Vault Guardian Protocol Secuity Review

Version 1.0

Tanu Gupta

August 23, 2025

Protocol Audit Report

Tanu Gupta

Aug 23, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] MEV Attack on `AllocationData` updates enables massive share inflation and leading to huge fund theft
    * [H-2] Missing decimal normalization for assets like USDC causing token amount mis-calculation

    * [H-3] `VaultShares::constructor` sets LP Token address to zero for WETH asset causing permanent loss of funds inside uniswap pool
    * [H-4] `UniswapAdapter::_uniswapInvest` incorrectly calculates liquidity to add, leading to higher than intended deposits in Uniswap
    * [H-5] `UniswapAdapter::_uniswapDivest` lacks the token approval for Uniswap router before swap operation for `counterPartyToken` leading to Denial of Service
    * [H-6] `VaultShares::deposit` causes incorrect share distribution by not accounting for invested assets
    * [H-7] `VaultShares::deposit` creates undercollaterization due to unbacked share minting for DAO and guardian
    * [H-8] `VaultShares::deposit` does not reduce the assets amount to invest after fees are deducted for DAO and guardian and returns the incorrect amount of user shares
    * [H-9] Zero allocations for Uniswap and Aave allow guardians to drain vault through share manipulation such as 1000_0_0 allocation
    * [H-10] Missing `Payable` Modifier in `VaultGuardianBase::becomeGuardian` Causes Loss of Guardian Fees
- Medium
    * [M-1] `UniswapAdapter::_uniswapDivest` returns incorrect asset amount extracted from Uniswap
    * [M-2] `VaultGuardians::updateGuardianAndDaoCut` emits the incorrect event with wrong parameters
- Low
    * [L-1] `UniswapAdapter::_uniswapInvest` allows excessive token approval beyond required amount
    * [L-2] Event Emission Inconsistency in `UniswapAdapter::_uniswapInvest`
    * [L-3] Event Emission Inconsistency in `UniswapAdapter::_uniswapDivest`
    * [L-4] Fee share can truncate to zero for small deposits bypassing protocol revenue
    * [L-5] `VaultGuardians::VaultGuardians__UpdatedStakePrice` is emitted with incorrect parameters
    * [L-6] `VaultGuardians::constructor` does not validate addresses against zero address
    * [L-7] `VaultGuardians::sweepErc20s` allows anyone to drain contract balances to owner() by passing any IERC20 address this can lead to non-standard ERC20 token attack
- Informational

* [I-1] Empty Interface Definition for `IInvestableUniverseAdapter` with unused import
* [I-2] Empty Interface Definition for `IVaultGuardians`
* [I-3] Missing NatSpec Documentation for Interface `IVaultShares`
* [I-4] Unused custom errors in the codebase
* [I-5] Unused event definitions in the codebase

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

The 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

Audit Details

The findings described in this document correspond to repository Vault Guardians.

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	10
Medium	2
Low	7
Info	5
Gas	0
Total	24

Findings

High

[H-1] MEV Attack on `AllocationData` updates enables massive share inflation and leading to huge fund theft

Description Sophisticated MEV operators can backrun/frontrun *guardian allocation update* `updateHoldingAllocation` transactions to exploit temporary `totalAssets()` miscalculation.

During allocation shifts between held assets and invested positions, attackers can `front-run` or `back-run` `deposit`/`withdraw` calls around the update. This allows them to obtain **disproportionately** high shares or assets when `totalAssets()` temporarily falls near zero or spikes to a maximum, despite the vault's actual value remaining stable in external protocols.

Impact

- Attackers can steal significant portions of vault funds through share manipulation.
- Complete breakdown of vault economics during allocation changes.

- MEV bots can systematically drain vault value over time.

Proof of Concepts

1. A user deposits assets into the vault, receiving shares based on the current `totalAssets()`.
2. The guardian updates the holding allocation, which invests 50% of the assets into Uniswap and the remaining 50% into the Aave.
3. The user deposits again after rebalancing, and due to the temporary `totalAssets()` being very low (as all assets are invested), they receive an inflated number of shares.
4. User will then redeems all their shares after the market conditions become favorable, receiving a massive amount of assets due to the inflated shares.

Paste this code in test file to reproduce this issue

Proof Of Code (POC)

```
function testMEVAttackWithUpdateAllocation() external hasGuardian
    hasTokenGuardian {

        usdc.mint(20 ether, user);
        vm.startPrank(user);
        usdc.approve(address(usdcVaultShares), 20 ether);
        VaultShares(usdcVaultShares).deposit(20 ether, user);
        uint256 userShare = VaultShares(usdcVaultShares).balanceOf(user
            );
        console2.log("userShare: ", userShare); //10.02
        vm.stopPrank();

        vm.prank(guardian);
        vaultGuardians.updateHoldingAllocation(usdc, newAllocationData)
            ;
        VaultShares(usdcVaultShares).rebalanceFunds();

        usdc.mint(20 ether, user);
        vm.startPrank(user);
        usdc.approve(address(usdcVaultShares), 20 ether);
        VaultShares(usdcVaultShares).deposit(20 ether, user);
        uint256 userShareNew = VaultShares(usdcVaultShares).balanceOf(
            user);
        //Without Rebalancing: 106.986879999999999982
        //WithRebalancing: 1003603199999999999920.079999999999999995
        console2.log("userShare: ", userShareNew);
        vm.stopPrank();

        //total shares user is capable of redeeming
        vm.startPrank(user);
        uint256 userMaxRedeem = VaultShares(usdcVaultShares).maxRedeem(
            user);
        console2.log("userMaxRedeem: ", userMaxRedeem);
```

```
29
30          deal(address(usdc), address(usdcVaultShares), 1000 ether);
31          //user waited till market condition becomes favourable
32          uint256 assetsReceived = usdcVaultShares.redeem(userMaxRedeem,
                user, user);
33          console2.log("assetsReceived: ", assetsReceived);
34          vm.stopPrank();
35      }
```

Recommended mitigation

1. Fix `totalAssets()` calculation to include invested assests

```
function totalAssets() public view override returns (uint256) {
    return asset().balanceOf(address(this)) +
           _getAaveInvestedAmount() +
           _getUniswapInvestedAmount();
}
```

2. Alternatively, set `MINIMUM_PROTOCOL_ALLOCATION` and `MINIMUM_HOLDING_ALLOCATION` to a non-zero value (e.g., 5%) to ensure some assets are always held in the vault, preventing `totalAssets()` from ever reaching zero or spiking to its max value.

