Vault Guardians Audit Report

Version 1.0

github.com/lucasfhope

October 27, 2025

Vault Guardians Audit Report

Lucas Hope

October 27, 2025

Prepared by: Lucas Hope

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. You can think of a vaultGuardian as a fund manager. The vaultGuardian can allocate the funds into Aave v3, Uniswap v2, and/or just hold the funds.

Disclaimer

Lucas Hope 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
	High	H	H/M	M
Likelihood	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

The findings described in this report corresponds to the following commit hash:

```
0600272180b6b6103f523b0ef65b070064158303
```

Scope

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

Roles

- The DAO: Guardians can earn DAO tokens by becoming guardians.
  - Update pricing parameters
  - Get a cut of all performance of all guardians
- Vault Guardian: Anyone can open a vault and become a vault guardian by contributing a sufficient stake.
  - Update vault allocations
  - Can charge a performance fee
  - Can quit as a guardian and get their stake back
- Investor: Can invest in any vault run by a vault guardian
  - Get shares of the vault based on what they invested
  - Can redeem their shares for their share of the vault

Executive Summary

The audit found many critical vulnerabilities that the protocol must address. This includes, but is not limited to, lack of slippage protection that creates opportunities for MEV frontrunning and sandwich attacks. There are also issues with function logic that neglect the protocol's claims.

Issues found

Severity	Number of issues found
High	5
Medium	3
Low	2
Info	6
Gas	1
Total	**17**

Findings

High

[H-1] WETH vaults will set the Uniswap pairing to WETH, resulting in liquidity token address being address(0), making it impossible to divest from Uniswap

Description: Vaults that are set to hold WETH will attempt to get the liquidity token between the pairing between WETH and WETH. Since this is an invalid pairing, the returned address will be address(0).

```
constructor(ConstructorData memory constructorData)
    ERC4626(constructorData.asset)
    ERC20(constructorData.vaultName, constructorData.vaultSymbol)
    AaveAdapter(constructorData.aavePool)
    UniswapAdapter(constructorData.uniswapRouter, constructorData.weth,
        constructorData.usdc)
{
    ...

    // External calls
    i_aaveAToken =
        IERC20(IPool(constructorData.aavePool).getReserveData(address(
            constructorData.asset)).aTokenAddress);
--> i_uniswapLiquidityToken = IERC20(i_uniswapFactory.getPair(address(
    constructorData.asset), address(i_weth)));
}
```

The function `divestThenInvest` will only attempt to divest from Uniswap if there are liquidity tokens in the vault. Since the liquidity token is address(0), it will never find a balance in the vault.

```
modifier divestThenInvest() {
--> uint256 uniswapLiquidityTokensBalance = i_uniswapLiquidityToken.
    balanceOf(address(this));
    uint256 aaveAtokensBalance = i_aaveAToken.balanceOf(address(this));

    // Divest
--> if (uniswapLiquidityTokensBalance > 0) {
        _uniswapDivest(IERC20(asset()), uniswapLiquidityTokensBalance);
    }

    ...
}
```

Impact: Funds allocated to Uniswap in a WETH vault will never be able to be divested.

Proof of Concept: Set up a fork test contract shown below with the added modifier and test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.19 <0.9.0;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/
    extensions/IERC20Metadata.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";

import {Fork_Test} from "./Fork.t.sol";
import {console} from "forge-std/Test.sol";

contract AuditForkTest is Fork_Test {
    address public guardian = makeAddr("guardian");
    address public user = makeAddr("user");

    VaultShares public wethVaultShares;

    uint256 mintAmount = 100 ether;

    function setUp() public virtual override {
        Fork_Test.setUp();
    }

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

    function testBadWethPairUniswap() public hasGuardianWeth {
        assert(wethVaultShares.getUniswapLiquidtyToken() == address(0))
            ;
    }
}
```

This test will show that the Uniswap liquidity token is set as address(0) in the contract.

Recommended Mitigation: Make sure that WETH vaults set the liquidity token pairing with another token.

