain threshold. • Consider using private transaction submission tools like Flashbots to prevent front-running. • Evaluate other decentralized price oracles to protect against price manipulation.

[H-6] Potential MEV/Sandwich Attack Vulnerability in `VaultShares::redeem` Function

{#h-6}

Description: The `VaultShares::redeem` function in the contract is susceptible to MEV (Miner Extractable Value) and sandwich attacks. Similar to the withdraw function, the redeem function calculates the amount of assets to be redeemed only after the function is called, making it vulnerable to price manipulation.

Impact: 1. Attackers could front-run the transaction by manipulating the asset price, resulting in fewer assets being returned to the user. 2. This could lead to unexpected losses for users, especially in volatile markets.

Proof of Concept: 1. Attacker monitors the mempool and identifies a pending redeem transaction. 2. The attacker manipulates the price of the underlying assets by executing transactions before and after the redeem transaction. 3. The manipulated price results in the user receiving fewer assets than expected.

Recommended Mitigation: • Implement a slippage tolerance to revert the transaction if the redeemed asset amount deviates significantly from expectations. • Consider using private transaction submission tools like Flashbots to reduce the risk of front-running. • Introduce a time-weighted average price (TWAP) or decentralized oracles to mitigate price manipulation.

[H-7] Potential for Sandwich Attack in `VaultShares::Deposit` function.

{#h-7}

Description: The `_deposit` function inside `VaultShares` contract could be vulnerable to a sandwich attack. An attacker could front-run a deposit transaction by making a deposit just before it. This action would affect the share price, causing the subsequent depositor to receive fewer shares. Afterward, the attacker can withdraw their deposit, potentially profiting at the expense of the legitimate depositor.

Impact: This vulnerability allows an attacker to manipulate the share calculation, leading to a financial loss for the legitimate depositor.

Proof of Concept:

1. An attacker monitors the mempool for a deposit transaction.
2. The attacker front-runs this transaction by depositing a large amount.
3. The legitimate depositor receives fewer shares due to the altered share price.
4. The attacker then withdraws their deposit, profiting from the imbalance created.

Recommended Mitigation:

1.Consider implementing a time-weighted average share price (TWAP) or another mechanism to prevent sudden share price manipulation. Additionally, implement slippage protection to mitigate the impact of rapid share price changes.

2.To further prevent front-running, consider using trusted private transaction relay networks like Flashbots.net or other private node solutions to keep transactions out of the public mempool until they are confirmed.

Medium

[M-1] Incorrect calculation of the asset returned leads to incorrect return value from `UniswapAdapter::_uniswapDivest` function.

{#m-1}

Description: In the `UniswapAdapter::_uniswapDivest` function, the `UniswapV2Router01::removeLiquidity` is called to liquidate the assets and it returns the associated `token` and `counterPartyToken` back to the `VaultShares` contract. After this, `UniswapV2Router01::swapExactTokensForTokens` is called to swap the `counterPartyToken` for `token`. At the end , total amount of assets returned from uniswap as `token` is calculated which must

include the tokens received from `removeLiquidity` and `swapExactTokensForTokens` both but the variable `amountOfAssetReturned` only adds up the ones received from `swapExactTokensForTokens`

Impact: This does not affect the protocol in any way because the return value from `_uniswapDIvest` is not captured or used anywhere in the protocol.But a future upgrade to the protocol may use it and if not taken care of, will lead to wrong value being used.

```
1      function _uniswapDivest(IERC20 token, uint256 liquidityAmount)
           internal returns (uint256 amountOfAssetReturned) {
2          IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
               i_weth;
3          (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
               i_uniswapRouter.removeLiquidity({
4              tokenA: address(token),
5              tokenB: address(counterPartyToken),
6              liquidity: liquidityAmount,
7              amountAMin: 0,
8              amountBMin: 0,
9              to: address(this),
10             deadline: block.timestamp
11         });
12         s_pathArray = [address(counterPartyToken), address(token)];
13         uint256[] memory amounts = i_uniswapRouter.
               swapExactTokensForTokens({
14             amountIn: counterPartyTokenAmount,
15             amountOutMin: 0,
16             path: s_pathArray,
17             to: address(this),
18             deadline: block.timestamp
19         });
20         emit UniswapDivested(tokenAmount, amounts[1]);
21 @>      amountOfAssetReturned = amounts[1];
22     }
```

Proof of Concept:

1. `_uniswapDivest` function when called from `VaultShares::redeem` or `VaultShares::withdraw` calls `UniswapV2Router01::removeLiquidity` and receives equivalent `token` and `counterPartyToken`.
2. Once `_uniswapDivest` function receives the tokens, it calls `UniswapV2Router01::swapExactTokensForTokens` to swap the `counterPartyToken` for `Token`.
3. Once the swap is executed, `_uniswapDivest` function calculates the total tokens returned from uniswap as result of both the previosu operatiosn and returns that value to the `_uniswapDivest` function caller.
4. As shown above in the code, while calculating `amountOfAssetReturned` , the amount of

tokens returned from first operation is omitted.