[H-2] Missing decimal normalization for assets like USDC causing token amount miscalculation

Description The functions performs arithmetic operations and token swaps without accounting for different token decimal places. This causes severe **miscalculations** when dealing with tokens that have different decimal precision (`e.g., WETH has 18 decimals, USDC has 6 decimals`), leading to protocol integration failures such as `Aave error code 51` (insufficient balance/allowance) when attempting to supply calculated amounts.

Impact

- Protocol integration failures preventing core functionality
- Massive over/under-calculation of token amounts leading to transaction reverts
- Contract unusable with multi-decimal token pairs

Proof of Concepts

- Trying to supply `2.5e18 USDC` *(which is 2,500,000,000,000,000,000 USDC - way too much!)*, but USDC only has 6 decimals.
- This leads to `Aave error code 51` (insufficient balance/allowance) when trying to supply the calculated amount.

- The `supply` function in the `AaveAdapter` contract is called with an incorrect amount due to missing decimal normalization.

Paste this code in a foked mainnet test file to reproduce the issue

Proof of Code (POC)

```
function testingDivestforUSDCVaultForkMainnet() external hasGuardian {
        deal(address(usdc), guardian, mintAmount);
        vm.startPrank(guardian);
        usdc.approve(address(vaultGuardians), mintAmount);
        address usdcVault = vaultGuardians.becomeTokenGuardian(
           allocationData, usdc);
        usdcVaultShares = VaultShares(usdcVault);
        vm.stopPrank();
    }
```

Recommended mitigation

1. Implement decimal normalization utility. The full code for normalization library is available here

```
function normalizeAmount(uint256 amount18Decimals, uint256 decimals)
    internal pure returns (uint256) {
    if (decimals == 18) {
        return amount18Decimals;
    } else if (decimals < 18) {
        return amount18Decimals / (10 ** (18 - decimals));
    } else {
        return amount18Decimals * (10 ** (decimals - 18));
    }
}
```

2. Update the `UniswapAdapter::_uniswapInvest` and `UniswapAdapter::_uniswapDivest` functions to use the normalization utility when calculating amounts for token swaps and liquidity provision.
3. Similary update the `AaveAdapter::_supply` and `AaveAdapter::_withdraw` functions to use the normalization utility when calculating amounts for Aave supply and withdrawal operations.

[H-3] `VaultShares::constructor` sets LP Token address to zero for WETH asset causing permanent loss of funds inside uniswap pool

Description When the `constructor's` asset parameter is `WETH`, the `getPair()` call becomes `getPair(address(i_weth), address(i_weth))`, attempting to create a pair with the

same token twice. Uniswap V2 factory returns `address(0)` for identical token pairs, causing `i_uniswapLiquidityToken` to be set to the **zero address**.

```
i_uniswapLiquidityToken = IERC20(
  i_uniswapFactory.getPair(address(constructorData.asset), address(
     i_weth))
);
```

Impact Divesting logic depends on LP token balance checks (`liquidityAmount > 0`), users can never withdraw their funds when the asset is WETH.

```
    uint256 uniswapLiquidityTokensBalance = i_uniswapLiquidityToken.
       balanceOf(address(this));
    if (uniswapLiquidityTokensBalance > 0) {
        _uniswapDivest(IERC20(asset()), uniswapLiquidityTokensBalance);
    }
```

Proof of Concepts

1. Deploy the `VaultShares` contract with `WETH` as the asset via the `becomeGuardian` function.
2. The `constructor` sets `i_uniswapLiquidityToken` to the zero address.

```
function testReturnsZeroLPTokenAddressForWETH() external {
        address tokenA = address(weth) < address(usdc) ? address(weth)
           : address(usdc);
        address tokenB = address(weth) < address(usdc)? address(usdc) :
            address(weth);
        IUniswapV2Factory factoryContract = IUniswapV2Factory((
           IUniswapV2Router02(uniswapRouter)).factory());

        address pair = UniswapV2Library.pairFor(address(factoryContract
           ), tokenA, tokenB);
        console2.log("Pair: ", pair);

        IERC20 uniswapLiquidityToken = IERC20(factoryContract.getPair(
           tokenA, tokenB));
        console2.log("uniswapLiquidityToken: ", address(
           uniswapLiquidityToken));


        deal(address(weth), guardian, mintAmount);
        vm.startPrank(guardian);
        weth.approve(address(vaultGuardians), mintAmount);
        address wethVault = vaultGuardians.becomeGuardian(
           allocationData);
        wethVaultShares = VaultShares(wethVault);
        vm.stopPrank();

        address lpToken = wethVaultShares.getUniswapLiquidtyToken();
        assertEq(lpToken, address(0));
```

```
22          console2.log("lpToken: ", lpToken);
23      }
```

Log output:

```
1  Pair: 0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc;
2  uniswapLiquidityToken: 0xb4e16d0168e52d35cacd2c6185b44281ec28c9dc;
3  lpToken: 0x0000000000000000000000000000000000000000;
```

Recommended mitigation

```
1  -    i_uniswapLiquidityToken = IERC20(i_uniswapFactory.getPair(address(
       constructorData.asset), address(i_weth)));
2  +    address counterToken =
3  +         address(constructorData.asset) == address(i_weth) ?
       constructorData.usdc : address(i_weth);
4  +         i_uniswapLiquidityToken = IERC20(i_uniswapFactory.getPair(
       address(constructorData.asset), counterToken));
```

[H-4] `UniswapAdapter::_uniswapInvest` incorrectly calculates liquidity to add, leading to higher than intended deposits in Uniswap

Description The `amountADesired` parameter uses `amountOfTokenToSwap + amounts[0]` instead of just `amountOfTokenToSwap`. Since `amounts[0]` equals the input amount to the swap, this doubles the intended liquidity provision amount, potentially depleting more tokens than the guardian intended to invest in uniswap.