```
    constructor(ConstructorData memory constructorData)
        ERC4626(constructorData.asset)
        ERC20(constructorData.vaultName, constructorData.vaultSymbol)
        AaveAdapter(constructorData.aavePool)
        UniswapAdapter(constructorData.uniswapRouter, constructorData.
            weth, constructorData.usdc)
    {
        ...

        // External calls
        i_aaveAToken =
            IERC20(IPool(constructorData.aavePool).getReserveData(
                address(constructorData.asset)).aTokenAddress);
-       i_uniswapLiquidityToken = IERC20(i_uniswapFactory.getPair(
    address(constructorData.asset), address(i_weth)));
+       if(constructorData.asset == address(i_weth)) {
+           i_uniswapLiquidityToken = IERC20(i_uniswapFactory.getPair(
    address(constructorData.asset), address(constructorData.usdc)));
+       } else {
+           i_uniswapLiquidityToken = IERC20(i_uniswapFactory.getPair(
    address(constructorData.asset), address(i_weth)));
+       }
    }
```

[H-2] Pair tokens are not approved before swapping in `UniswapAdapter_uniswapDivest`, causing the function to always revert

Description: The function `_uniswapDivest` does not approve the Uniswap contract to handle liquidity tokens when redeeming liquidity. The function also fails to approve the pair token that is being swapped back to the vault token.

```
// missing approval for the liquidity token
(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)];
// missing approval for the pair token
uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
    amountIn: counterPartyTokenAmount,
    amountOutMin: 0,
    path: s_pathArray,
    to: address(this),
    deadline: block.timestamp
});
```

Impact: `_uniswapDivest` will always revert, meaning the `VaultShares::divestThenInvest` will also always revert. Because `divestThenInvest` handles all divesting within the vault, funds that are allocated to either protocol will never be able to be recovered.

Proof of Concept: Set up a fork test contract shown below with the added modifiers and test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.19 <0.9.0;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/
    extensions/IERC20Metadata.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";

import {Fork_Test} from "./Fork.t.sol";
import {console} from "forge-std/Test.sol";

contract AuditForkTest is Fork_Test {
    address public guardian = makeAddr("guardian");
    address public user = makeAddr("user");

```

```solidity
    VaultShares public wethVaultShares;
    VaultShares public usdcVaultShares;

    uint256 stakePrice;

    uint256 mintAmount = 100 ether;
    uint256 mintAmountUSDC = 100000e6;

    AllocationData allocationData = AllocationData(500, 250, 250);
    AllocationData newAllocationData = AllocationData(0, 500, 500);

    function setUp() public virtual override {
        Fork_Test.setUp();
    }

    modifier lowerStakePrice() {
        stakePrice = vaultGuardians.getGuardianStakePrice();
        vm.prank(vaultGuardians.owner());
        vaultGuardians.updateGuardianStakePrice(mintAmountUSDC); //
            100000 USDC
        _;
    }

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

    modifier hasGuardianUSDC() {
        deal(address(usdc), guardian, mintAmountUSDC);
        vm.startPrank(guardian);
        usdc.approve(address(vaultGuardians), mintAmountUSDC);
        address usdcVault = vaultGuardians.becomeTokenGuardian(
            allocationData, usdc);
        usdcVaultShares = VaultShares(usdcVault);
        vm.stopPrank();
        _;
    }

    function testDivestRevertsForNoApprovalUniswap() public
        lowerStakePrice hasGuardianWeth hasGuardianUSDC {
        vm.expectRevert();
        usdcVaultShares.rebalanceFunds();
        vm.stopPrank();
    }
```

```
}
```

This test will revert because Uniswap attempts to transfer liquidity tokens from the vault contract, but the vault contract never approves the liquidity tokens for Uniswap to handle.

Recommended Mitigation: Include token approvals in `_uniswapDivest`.