Recommended Mitigation:

Considering updating `UniswapAdapter::_uniswapDivest` function as follows.

```
1   function _uniswapDivest(IERC20 token, uint256 liquidityAmount)
       internal returns (uint256 amountOfAssetReturned) {
2           IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
               i_weth;
3           (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
               i_uniswapRouter.removeLiquidity({
4               tokenA: address(token),
5               tokenB: address(counterPartyToken),
6               liquidity: liquidityAmount,
7               amountAMin: 0,
8               amountBMin: 0,
9               to: address(this),
10              deadline: block.timestamp
11          });
12          s_pathArray = [address(counterPartyToken), address(token)];
13          uint256[] memory amounts = i_uniswapRouter.
               swapExactTokensForTokens({
14              amountIn: counterPartyTokenAmount,
15              amountOutMin: 0,
16              path: s_pathArray,
17              to: address(this),
18              deadline: block.timestamp
19          });
20          emit UniswapDivested(tokenAmount, amounts[1]);
21  -       amountOfAssetReturned = amounts[1];
22  +       amountOfAssetReturned = amounts[1] + tokenAmount;
23      }
```

Low

[L-1] No Zero Check on input argument in `VaultGuardians::updateGuardianAndDaoCut` function leading to a division by zero

{#l-1}

Description: While calling the function `VaultGuardians::updateGuardianAndDaoCut` with the argument `newCut`, the function does not contain a statement to check if the input argument is non zero. So the `VaultGuardians::s_guardianAndDaoCut` variable can be set to zero by the owner.

Impact: If this happens, it can lead to a revert caused by `division by zero` in the `VaultShares::deposit` function as shown in the code below, when it is called.

```
    function deposit(uint256 assets, address receiver)
        public
        override(ERC4626, IERC4626)
        isActive
        nonReentrant
        returns (uint256)
    {
        if (assets > maxDeposit(receiver)) {
            revert VaultShares__DepositMoreThanMax(assets, maxDeposit(
                receiver));
        }

        uint256 shares = previewDeposit(assets);
        _deposit(_msgSender(), receiver, assets, shares);
@>      _mint(i_guardian, shares / i_guardianAndDaoCut);
@>      _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);

        _investFunds(assets);
        return shares;
    }
```

Proof of Concept: Add the following code to the `VaultGuardiansBaseTest.t.sol` file.

Code

```
function testNoZeroCheck() public {
        uint256 newGuardianAndDaoCut = 0;
        vm.startPrank(vaultGuardians.owner());
        vaultGuardians.updateGuardianAndDaoCut(newGuardianAndDaoCut);
        vm.stopPrank();
        weth.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        weth.approve(address(vaultGuardians), mintAmount);
        vm.expectRevert();
        vaultGuardians.becomeGuardian(allocationData);
        vm.stopPrank();
    }
```

Recommended Mitigation:

Consider updating the `VaultGuardians::updateGuardianAndDaoCut` function as follows:

```
  function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
+       require(newCut!=0, "s_guardianAndDaoCut should not be zero")
        s_guardianAndDaoCut = newCut;
        emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut,
            newCut);
    }
```

[L-2] Missing Zero Value Checks on `VaultShares::deposit` function arguments

{#l-2}

Description: The `VaultShares::deposit` function in the Vault contract does not include checks to ensure that the `assets` and `receiver` arguments are non-zero. This omission may result in the transaction reverting at a later stage during execution, causing unnecessary gas costs and a poor user experience. The absence of such validation could also lead to potential unexpected behavior, particularly in edge cases where these arguments might be zero.

Impact: - **Gas Waste:** Users may lose significant gas fees if the transaction reverts at a late stage when the contract logic encounters the zero-value arguments. - **User Experience:** Failing to provide early feedback to users about invalid inputs could lead to confusion and frustration. - **Potential Edge Case Issues:** Lack of checks for zero-value inputs might expose the contract to edge cases that could be exploited or cause unexpected behavior.

Proof of Concept:

1. A user calls the `deposit` function with `assets` set to `0` or `receiver` set to the zero address (`0x0000000000000000000000000000000000000000`).
2. The transaction proceeds through various steps, possibly including `_deposit`, `_mint`, and `_investFunds`, only to revert at a later point, wasting the user's gas.