Impact

- User funds at risk - investing more than intended
- Potential contract insolvency if insufficient balance

Proof of Concepts

1. Guardian becomes the guardian of weth vault.
2. Guardian calls `updateHoldingAllocation` to set a new allocation for WETH.
3. Setting the hold allocation to 0, 50% for Uniswap and the remaining 50% for the Aave.
4. Guardian calls `rebalanceFunds` to divest the funds and invest based on the new allocation.
5. The `UniswapAdapter::_uniswapInvest` function is called, which calculates the liquidity provision amount as `amountADesired = amountOfTokenToSwap + amounts[0]`, leading to a higher deposit than intended.
6. The amount allocated to Aave is higher than the available balance, causing the transaction to revert with `ERC20InsufficientBalance`.

```
function testInvestmentUnableToGoThrough() external hasGuardian {
        vm.prank(guardian);
        vaultGuardians.updateHoldingAllocation(weth, newAllocationData)
            ;

        vm.expectRevert();
        //Reverting with ERC20InsufficientBalance
        wethVaultShares.rebalanceFunds();
    }
```

Recommended mitigation

```
    (uint256 tokenAmount, uint256 counterPartyTokenAmount, uint256
        liquidity) = i_uniswapRouter.addLiquidity({
        tokenA: address(token),
        tokenB: address(counterPartyToken),
-       amountADesired: amountOfTokenToSwap + amounts[0],
+       amountADesired: amountOfTokenToSwap,
        amountBDesired: amounts[1],
        // ... rest of parameters
    });
```

[H-5] `UniswapAdapter::_uniswapDivest` lacks the token approval for Uniswap router before swap operation for `counterPartyToken` leading to Denial of Service

Description The function attempts to swap `counterPartyToken` tokens via `swapExactTokensForTokens` without first approving the Uniswap router to spend the tokens.

```
function _uniswapDivest(IERC20 token, uint256 liquidityAmount) internal
    returns (uint256 amountOfAssetReturned) {
        IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
            i_weth;

        (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
            i_uniswapRouter.removeLiquidity({
            tokenA: address(token),
            tokenB: address(counterPartyToken),
            liquidity: liquidityAmount,
            amountAMin: 0,
            amountBMin: 0,
            to: address(this),
            deadline: block.timestamp
        });
        s_pathArray = [address(counterPartyToken), address(token)];

@>      uint256[] memory amounts = i_uniswapRouter.
    swapExactTokensForTokens({
            amountIn: counterPartyTokenAmount,
```

```
17                 amountOutMin: 0,
18                 path: s_pathArray,
19                 to: address(this),
20                 deadline: block.timestamp
21             });
22             emit UniswapDivested(tokenAmount, amounts[1]);
23             amountOfAssetReturned = amounts[1];
24         }
```

Impact

- Function will always revert during swap operation, making divest functionality completely broken
- Users unable to withdraw their invested funds can cause denial of service (DOS) for the vault

Proof of Concepts

```
1  // After removeLiquidity, contract receives counterPartyToken
2  (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
       i_uniswapRouter.removeLiquidity(...);
3
4  // This call will REVERT - router has no permission to spend
       counterPartyToken
5  uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
6      amountIn: counterPartyTokenAmount, // Router tries to spend tokens
           it can't access
7      // ... other parameters
8  });
9  // Result: Transaction reverts with "ERC20: transfer amount exceeds
       allowance"
```

Recommended mitigation Add token approval for the Uniswap router before the swap operation

```
1  +    bool succ = counterPartyToken.approve(address(i_uniswapRouter),
       counterPartyTokenAmount);
2  +    if (!succ) {
3  +        revert UniswapAdapter__TransferFailed();
4  +    }
5
6  uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
7      amountIn: counterPartyTokenAmount,
8      amountOutMin: 0,
9      path: s_pathArray,
10     to: address(this),
11     deadline: block.timestamp
12 });
```

[H-6] `VaultShares::deposit` causes incorrect share distribution by not accounting for invested assets

Description The `previewDeposit(assets)` function relies on `totalAssets()` to calculate the *share-to-asset* ratio. However, `ERC4626::totalAssets()` only checks the vault's underlying asset balance and does not account for assets that have already been invested in external protocols.

This results in artificially low `totalAssets()` values, leading to incorrect share calculations where users receive more shares than they should based on the true total vault value.

Impact

- Users receive significantly more shares than deserved when assets are invested
- Severe dilution of existing shareholders when vault has low cash reserves

Proof of Concepts

- A user deposits assets into the vault, and the function calculates shares based on the current `totalAssets()`.
- Guardian then updates the holding allocation, which invests 50% of the assets into Uniswap and the remaining 50% into Aave.
- When the user deposits again after rebalancing, the function calculates shares based on the new `totalAssets()` which is `0` in this scenario as the vault has no cash reserve.
- Causing highly inflated share amounts to be minted for the user.

Paste this code in test file to reproduce this issue

Proof Of Code (POC)

```
+   // Resolve this issue first by adding correct amount of liquidity to the uniswap pool to simulate this scenario
-   amountADesired: amountOfTokenToSwap + amounts[0],
+   amountADesired: amountOfTokenToSwap,
```

```
function testUnfairShareDistributionToUsers() external hasGuardian{
        weth.mint(20 ether, user);
        vm.startPrank(user);
        weth.approve(address(wethVaultShares), 20 ether);
        VaultShares(wethVaultShares).deposit(20 ether, user);
        uint256 userShare = VaultShares(wethVaultShares).balanceOf(user);
        console2.log("userShare: ", userShare); //40.079999999999999995
        vm.stopPrank();

        vm.prank(guardian);
        // Holding 0 amount of assets in the vault
        // Investing 50% of the total assets in uniswap
```

```
13          // Investing remaining 50% to the Aave
14          // newAllocationData = AllocationData(0, 500, 500);
15          vaultGuardians.updateHoldingAllocation(weth, newAllocationData)
                ;
16          // Rebalancing invest assets based off new allocation data
17          VaultShares(wethVaultShares).rebalanceFunds();
18
19          assertEq(wethVaultShares.totalAssets(), 0);
20          //user deposits again
21          weth.mint(20 ether, user);
22          vm.startPrank(user);
23          weth.approve(address(wethVaultShares), 20 ether);
24          VaultShares(wethVaultShares).deposit(20 ether, user);
25          uint256 userShareNew = VaultShares(wethVaultShares).balanceOf(
                user);
26          //Without Rebalancing: 106.986879999999999982
27          //WithRebalancing: 1003603199999999999920.079999999999999995
28          console2.log("userShare: ", userShareNew);
29          vm.stopPrank();
30
31          uint256 totalSharesMinted = wethVaultShares.totalSupply();
32          vm.expectRevert(Math.MathOverflowedMulDiv.selector);
33          wethVaultShares.previewWithdraw(totalSharesMinted);
34      }
```