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

+       IERC20 liquidityToken = IERC20(i_uniswapFactory.getPair(address(token), address(counterPartyToken)));
+       liquidityToken.approve(address(i_uniswapRouter), liquidityAmount);

        (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)];

+       IERC20(counterPartyToken).approve(address(i_uniswapRouter), counterPartyTokenAmount);

        uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
            amountIn: counterPartyTokenAmount,
            amountOutMin: 0,
            path: s_pathArray,
            to: address(this),
            deadline: block.timestamp
        });
        emit UniswapDivested(tokenAmount, amounts[1]);
        amountOfAssetReturned = amounts[1];
    }
```

[H-3] Lack of slippage protecton in `UniswapAdapter`, allowing for frontruns and sandwich attacks to create bad swap rates.

Description: In `UniswapAdapter`, swapping and providing liquidity in `_uniswapInvest`, and redeeming liquidity and swapping in `_uniswapDivest` lack proper slippage protection. For example, in `_uniswapInvest`, the vault token in swapped for a pair token before adding liquidity to Uniswap. However, the parameter `amountOutMin` is set to 0, which means that the swap could result in receiving a small amount of the pair token.

Similarly, using block.timestamp as the deadline allows a maliocious validator to push back the transaction to a more favorable block number.

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

This opens the transaction to be frontrun. An MEV bit could see your transaction, take out a flash loan, and skew the balance in Uniswap. If this happens, the vault would receive much less of the pair token then would be expected. This lack of slippage protection also applies when providing/redeeming liquidity and receiving/sending liquidity tokens.

Impact: Uniswap balances can be manipulated before the transaction, resulting in lost funds.

Recommended Mitigation: Add slippage protection when interacting with the Uniswap protocol.

[H-4] Opening a vault mints DAO tokens to guardians, allowing a malicious gaurdian to take over the DAO and steal funds.

Description: Vault guardians opening a new vault will be minted DAO tokens in `VaultGuardiansBase::_becomeTokenGuardian`. There is also no limit on how many vaults can be opened, as any new vault for the same token will just overwrite the old vault in the `s_guardians` mapping.

```
function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
    private returns (address) {
--> s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));
    emit GuardianAdded(msg.sender, token);
--> i_vgToken.mint(msg.sender, s_guardianStakePrice);
    token.safeTransferFrom(msg.sender, address(this),
        s_guardianStakePrice);

    ...
}
```

Impact: Vault guardians can open many vaults and mint themselves many DAO tokens. This could be used to take control of the DAO and steal funds from the protocol. It is also possible for vaults to be lost in the system if a guardian opens a new vault with the same token, which will overwrite the original vault in the `s_guardians` mapping.

Recommended Mitigation: Do not mint DAO tokens to guardians when they open a vault. Consider using a different machanism to distrubute or sell DAO tokens, and it may be in the best interest of the protocol to mint all DAO tokens upon deployment of the contract and not allow further minting.

Also consider preventing guardians from opening a new vault while having an active vault that holds the same token.

[H-5] Invested funds cause issues when calculating share balances, and the guardian and DAO take shares when a user deposits.

Description: When users deposit funds, invested vault funds are not divested, so shares will be calculated based on the reserves held in the vault rather than the total balance. Similarly, any ERC4626 function that relies on `totalAssets()` will base their calculations off of only the funds held in reserve.

Furthermore, the vault gauardian and the DAO receive extra shares during every deposit. This will affect the share percentage of the user depositing, letting the guardian and DAO take a percentage of the initially deposited funds rather than just receiving a share of the profits.

Impact: Shares will be diiluted and mismanaged, affecting the percentage of the vault that any user will own.

Proof of Code: Set up a fork test contract shown below with the added modifiers and test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.19 <0.9.0;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/
    extensions/IERC20Metadata.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";

import {Fork_Test} from "./Fork.t.sol";
import {console} from "forge-std/Test.sol";

contract AuditForkTest is Fork_Test {
    address public guardian = makeAddr("guardian");
    address public user = makeAddr("user");

    VaultShares public wethVaultShares;
    VaultShares public usdcVaultShares;

    uint256 stakePrice;

    uint256 mintAmount = 100 ether;

    AllocationData allocationData = AllocationData(500, 250, 250);
    AllocationData newAllocationData = AllocationData(0, 500, 500);

    function setUp() public virtual override {
        Fork_Test.setUp();
    }

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

    modifier userIsInvestedWeth() {
        deal(address(weth), user, mintAmount);
        vm.startPrank(user);
        weth.approve(address(wethVaultShares), mintAmount);
        wethVaultShares.deposit(mintAmount, user);
        vm.stopPrank();
        _;
    }

    function testTotalAssetsDoesNotIncludeInvestedFunds() public
       hasGuardianWeth userIsInvestedWeth {
        // vault has been given 10 ether + 100 ether
        uint256 totalAssets = wethVaultShares.totalAssets();
        console.log("Total Assets: ", totalAssets);
        assert(totalAssets < 110 ether);
    }
}
```