Recommended Mitigation:

Add input validation checks at the beginning of the `deposit` function to ensure `assets` is greater than zero and `receiver` is not the zero address. This can be done by adding the following lines in the function.

```
function deposit(uint256 assets, address receiver)
        public
        override(ERC4626, IERC4626)
        isActive
        nonReentrant
        returns (uint256)
    {
+       require(assets > 0, "Assets amount must be greater than zero");
+       require(receiver != address(0), "Receiver address cannot be
    zero");
        if (assets > maxDeposit(receiver)) {
            revert VaultShares__DepositMoreThanMax(assets, maxDeposit(
                receiver));
        }

        uint256 shares = previewDeposit(assets);
        _deposit(_msgSender(), receiver, assets, shares);
        _mint(i_guardian, shares / i_guardianAndDaoCut);
```

```
17          _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);
18
19          _investFunds(assets);
20          return shares;
21      }
```

Informational

[I-1] Lack of Dedicated Emit Event for Minted Shares in `VaultShares::deposit` Function

{#i-1}

Description: The `VaultShares::deposit` function currently mints additional shares to `i_guardian` and `i_vaultGuardians` without a dedicated emit event to notify users about these mints. While the function does emit a `Deposit` event for the primary transaction, there is no separate event for the minted shares related to fees, leading to a lack of transparency.

Impact: - **Lack of Transparency:** Users and off-chain monitoring tools may not be fully informed about the additional shares minted as fees. This could lead to confusion or difficulty in tracking the distribution of shares. - **Auditing Complexity:** The absence of specific emit events for these mints complicates the auditing process, as the event logs won't clearly indicate the minting of these shares. - **Potential User Mistrust:** Users might lose trust in the system if they are unaware of how many shares are being allocated to the guardian and vault guardian.

Proof of Concept: 1. A user calls the `deposit` function. 2. The function mints shares for `i_guardian` and `i_vaultGuardians` without emitting a dedicated event. 3. This results in a lack of visibility for these mints in the event logs, making it harder to track.

Recommended Mitigation: Introduce dedicated emit events to log the minting of shares to `i_guardian` and `i_vaultGuardians` within the `deposit` function. For example:

```
1  + event SharesMintedForFees(address,uint256);
2  function deposit(uint256 assets, address receiver)
3          public
4          override(ERC4626, IERC4626)
5          isActive
6          nonReentrant
7          returns (uint256)
8      {
9          if (assets > maxDeposit(receiver)) {
10             revert VaultShares__DepositMoreThanMax(assets, maxDeposit(
                   receiver));
11         }
12
13         uint256 shares = previewDeposit(assets);
```

```
14          _deposit(_msgSender(), receiver, assets, shares);
15          _mint(i_guardian, shares / i_guardianAndDaoCut);
16          _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);
17  +       emit SharesMintedForFees(i_guardian, shares /
        i_guardianAndDaoCut);
18  +       emit SharesMintedForFees(i_vaultGuardians, shares /
        i_guardianAndDaoCut);
19
20          _investFunds(assets);
21          return shares;
22      }
```

[I-2] Lack of Zero Checks in `VaultShares::withdraw` Function

{#i-2}

Description: The `VaultShares::withdraw` function does not perform zero checks on the assets, receiver, and owner parameters. If assets is zero, the function will still proceed to execute, potentially wasting gas and unnecessarily invoking other functions.

Impact: 1. Users may waste gas fees when trying to withdraw zero assets. 2. It could lead to unnecessary state changes and events being emitted, which could complicate debugging and auditing.

Proof of Concept: 1. A user mistakenly calls the withdraw function with assets set to zero. 2. The transaction proceeds, consuming gas and performing unnecessary operations. **Recommended Mitigation:** Add zero checks for assets, receiver, and owner parameters at the beginning of the function to revert early if any of these parameters are invalid. For example, as added in the follwoing code.

```
1       function withdraw(uint256 assets, address receiver, address owner)
2               public
3               override(IERC4626, ERC4626)
4               divestThenInvest
5               nonReentrant
6               returns (uint256)
7           {
8   +           require(assets > 0, "Assets must be greater than zero");
9   +           require(receiver != address(0), "Receiver cannot be the
        zero address");
10  +           require(owner != address(0), "Owner cannot be the zero
        address");
11              uint256 shares = super.withdraw(assets, receiver, owner);
12              return shares;
13          }
```

[I-3] Lack of Zero Checks in `VaultShares::redeem` Function

{#i-3}

Description: The `VaultShares::redeem` function lacks zero checks for the shares, receiver, and owner parameters. Allowing these parameters to be zero could lead to wasted gas fees and unnecessary function execution.

Impact: 1. Users could incur gas costs for operations that ultimately yield no result. 2. It may result in unnecessary state changes and events, making the contract harder to audit and debug.