Recommended mitigation Override `totalAssets()` to account for invested amounts

```
+   function totalAssets() public view override returns (uint256) {
+       return asset().balanceOf(address(this)) + _getInvestedAssets();
+   }
+
+   function _getInvestedAssets() internal view returns (uint256) {
+       // Sum all invested positions across protocols
+       uint256 aaveBalance = _getAaveBalance();
+       uint256 uniswapBalance = _getUniswapBalance();
+       return aaveBalance + uniswapBalance;
+   }
```

[H-7] `VaultShares::deposit` creates undercollaterization due to unbacked share minting for DAO and guardian

Description The function **mints** additional fee shares to *guardian* and *DAO* addresses (`shares / i_guardianAndDaoCut`) each after already calculating and minting the correct share amount to the user. This creates more total shares than the deposited assets can back, leading to vault **undercollateralization** where total shares exceed the asset backing ratio.

Impact

- Vault becomes undercollateralized with unbacked shares
- Dilution of all existing shareholders' value
- Economic loss for all vault participants

Proof of Concepts In the following code, we simulate a scenario where a user deposits assets into the vault, and the function mints additional shares to the guardian and DAO.

Here, the total shares minted exceed the assets backing them, causing the vault to become undercollateralized.

Proof Of Code (POC)

```
function testUnbackedSharesMintingCausingUnderCollateralization()
    external{
        AllocationData memory allocationDataNew = AllocationData(1000,
            0, 0);

        weth.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        weth.approve(address(vaultGuardians), mintAmount);
        address wethVault = vaultGuardians.becomeGuardian(
            allocationDataNew);
        wethVaultShares = VaultShares(wethVault);
        vm.stopPrank();

        // user makes a deposit
        weth.mint(7 ether, user);
        vm.startPrank(user);
        weth.approve(address(wethVaultShares), 7 ether);
        wethVaultShares.deposit(7 ether, user);
        vm.stopPrank();

        //user makes another deposit
        weth.mint(5 ether, user);
        vm.startPrank(user);
        weth.approve(address(wethVaultShares), 5 ether);
        wethVaultShares.deposit(5 ether, user);
        vm.stopPrank();

        uint256 totalSharesMinted = wethVaultShares.totalSupply(); //
            10.02
        console2.log(totalSharesMinted);

        //preview Withdraw
        uint256 actualAssets = wethVaultShares.totalAssets();
        uint256 expectedAssets = wethVaultShares.previewWithdraw(
            totalSharesMinted);
        console2.log(expectedAssets, actualAssets);
        assertGt(expectedAssets, actualAssets);
    }
```

Recommended mitigation

```
 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);

+       uint256 grossShares = previewDeposit(assets);
+       uint256 feeShares = grossShares * 2 / i_guardianAndDaoCut; // 2
    because guardian + DAO
+       uint256 netSharesForUser = grossShares - feeShares;

+       _deposit(_msgSender(), receiver, assets, netSharesForUser);
+       _mint(i_guardian, feeShares / 2);
+       _mint(i_vaultGuardians, feeShares / 2);
        _investFunds(assets);
        return shares;
    }
```

[H-8] `VaultShares::deposit` does not reduce the assets amount to invest after fees are deducted for DAO and guardian and returns the incorrect amount of user shares

Description The function invests the full deposited assets amount via `_investFunds(assets)` without accounting for the fact that additional fee shares were minted. Since more shares now exist than the original calculation anticipated, the investment amount should be proportionally reduced to maintain proper *asset-to-share* backing ratio.

The function also returns the incorrect amount of user shares without considering the fee shares that were minted for the guardian and DAO.

```
function deposit(uint256 assets, address receiver)
        public
        override(ERC4626, IERC4626)
        isActive
        nonReentrant
        returns (uint256)
    {
```

```
8           if (assets > maxDeposit(receiver)) {
9               revert VaultShares__DepositMoreThanMax(assets, maxDeposit(
                    receiver));
10          }
11          uint256 shares = previewDeposit(assets);
12          _deposit(_msgSender(), receiver, assets, shares);
13          _mint(i_guardian, shares / i_guardianAndDaoCut);
14          _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);
15  @>      _investFunds(assets);
16          return shares;
17      }
```

Impact

- Over-investment of funds relative to share distribution
- Mismatch between invested assets and total share claims

Proof of Concepts

1. A user deposits assets into the vault. and instead of investing the full amount, the function should reduce the assets to invest by the fees.
2. This shows the `actualAmountToInvest` is less than the assets amount that is being invested.