This test will show that `totalAssets()` will only return as half of the deposited vault balance because the rest is invested.

Recommended Mitigation: The guardian and DAO fees should be reworked to take a percentage from the profit, or just take a fee on any withdrawal instead of receiving extra shares of the vault. There should also be a way to calculate the current value of invested assets to calculate the true share percentage of the vault.

Medium

[M-1] Guardian stake price is too high for USDC vaults to be created.

Description: The guardian stake price required to open a vault is 10e18. While this value is normal for 18 decomal tokens (WETH, LINK), it is an extremely large value for a 6 decimal token (USDC). A guardian would need to deposit 1e13 USDC to become a guardian of a USDC vault. If a guardian attempts to open a USDC vault, it may revert while attempting to supply too much liquidity to Aave.

Impact: USDC vaults will not be able to be opened.

Proof of Concept: Set up a fork test contract shown below with the added modifier and test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.19 <0.9.0;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/
    extensions/IERC20Metadata.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";

import {Fork_Test} from "./Fork.t.sol";
import {console} from "forge-std/Test.sol";

contract AuditForkTest is Fork_Test {
    address public guardian = makeAddr("guardian");
    address public user = makeAddr("user");

    VaultShares public wethVaultShares;
    VaultShares public usdcVaultShares;

    uint256 stakePrice;

    uint256 mintAmount = 100 ether;
    uint256 mintAmountUSDC = 100000e6;
    uint256 mintBrokenUSDCAmount = 10 ether;

    AllocationData allocationData = AllocationData(500, 250, 250);
    AllocationData newAllocationData = AllocationData(0, 500, 500);

    function setUp() public virtual override {
        Fork_Test.setUp();
    }

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

    function testHighUsdcStakePrice() public hasGuardianWeth {
        deal(address(usdc), guardian, mintBrokenUSDCAmount);
        vm.startPrank(guardian);
        usdc.approve(address(vaultGuardians), mintBrokenUSDCAmount);
        vm.expectRevert();
        address usdcVault = vaultGuardians.becomeTokenGuardian(
            allocationData, usdc);
        vm.stopPrank();
    }
}
```

This test shows the initial invest revert when attempting to supply Aave with 2.5e18 USDC.

Recommended Mitigation: Have a different stake price for each token.

[M-2] More funds than allocated are deposited into Uniswap.

Description: `UniswapAdapter::_uniswapInvest` intends to swap half of the allocated funds into the pair token and add liquidity to balance maintain the relative balance of the DEX. Instead, `_uniswapInvest` sends the amount of pair tokens from the swap as `amountBDesired` and sends `amountOfTokenToSwap + amounts[0]` as `amountADesired` . Since `amountOfTokenToSwap` is half of the tokens allocated to Uniswap and `amounts[0]` was the input to swap to the pair token, `_uniswapInvest` is investing 1.5 times the intended amount into Uniswap.

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

...

(uint256 tokenAmount, uint256 counterPartyTokenAmount, uint256
    liquidity) = i_uniswapRouter.addLiquidity({
```

```
    tokenA: address(token),
    tokenB: address(counterPartyToken),
--> amountADesired: amountOfTokenToSwap + amounts[0],   /// @audit
   overspend
    amountBDesired: amounts[1],
    amountAMin: 0,
    amountBMin: 0,
    to: address(this),
    deadline: block.timestamp
});
```

Impact: More funds will be allocated to Uniswap than intended. This will result in less cash being sent in reserve, or reallocation can revert for certain allocations.