Proof of Concept: 1. A user calls the redeem function with shares set to zero. 2. The function proceeds with the transaction, resulting in wasted gas and unnecessary operations.

Recommended Mitigation: Implement checks to ensure that the shares, receiver, and owner parameters are non-zero before proceeding with the function's logic.For example, as added in the following code.

```
    function redeem(uint256 shares, address receiver, address owner)
    public
    override(IERC4626, ERC4626)
    divestThenInvest
    nonReentrant
    returns (uint256)
        {
+           require(shares > 0, "Shares must be greater than zero");
+           require(receiver != address(0), "Receiver cannot be the
    zero address");
+           require(owner != address(0), "Owner cannot be the zero
    address");

            uint256 assets = super.redeem(shares, receiver, owner);
            return assets;
        }
```

[I-4] The `nonReentrant modifier` should occur before all other modifiers

{#i-4}

Description: In the following listed functions,the `nonReentrant modifier` should occur before all other modifiers.This is a best-practice to protect against reentrancy in other modifiers

List of Functions

1.`VaultShares::deposit` 2.`VaultShares::withdraw` 3.`VaultShares::redeem`

Gas

[G-1] Inefficient Use of `VaultGuardiansBase::s_guardianStakePrice` in `VaultGuardiansBase::_becomeTokenGuardian` Function resulting in unoptimized gas usage.

{#g-1}

Description: In the `VaultGuardiansBase::_becomeTokenGuardian` function, the variable `VaultGuardiansBase::s_guardianStakePrice` is accessed multiple times throughout the function. Each access involves reading the value from storage, which is a relatively expensive operation in terms of gas costs.

Impact: Repeatedly accessing the `VaultGuardiansBase::s_guardianStakePrice` variable from storage increases the gas costs of the `VaultGuardiansBase::_becomeTokenGuardian` function. While the individual gas cost might seem small, in aggregate, this can lead to significant inefficiencies, especially if the function is called frequently.

Proof of Concept:

As indicated below, `s_guardianStakePrice` storage variable is repeatedly accessed from storage directly.

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

Recommended Mitigation:

Consider caching that particular variabel and using acorss the function as follows.

```
1
2       function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
            private returns (address) {
3   +           uint256 stakePrice = s_guardianStakePrice;
4               s_guardians[msg.sender][token] = IVaultShares(address(
                    tokenVault));
5               emit GuardianAdded(msg.sender, token);
6   -           i_vgToken.mint(msg.sender, s_guardianStakePrice);
7   +           i_vgToken.mint(msg.sender, stakePrice);
8   -           token.safeTransferFrom(msg.sender, address(this),
        s_guardianStakePrice);
9   +           token.safeTransferFrom(msg.sender, address(this),
        stakePrice);
10  -           bool succ = token.approve(address(tokenVault),
        s_guardianStakePrice);
11  +           bool succ = token.approve(address(tokenVault), stakePrice);
12              if (!succ) {
13                  revert VaultGuardiansBase__TransferFailed();
14              }
15  -           uint256 shares = tokenVault.deposit(s_guardianStakePrice,
        msg.sender);
16  +           uint256 shares = tokenVault.deposit(stakePrice, msg.sender)
        ;
17              if (shares == 0) {
18                  revert VaultGuardiansBase__TransferFailed();
19              }
20              return address(tokenVault);
21          }
```

[G-2] Functions not used internally could be marked external

{#g-2}

Description: The following listed have not been used internally but marked as public. Consider marking them as external as a better practice and to save gas when the fucntion is called.

List of functions

1. `VaultGuardianGivernor::votingPeriod`
2. `VaultGuardianGovernor::quorum`
3. `VaultGuardianGovernor::votingDelay`
4. `VaultGuardianToken::nonces`
5. `VaultShares::setNotActive`
6. `VaultShares::deposit`
7. `VaultShares::withdraw`
8. `VaultShares::redeem`

[G-3] Constants should be defined and used instead of literals

{#g-3}

Description: The following listed literals can be replaced with constants which offers better readability fo the code and optimizes gas usage.

List of literals used

1. `VaultGuardianGivernor::votingPeriod:: 7 days`
2. `UniswapAdapter::_uniswapInvest::amountofTokenToSwap = amount/2`
3. `VaultGuardianGovernor::votingDelay:: 1 day`
4. `VaultGuardianGovernor::constructor::GovernorVotesQuorumFraction(4)`

[G-4] Unused imports can be removed

{#g-4}

Description: The following imports are never used and can be considered to be removed for better gas optimisation.

List of unused imports

1. `InvestableUniverseAdapter.sol::{IERC20}`
2. `VaultShares.sol::{Datatypes}`