Paste this code in a test file to reproduce the issue

Proof Of Code (POC)

```
1  function testOverInvestmentOfFunds() external{
2          AllocationData memory allocationDataNew = AllocationData(1000,
               0, 0);
3
4          weth.mint(mintAmount, guardian);
5          vm.startPrank(guardian);
6          weth.approve(address(vaultGuardians), mintAmount);
7          address wethVault = vaultGuardians.becomeGuardian(
               allocationDataNew);
8          wethVaultShares = VaultShares(wethVault);
9          vm.stopPrank();
10
11         uint256 assetsToInvest = 25 ether;
12         weth.mint(assetsToInvest, user);
13         vm.startPrank(user);
14         weth.approve(address(wethVaultShares), assetsToInvest);
15
16         uint256 userShares = wethVaultShares.previewDeposit(
               assetsToInvest);
17         uint256 feeShares = userShares * 2 / wethVaultShares.
               getGuardianAndDaoCut();
18
19         uint256 totalSharesCreated = userShares + feeShares;
```

```
20
21          // The function invests the full assets amount without reducing
                it for fees
22          // This leads to over-investment of funds relative to the
               shares created
23          uint256 actualAssetsToInvest = assetsToInvest * userShares /
               totalSharesCreated;
24          wethVaultShares.deposit(assetsToInvest, user);
25          vm.stopPrank();
26          assertLt(actualAssetsToInvest, assetsToInvest);
27      }
```

Recommended mitigation

```
1  function deposit(uint256 assets, address receiver)
2          public
3          override(ERC4626, IERC4626)
4          isActive
5          nonReentrant
6          returns (uint256)
7      {
8          if (assets > maxDeposit(receiver)) {
9              revert VaultShares__DepositMoreThanMax(assets, maxDeposit(
                   receiver));
10         }
11         uint256 shares = previewDeposit(assets);
12 +       uint256 feeShares = (shares / i_guardianAndDaoCut) * 2;
13 +       uint256 totalSharesCreated = shares + feeShares;
14 +       uint256 userShares = shares - feeShares;
15 -       _deposit(_msgSender(), receiver, assets, shares);
16 +       _deposit(_msgSender(), receiver, assets, userShares);
17
18         _mint(i_guardian, shares / i_guardianAndDaoCut);
19         _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);
20 +       // Calculate the actual amount to invest after accounting for
       fees
21 +       uint256 actualAmountToInvest = assets * shares /
       totalSharesCreated;
22 -       _investFunds(assets);
23 +       _investFunds(actualAmountToInvest);
24 -        return shares;
25 +        return userShares;
26     }
```

[H-9] Zero allocations for Uniswap and Aave allow guardians to drain vault through share manipulation such as 1000_0_0 allocation

Description Guardians can manipulate vault allocations by setting *Uniswap* and *Aave* allocations to `zero`, causing all deposited funds to remain uninvested in the vault contract.

This creates a scenario where `totalAssets()` accurately reflects the vault balance while share calculations remain based on artificially low values.

Guardians can exploit this by timing their withdrawals by seeting allocations to `0` to redeem significantly more assets than their shares should represent.

Impact

- Guardians can drain vault funds through allocation manipulation
- Complete breakdown of vault share-to-asset ratio integrity
- After sometime protocol becomes unusable due to guardian exploitation risk.

Proof of Concepts

Proof Of Code (POC)

```
function testGuardianStealingByUpdatingAllocation() external
   hasGuardian hasTokenGuardian{
        usdc.mint(20 ether, user);
        vm.startPrank(user);
        usdc.approve(address(usdcVaultShares), 20 ether);
        VaultShares(usdcVaultShares).deposit(20 ether, user);
        uint256 userShare = VaultShares(usdcVaultShares).balanceOf(user
            );
        console2.log("userShare: ", userShare);
        vm.stopPrank();

        uint256 guardianShare = VaultShares(usdcVaultShares).balanceOf(
            guardian);
        console2.log("guardianShare: ", guardianShare);

        AllocationData memory allocationDataNew = AllocationData(1000,
            0, 0);
        vm.prank(guardian);
        vaultGuardians.updateHoldingAllocation(usdc, allocationDataNew)
            ;

        //before calling the quit guardian, guardian waits for enough
            tokens to accumulate
        deal(address(usdc), address(usdcVaultShares), 1000 ether);


        vm.startPrank(guardian);
```

```
22          VaultShares(usdcVaultShares).approve(address(vaultGuardians),
                guardianShare);
23          uint256 assetsRecovered = vaultGuardians.quitGuardian(usdc);
24          vm.stopPrank();
25
26          console2.log("assetsRecovered: ", assetsRecovered); //
                158.31263072845481799
27      }
```

Recommended mitigation Minimum allocation per protocol

```
1  +    uint256 public constant MINIMUM_PROTOCOL_ALLOCATION = 100; //10%
2
3  function updateHoldingAllocation(AllocationData memory
       tokenAllocationData) public onlyVaultGuardians isActive {
4           uint256 totalAllocation = tokenAllocationData.holdAllocation +
                tokenAllocationData.uniswapAllocation
5               + tokenAllocationData.aaveAllocation;
6           if (totalAllocation != ALLOCATION_PRECISION) {
7               revert VaultShares__AllocationNot100Percent(totalAllocation
                    );
8           }
9  +        require(tokenAllocationData.uniswapAllocation >=
       MINIMUM_PROTOCOL_ALLOCATION, "Uniswap allocation too low");
10 +        require(tokenAllocationData.aaveAllocation >=
       MINIMUM_PROTOCOL_ALLOCATION, "Aave allocation too low");
11          s_allocationData = tokenAllocationData;
12          emit UpdatedAllocation(tokenAllocationData);
13      }
```

[H-10] Missing `Payable` Modifier in `VaultGuardianBase::becomeGuardian` Causes Loss of Guardian Fees

Description The `becomeGuardian` function is intended to require guardians to pay a fee in `ETH` *(as per the NatSpec documentation and the declared constant GUARDIAN_FEE)*.

However, the function is not marked as `payable`, meaning no `ETH` can be sent along with the call. As a result, the protocol is not collecting the required guardian fees.

Additionally, the variable `GUARDIAN_FEE` is declared but never used, further confirming that fees are not enforced.

Impact The protocol permanently loses revenue from guardian onboarding fees.