Proof of Concept: Set up a fork test contract shown below with the added modifier and test.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.19 <0.9.0;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/
   extensions/IERC20Metadata.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";

import {Fork_Test} from "./Fork.t.sol";
import {console} from "forge-std/Test.sol";

contract AuditForkTest is Fork_Test {
    address public guardian = makeAddr("guardian");
    address public user = makeAddr("user");

    VaultShares public wethVaultShares;
    VaultShares public usdcVaultShares;

    uint256 stakePrice;

    uint256 mintAmount = 100 ether;
    uint256 mintAmountUSDC = 100000e6;
    uint256 mintBrokenUSDCAmount = 10 ether;

    AllocationData allocationData = AllocationData(500, 250, 250);
    AllocationData newAllocationData = AllocationData(0, 500, 500);

    function setUp() public virtual override {
        Fork_Test.setUp();
    }

    modifier lowerStakePrice() {
        stakePrice = vaultGuardians.getGuardianStakePrice();
        vm.prank(vaultGuardians.owner());
        vaultGuardians.updateGuardianStakePrice(mintAmountUSDC); //
```

```
 100000 USDC
        _;
    }

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

    modifier hasGuardianUSDC() {
        deal(address(usdc), guardian, mintAmountUSDC);
        vm.startPrank(guardian);
        usdc.approve(address(vaultGuardians), mintAmountUSDC);
        address usdcVault = vaultGuardians.becomeTokenGuardian(allocationData, usdc);
        usdcVaultShares = VaultShares(usdcVault);
        vm.stopPrank();
        _;
    }

    modifier userIsInvestedUSDC() {
        vm.startPrank(user);
        deal(address(usdc), user, mintAmountUSDC);
        usdc.approve(address(usdcVaultShares), mintAmountUSDC);
        usdcVaultShares.deposit(mintAmountUSDC, user);
        vm.stopPrank();
        _;
    }


    function testOverspendOnUniswap() public lowerStakePrice hasGuardianWeth hasGuardianUSDC userIsInvestedUSDC {
        AllocationData memory updatedAllocationData = AllocationData(0, 900, 100); // 90/10 uniswap/aave allocation

        uint256 allocationPrecision = 1000;
        uint256 vaultBalance = weth.balanceOf(address(wethVaultShares));
        uint256 uniswapAllocation = (vaultBalance * updatedAllocationData.uniswapAllocation) / allocationPrecision;
        uint256 aaveAllocation = (vaultBalance * updatedAllocationData.aaveAllocation) / allocationPrecision;
        uint256 totalAllocation = uniswapAllocation + aaveAllocation;
        console.log("Vault Balance:      ", vaultBalance);
        console.log("Expected Uniswap Allocation: ", uniswapAllocation)
            ;
        vm.startPrank(guardian);
        vaultGuardians.updateHoldingAllocation(
            usdc, updatedAllocationData
        );
        vm.expectRevert();
        usdcVaultShares.rebalanceFunds();
        vm.stopPrank();
    }
}
```

This test will revert because it invests too much of the vault into Uniswap and cannot supply Aave with the expected amount of funds.

Recommended Mitigation: Remove the extra funds being send to Uniswap in `_uniswapInvest`.

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

It may also be benificial to make sure that calculation of `aaveAllocation` and `uniswapAllocation` in `VaultShares::_investFunds` does not exceed the vault balance.

[M-3] `VaultShares::divestThenInvest` is inefficient for compounding investments and `VaultShares::rebalanceFunds` allows for griefing.

Description: The modifier `divestThenInvest` is used to divest all funds before certain actions like withdrawing from the vault. That means that every time someone withdraws/redeems funds from the vault, all funds allocated to Uniswap or Aave will be divested from them. This forces high gas fees upon the caller and defeats the purpose of providing liquidity to these protocols, since it sacrifices compounding and incurs fee payments.

Griefing is also possible by forcing the vault to divest all its assets. This will similarly hurt compounding investments and force the vault to pay fees.

Impact: The vault must pay fees and sacrifice compounding investments every time the vault must divest the funds.

Recommended Mitigation: Consider requiring the vault to have funds in reserve so the vault can payout certain withdraw requests without having to divest. The vault should also be able to calculate the exact amount required to divest such that all funds are not divested for every withdrawal, just the amount neccessary.

Also condiser only allowing the guardian to rebalance funds. Since the protocol is set such that guardians have control over the vaults, they should also control when funds get rebalanced, eliminating outside griefing.

Low

[L-1] Wrong vault name set when creating a vault for token two

Description: When creating a vault for `i_tokenTwo`, the vault's name and symbol are set as `TOKEN_ONE_VAULT_NAME` and `TOKEN_ONE_VAULT_SYMBOL`.

Recommended Mitigation:

```
else if (address(token) == address(i_tokenTwo)) {
    tokenVault =
    new VaultShares(IVaultShares.ConstructorData({
        asset: token,
-       vaultName: TOKEN_ONE_VAULT_NAME,
-       vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
+       vaultName: TOKEN_TWO_VAULT_NAME,
+       vaultSymbol: TOKEN_TWO_VAULT_SYMBOL,
        guardian: msg.sender,
        allocationData: allocationData,
        aavePool: i_aavePool,
        uniswapRouter: i_uniswapV2Router,
        guardianAndDaoCut: s_guardianAndDaoCut,
        vaultGuardians: address(this),
        weth: address(i_weth),
        usdc: address(i_tokenOne)
    }));
}
```

[L-2] `VaultGuardians::sweepErc20s` will never collect and tranfer any tokens

Description: The function `sweepErc20s` will not be able to send any tokens to the DAO. This is because not tokens are ever handled or sent to `VaultGuardians`, and instead are sent to `VaultGuardiansBase`.

Recommended Mitigation: Consider having DAO fees and shares sent to `VaultGuardians` rather than `VaultGuardiansBase`. It is also possible to set up a machanism to sweep ERC20s from `VaultGuardiansBase` instead.

Informational

[I-1] `AaveAdapter::_aaveDivest` is set to return a `uint256` but never returns anything

This return value is never used, so it can be removed from the function.

```diff
-   function _aaveDivest(IERC20 token, uint256 amount) internal returns
    (uint256 amountOfAssetReturned) {
+   function _aaveDivest(IERC20 token, uint256 amount) internal {
        i_aavePool.withdraw({
            asset: address(token),
            amount: amount,
            to: address(this)
        });
    }
```

[I-2] `VaultGuardianBase::_quitGuardians` removes the vault address from `s_guardians`, which can make it difficult for users invested in the vault to redeem their funds.

When a guardian quits a vault, they redeem their shares and the vault is set to address(0) in the `s_guardians` array. This can make it difficult for users invested in the vault to locate the vault contract and redeem their funds.

Consider tracking vaults that have been quit for the sake of the users invested.

[I-3]: `nonReentrant` is Not the First Modifier

To protect against reentrancy in other modifiers, the `nonReentrant` modifier should be the first modifier in the list of modifiers.

4 Found Instances

- Found in src/protocol/VaultShares.sol Line: 144

  ```
          nonReentrant
  ```

- Found in src/protocol/VaultShares.sol Line: 181

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

- Found in src/protocol/VaultShares.sol Line: 193

```
        nonReentrant
```

- Found in src/protocol/VaultShares.sol Line: 210

```
        nonReentrant
```

[I-4]: Unused Error

Consider using or removing the unused error.

4 Found Instances

- Found in src/protocol/VaultGuardians.sol Line: 43

```
    error VaultGuardians__TransferFailed();
```

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

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

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

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

- Found in src/protocol/VaultGuardiansBase.sol Line: 51

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

[I-5]: Unused State Variable

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

1 Found Instances

- Found in src/protocol/VaultGuardiansBase.sol Line: 65

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

[I-6]: Unused Import

Redundant import statement. Consider removing it.

1 Found Instances

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

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

Gas

[G-1]: Public Function Not Used Internally

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

2 Found Instances

- Found in src/protocol/VaultShares.sol Line: 115

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

- Found in src/protocol/VaultShares.sol Line: 181

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