Proof of Concepts

1. Call `becomeGuardian(...)` without sending ETH.

2. The function executes successfully, and the caller becomes a guardian without paying the documented `GUARDIAN_FEE`.

```
function testBecomeGuardianWithoutPayingFee() external {
        weth.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        weth.approve(address(vaultGuardians), mintAmount);
        address wethVault = vaultGuardians.becomeGuardian(
           allocationData);
        wethVaultShares = VaultShares(wethVault);
        vm.stopPrank();

        address expectedGuardian = wethVaultShares.getGuardian();
        assertEq(expectedGuardian, guardian);
    }
```

Recommended mitigation

1. Mark `becomeGuardian` as `payable`.
2. Enforce the guardian fee by requiring `msg.value` to equal `GUARDIAN_FEE`.

```
-    function becomeGuardian(AllocationData memory wethAllocationData)
    external returns (address)
+    function becomeGuardian(AllocationData memory wethAllocationData)
    external payable returns (address) {
+        require(msg.value == GUARDIAN_FEE, "Incorrect guardian fee sent
    ");
        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);
    }
```

Medium

[M-1] `UniswapAdapter::_uniswapDivest` returns incorrect asset amount extracted from Uniswap

Description The function returns only `amounts[1]` (tokens received from swap) but ignores `tokenAmount` (tokens received directly from liquidity removal). This results in an incomplete accounting of the total assets returned to the user, potentially causing economic loss in calling contracts that rely on this return value.

Though the return value is not used in the current implementation, it can lead to issues if the function is called by other contracts that expect the total amount returned.

```
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      }
```

Impact

- Incorrect asset accounting leading to potential economic loss
- Calling contracts may miscalculate available assets

Proof of Concepts Assuming a scenario where a user divests liquidity from Uniswap, the function `_uniswapDivest` is called, and the user expects to receive back the total assets (both direct token receipt and swap proceeds).

1. User divests liquidity worth 1000 tokens total.
2. The `removeLiquidity` call returns 500 USDC and 500 WETH equivalent
3. The swap operation converts 500 WETH to ~500 USDC.
4. The function only returns the swap proceeds (~500 USDC) `amountOfAssetReturned = amounts[1]` instead of the total assets (1000 USDC equivalent).

Recommended mitigation

```
-   amountOfAssetReturned = amounts[1];
+   amountOfAssetReturned = tokenAmount + amounts[1];
```

[M-2] `VaultGuardians::updateGuardianAndDaoCut` emits the incorrect event with wrong parameters

Description The `updateGuardianAndDaoCut()` function emits the wrong event (`VaultGuardians__Update`) instead of the appropriate event for **fee cut updates**, and uses incorrect parameters where both values represent the new cut amount rather than the old and new values.

```
function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
        s_guardianAndDaoCut = newCut;
@>      emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut,
    newCut);
     }
```

Impact This creates completely misleading event logs that corrupt off-chain monitoring systems tracking both stake price changes and fee cut modifications.

Proof of Concepts

1. The owner calls `updateGuardianAndDaoCut(2560)`.
2. The emitted event is `VaultGuardians__UpdatedStakePrice(2560, 2560)` instead of a dedicated event like `VaultGuardians__UpdatedGuardianAndDaoCut(oldCut, newCut)`.
3. And both parameters are the same, not reflecting the actual old value of guardian and dao cut.

```
function testUpdateGuardianAndDaoCutIncorrectEmission() external {
        uint256 newGuardianAndDAOCut = 2560;
        vm.prank(vaultGuardians.owner());
        vm.expectEmit(address(vaultGuardians));
        emit VaultGuardians__UpdatedStakePrice(newGuardianAndDAOCut,
            newGuardianAndDAOCut);
        vaultGuardians.updateGuardianAndDaoCut(newGuardianAndDAOCut);
    }
```

Recommended mitigation

```
+  event VaultGuardians__UpdatedGuardianAndDaoCut(uint256 oldCut,
    uint256 newCut);

function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
+       uint256 oldCut = s_guardianAndDaoCut;
        s_guardianAndDaoCut = newCut;
-       emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut,
    newCut);
+       emit VaultGuardians__UpdatedGuardianAndDaoCut(oldCut, newCut);
     }
```

Low

[L-1] `UniswapAdapter::_uniswapInvest` allows excessive token approval beyond required amount

Description The function `_uniswapInvest` approves `amountOfTokenToSwap + amounts[0]` tokens for the Uniswap router during liquidity addition, but `amounts[0]` represents the input amount from the swap (which equals `amountOfTokenToSwap`). This results in approving double the required amount (`2 * amountOfTokenToSwap`).

```
   function _uniswapInvest(IERC20 token, uint256 amount) internal {
        .
        .
        .
@>      succ = token.approve(address(i_uniswapRouter),
    amountOfTokenToSwap + amounts[0]);
        if (!succ) {
            revert UniswapAdapter__TransferFailed();
        }
        .
     }
```

Impact

1. Unnecessary token approval exposure to the router contract enable a contract to spend more tokens than needed.
2. Gas waste from redundant approval
3. Increased risk of potential token loss if the router contract is compromised or misused.

Recommended mitigation Approve only what's needed for liquidity provision

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

```
2 +     succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap)
      ;
```

[L-2] Event Emission Inconsistency in `UniswapAdapter::_uniswapInvest`

Description The emitted `UniswapInvested` event uses `counterPartyTokenAmount` as the second parameter, but the event definition says it should specifically be `wethAmount`. This creates inconsistency between definition and implementation, potentially confusing external monitoring systems.

```
1     event UniswapInvested(uint256 tokenAmount, uint256 wethAmount,
          uint256 liquidity);
2     emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
          liquidity);
```

The `counterPartyToken` can be any token WETH or USDC depending on the token being invested, so the event should be updated to reflect this.

Impact

- Monitoring and analytics systems may misinterpret event data
- Increases risk of incorrect assumptions about the event data.
- Reduces maintainability of the codebase.

Proof of Concepts

Showing the discrepancy in the event emission of `UniswapInvested` via the `becomeGuardian` function call:

```
1  function testDiscrepancyInEventEmission() external {
2          weth.mint(mintAmount, guardian);
3          uint256 amountToDeosit = stakePrice; //WETH amount
4          uint256 amountAddedToUniswap = amountToDeosit * allocationData.
               uniswapAllocation / 1000;
5          vm.startPrank(guardian);
6          weth.approve(address(vaultGuardians), mintAmount);
7          vm.expectEmit();
8          //event UniswapInvested(uint256 tokenAmount, uint256 wethAmount
               , uint256 liquidity);
9          emit UniswapInvested(amountAddedToUniswap, 0, 0);
10         address wethVault = vaultGuardians.becomeGuardian(
               allocationData);
11         wethVaultShares = VaultShares(wethVault);
12         vm.stopPrank();
13     }
```

Recommended mitigation

1. Update event definition to match implementation

```
-   event UniswapInvested(uint256 tokenAmount, uint256 wethAmount,
    uint256 liquidity);
+   event UniswapInvested(uint256 tokenAmount, uint256
    counterPartyTokenAmount, uint256 liquidity);
```

2. Or update emission to match documentation (if WETH is always expected as second parameter):

```
-   emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
    liquidity);

+   if (token == i_weth) {
+       emit UniswapInvested(counterPartyTokenAmount, tokenAmount,
    liquidity);
+   } else {
+       emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
    liquidity);
+   }
```

[L-3] Event Emission Inconsistency in `UniswapAdapter::_uniswapDivest`

Description The emitted `UniswapDivested` event uses `amounts[1]` as the second parameter, but the event definition says it should specifically be `wethAmount`. This creates inconsistency between definition and implementation, potentially confusing external monitoring systems.

```
    event UniswapDivested(uint256 tokenAmount, uint256 wethAmount);
    emit UniswapDivested(tokenAmount, amounts[1]);
```

Impact

- External systems expecting WETH amounts will receive incorrect data
- Event monitoring and analytics systems may misinterpret token flows

Proof of Concepts

1. Token = USDC, counterPartyToken = WETH,
2. WETH are swapped for USDC, amounts[1] is NOT WETH rather USDC
3. This leads to confusion as the event suggests WETH amounts, but it is actually USDC amounts.

Recommended mitigation

1. Update event definition to match implementation

```
-   event UniswapDivested(uint256 tokenAmount, uint256 wethAmount);
```

```
2 +    event UniswapDivested(uint256 tokenAmount, uint256
      counterPartyTokenAmount);
```

2. Emit consistent WETH amounts

```
1 -    emit UniswapDivested(tokenAmount, amounts[1]);
2 +    if (token == i_weth) {
3 +        emit UniswapDivested(counterPartyTokenAmount, tokenAmount);
4 +    } else {
5 +        emit UniswapDivested(tokenAmount, counterPartyTokenAmount);
6 +    }
```

[L-4] Fee share can truncate to zero for small deposits bypassing protocol revenue

Description The fee calculation using integer division (`shares / i_guardianAndDaoCut`) truncates to zero when the user's share amount is smaller than the `i_guardianAndDaoCut` value.

This allows users to make small deposits without paying any fees to the guardian and DAO, completely bypassing the intended fee mechanism and depriving the protocol of revenue.

Impact

- Unfair advantage for users making multiple small deposits vs. single large deposit
- Protocol loses fee revenue from small deposits

Proof of Concepts

1. User intends to deposit a small amount of WETH (e.g., 899 wei).
2. The `previewDeposit` function calculates shares based on the current total assets and returns a share amount that is less than `i_guardianAndDaoCut`.
3. Resulting fee shares calculated as (`userShares / i_guardianAndDaoCut`) truncates to zero.

```
1  function testPrecisionLossDueToSmallDeposits() external {
2          AllocationData memory allocationDataNew = AllocationData(1000,
               0, 0);
3
4          weth.mint(mintAmount, guardian);
5          vm.startPrank(guardian);
6          weth.approve(address(vaultGuardians), mintAmount);
7          address wethVault = vaultGuardians.becomeGuardian(
               allocationDataNew);
8          wethVaultShares = VaultShares(wethVault);
9          vm.stopPrank();
10
11         uint256 amountToInvest = 899;
```

```
12          weth.mint(amountToInvest, user);
13          vm.startPrank(user);
14          weth.approve(address(wethVaultShares), amountToInvest);
15          uint256 userShares = wethVaultShares.previewDeposit(
                amountToInvest);
16          uint256 feeSharesEach = userShares / wethVaultShares.
                getGuardianAndDaoCut();
17          uint256 feeShares = feeSharesEach * 2;
18          // This will truncate to zero if userShares <
                i_guardianAndDaoCut
19          assertEq(feeShares, 0);
20      }
```

Recommended mitigation

1. Set a minimum deposit threshold for refraining users from depositing dust amounts to avoid fee truncation to zero.
2. Alternatively, revert when the shares are zero.

[L-5] `VaultGuardians::VaultGuardians__UpdatedStakePrice` is emitted with incorrect parameters

Description The `VaultGuardians::updateGuardianStakePrice()` function emits the `VaultGuardians_UpdatedStakePrice` event with incorrect parameters.

The event expects (`oldStakePrice, newStakePrice`) but the function emits (`s_guardianStakePrice, newStakePrice`).

Since `s_guardianStakePrice` is updated to `newStakePrice` before the event emission, both parameters contain the same new value, making it impossible to track the actual price change history.

```
1  function updateGuardianStakePrice(uint256 newStakePrice) external
       onlyOwner {
2          s_guardianStakePrice = newStakePrice;
3  @>      emit VaultGuardians__UpdatedStakePrice(s_guardianStakePrice,
       newStakePrice);
4      }
```

Impact

- Misleading event logs.
- Loss of historical stake price change tracking
- Inaccurate data for off-chain monitoring and analytics.

Proof of Concepts

```
function testUpdateStakePriceIncorrectEmission() external {
        uint256 newStakePrice = 2450;
        vm.prank(vaultGuardians.owner());
        vm.expectEmit(address(vaultGuardians));
        // Emitting VaultGuardians__UpdatedStakePrice with incorrect parameters
        // It should emit (oldStakePrice, newStakePrice) but emits (newStakePrice, newStakePrice)
        emit VaultGuardians__UpdatedStakePrice(newStakePrice, newStakePrice);
        vaultGuardians.updateGuardianStakePrice(newStakePrice);
    }
```

Recommended mitigation

```
function updateGuardianStakePrice(uint256 newStakePrice) external onlyOwner {
+       uint256 oldStakePrice = s_guardianStakePrice;
        s_guardianStakePrice = newStakePrice;
-       emit VaultGuardians__UpdatedStakePrice(s_guardianStakePrice, newStakePrice);
+       emit VaultGuardians__UpdatedStakePrice(oldStakePrice, newStakePrice);
    }
```

[L-6] `VaultGuardians::constructor` does not validate addresses against zero address

Description The constructor accepts six critical address parameters (`aavePool`, `uniswapV2Router`, `weth`, `tokenOne`, `tokenTwo`, `vaultGuardiansToken`) without performing `zero address` validation.

If any of these addresses are accidentally set to `address(0)` during deployment, the contract will be permanently deployed with invalid addresses that cannot be updated, rendering the contract partially or completely non-functional.

Impact Contract becomes permanently unusable if core protocol addresses are zero

Recommended mitigation

```
+   error ZeroAddress(string paramName);
constructor(
        address aavePool,
        address uniswapV2Router,
        address weth,
        address tokenOne,
        address tokenTwo,
        address vaultGuardiansToken
```

```
9       )
10          Ownable(msg.sender)
11          VaultGuardiansBase(aavePool, uniswapV2Router, weth, tokenOne,
                tokenTwo, vaultGuardiansToken)
12      {
13  +     if (aavePool == address(0)) revert ZeroAddress("aavePool");
14  +     if (uniswapV2Router == address(0)) revert ZeroAddress("
        uniswapV2Router");
15  +     if (weth == address(0)) revert ZeroAddress("weth");
16  +     if (tokenOne == address(0)) revert ZeroAddress("tokenOne");
17  +     if (tokenTwo == address(0)) revert ZeroAddress("tokenTwo");
18  +     if (vaultGuardiansToken == address(0)) revert ZeroAddress("
        vaultGuardiansToken");
19      }
```

[L-7] `VaultGuardians::sweepErc20s` allows anyone to drain contract balances to owner() by passing any IERC20 address this can lead to non-standard ERC20 token attack

Description The `sweepErc20s` function allows the contract owner to transfer any ERC20 tokens held by the contract to the owner's address. However, it does not validate whether the token is a standard ERC20 or if it has any malicious behavior.

```
1  function sweepErc20s(IERC20 asset) external {
2          uint256 amount = asset.balanceOf(address(this));
3          emit VaultGuardians__SweptTokens(address(asset));
4          asset.safeTransfer(owner(), amount);
5      }
```

Impact

- This can become a theft vector if a malicious actor becomes the contract owner and sweeps tokens that are not meant for someone else.
- This function with non-standard ERC20s, calling `balanceOf` or `safeTransfer` could lead to unexpected behavior or even loss of funds.

Recommended mitigation

1. Restrict the function to only allow sweeping of specific, known ERC20 tokens.
2. Add a whitelist mechanism to ensure only approved tokens can be swept.
3. Add a reentrancy guard (nonReentrant) to sweepErc20s.

Informational

[I-1] Empty Interface Definition for `IInvestableUniverseAdapter` with unused import

Description The `IInvestableUniverseAdapter` interface is essentially empty and does not define any methods or properties. This can lead to confusion and does not provide any functionality.

This creates dead code that unnecessarily increases the contract's bytecode size and deployment costs while providing no functional value.

Impact

- Creates confusion for developers regarding its intended purpose.
- Increases complexity in the codebase without contributing any functionality.
- Adds unnecessary bytecode and deployment overhead.

Recommended mitigation

- Remove the empty interface if it is not required.
- If it is intended to serve as a contract abstraction, define meaningful methods and properties that align with its purpose.
- Eliminate unused imports to maintain a clean and maintainable codebase.

[I-2] Empty Interface Definition for `IVaultGuardians`

Description The `IVaultGuardians` interface is declared but does not define any methods or properties.

Impact

- May confuse developers about its intended role.
- Adds unnecessary complexity to the codebase without any functionality.
- Slightly increases bytecode and deployment overhead.

Recommended mitigation

- Remove the empty interface if it has no functional use.
- If it is intended as a placeholder for future contract abstractions, define the relevant methods and properties that reflect its purpose.

[I-3] Missing NatSpec Documentation for Interface `IVaultShares`

Description The interface `IVaultShares` is defined without NatSpec (/// or /*_... _/) documentation for its purpose or intended usage

Impact

- Lowers maintainability and readability of the codebase.
- Increases the risk of incorrect assumptions or misuse by developers.
- Reduces clarity for auditors and integrators relying on the interface.

Recommended mitigation Add NatSpec documentation for all interfaces

[I-4] Unused custom errors in the codebase

Description The contract defines custom error types that are never referenced or thrown anywhere in the codebase.

These unused error definitions are compiled into the contract bytecode, increasing deployment costs and overall contract size without providing any functional benefit.

```
1     //VaultGuardians.sol
2     error VaultGuardians__TransferFailed();
3     //VaultGuardiansBase.sol
4     error VaultGuardiansBase__NotEnoughWeth(uint256 amount, uint256
          amountNeeded);
5     //VaultGuardiansBase.sol
6     error VaultGuardiansBase__CantQuitGuardianWithNonWethVaults(address
          guardianAddress);
7     //VaultGuardiansBase.sol
8     error VaultGuardiansBase__FeeTooSmall(uint256 fee, uint256
          requiredFee);
```

Impact Increased deployment gas costs due to unused error selectors in bytecode

Recommended mitigation Remove unused custom error definitions from the codebase

[I-5] Unused event definitions in the codebase

Description The contract declares events that are never emitted anywhere in the codebase. This leads to incomplete event monitoring and potential integration failures.

```
1     //VaultGuardians.sol
2     event VaultGuardians__UpdatedFee(uint256 oldFee, uint256 newFee);
3     //VaultGuardiansBase.sol
```

```
4       event InvestedInGuardian(address guardianAddress, IERC20 token,
            uint256 amount);
5       //VaultGuardiansBase.sol
6       event DinvestedFromGuardian(address guardianAddress, IERC20 token,
            uint256 amount);
```

Impact

- Off-chain systems expect events that are never emitted
- Increased contract deployment costs from unused event metadata

Recommended mitigation Remove unused event definitions or implement their emission where appropriate.