



Vault Guardians Audit Report

Version 0.1

November 19, 2025

Vault Guardians Audit Report

mhng

November 19, 2025

Vault Guardians Audit Report

Prepared by: mhng Lead Auditors: mhng

- mhng

Assisting Auditors:

- None

Table of contents

See table

- Vault Guardians Audit Report
- Table of contents
- About mhng
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
- Protocol Summary
 - Roles
- Executive Summary

- Issues found
- Findings
 - High
 - * [H-1] totalAssets() ignores all invested funds which leads to severe disruptions of protocol functionality
 - * [H-2] Missing decimal normalization causes reverts when supplying non 18 decimal tokens (USDC) to Aave
 - * [H-3] Missing counterparty token approvals in _uniswapDivest result in redeem DoS and locked funds
 - * [H-4] Guardians have no restrictions on minting vgToken and can accumulate high amounts to take over DAO, steal DAO fees and set parameters for own interest
 - * [H-5] Double accounting of swap amount leads to excess token spending and fund loss or DoS
 - * [H-6] No slippage protection in uniswap operations in UniswapAdapter can lead to loss of funds due to price changes
 - * [H-7] Uniswap LP token can be address(0) when vault asset is weth(weth/weth pair), causing functional failure and potential token loss.
 - * [H-8] nonReentrant is not the First Modifier in VaultShares which makes the functions vulnerable to reentrancy attacks
 - * [H-9] Using block.timestamp for swap deadline offers no protection
 - Medium
 - * [M-1] Using unsafe ERC20 approve operation can result in stolen assets
 - * [M-2] Centralization Risk
 - * [M-3] Missing validation for address(0) in VaultShares constructor
 - * [M-4] Anyone can force rebalances causing value leakage and fund instability
 - * [M-5] Potentially incorrect voting period and delay in governor may affect governance
 - * [M-6] Low quorum threshold can lead to governance takeover
 - Low
 - * [L-1] Unused event in VaultGuardians
 - * [L-2] Unused State Variable in VaultGuardiansBase
 - * [L-3] Unused error in VaultGuardians, VaultGuardiansBase
 - * [L-4] PUSH0 Opcode
 - * [L-5] No address(0) check in VaultGuardiansBase _becomeTokenGuardian for token-Vault and token
 - * [L-6] The guardian receives its vgToken before the vault creation is confirmed introducing a risk of reentrancy

- * [L-7] A GuardianAdded event is emitted before the vault creation is confirmed leading to a false positive in the off-chain information flow
- * [L-8] Incorrect Name and Symbol when creating tokenVault for i_TokenTwo
- * [L-9] Return value not assigned to named return variable in _aaveDivest
- * [L-10] Missing check for sufficient liquidity before calling removeLiquidity()
- * [L-11] Misleading/Incorrect event emission in updateGuardianAndDaoCut
- * [L-12] Ambiguous guardian and DAO cut calculation can lead to misleading configuration
- * [L-13] Missing event emission on user deposit
- Informational
 - * [I-1] Unused interfaces IInvestableUniverseAdapter and IVaultGuardians with commented-out functions
 - * [I-2] Insufficient test coverage
 - * [I-3] No check on return value in _aaveDivest
 - * [I-4] Typo in function title in getUniswapLiquidityToken
 - * [I-5] Missing mechanism for adding new investment platforms
- Gas
 - * [G-1] Functions marked public are not used internally

About mhng

Disclaimer

We are making all efforts to find as many vulnerabilities in the code in the given time period, but we are not holding any responsibilities for the findings provided in this document. A security audit 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 the following commit hash:

```
1 0600272180b6b6103f523b0ef65b070064158303
```

Scope

```
1 ./src/
2 #-- abstract
3 |   #-- AStaticTokenData.sol
4 |   #-- AStaticUSDCData.sol
5 |   #-- AStaticWethData.sol
6 #-- dao
7 |   #-- VaultGuardianGovernor.sol
8 |   #-- VaultGuardianToken.sol
9 #-- interfaces
10 |  #-- IVaultData.sol
11 |  #-- IVaultGuardians.sol
12 |  #-- IVaultShares.sol
13 |  #-- InvestableUniverseAdapter.sol
14 #-- protocol
15 |  #-- VaultGuardians.sol
16 |  #-- VaultGuardiansBase.sol
17 |  #-- VaultShares.sol
18 |  #-- investableUniverseAdapters
19 |    #-- AaveAdapter.sol
20 |    #-- UniswapAdapter.sol
21 #-- vendor
22 |  #-- DataTypes.sol
23 |  #-- IPool.sol
24 |  #-- IUniswapV2Factory.sol
```

```
25      #-- IUniswapV2Router01.sol
```

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

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 deposited money into the vault.

You can think of a [vaultGuardian](#) as a fund manager.

To prevent a vault guardian from running off with the funds, the vault guardians are only allowed to deposit and withdraw the ERC20s into specific protocols.

- Aave v3
- Uniswap v2
- None (just hold)

These 2 protocols (plus “none” makes 3) are known as the “investable universe”.

The guardian can move funds in and out of these protocols as much as they like, but they cannot move funds to any other address.

The goal of a vault guardian, is to move funds in and out of these protocols to gain the most yield. Vault guardians charge a performance fee, the better the guardians do, the larger fee they will earn.

Anyone can become a Vault Guardian by depositing a certain amount of the ERC20 into the vault. This is called the [guardian stake](#). If a guardian wishes to stop being a guardian, they give out all user deposits and withdraw their guardian stake.

Users can then move their funds between vault managers as they see fit.

The protocol is upgradeable so that if any of the platforms in the investable universe change, or we want to add more, we can do so.

User flow

1. User deposits an ERC20 into a guardian's vault
2. The guardian automatically move the funds based on their strategy
3. The guardian can update the settings of their strategy at any time and move the funds
4. To leave the pool, a user just calls `redeem` or `withdraw`

The DAO

Guardians can earn DAO tokens by becoming guardians. The DAO is responsible for: - Updating pricing parameters - Getting a cut of all performance of all guardians

Summary

Users can stake some ERC20s to become a vault guardian. Other users can allocate them funds in order to maximize yield. The guardians can move the funds between Uniswap, Aave, or just hold the funds. The guardians are incentivized to maximize yield, as they earn a performance fee.

Roles

- User: Someone that wants to deposit assets in a vault to maximize yield
- Guardian: Someone that stakes an ERC20 amount to become a Vault Guardian by creating a new vault and are incentivized to maximize yield, by earning a fee.
- Vault Guardian DAO: Organization that controls Vault operations and earns a fee as incentive

Executive Summary

Issues found

Severity	Number of issues found
High	9
Medium	6
Low	13
Info	5

Severity	Number of issues found
Gas	1
Total	34

Findings

High

[H-1] `totalAssets()` ignores all invested funds which leads to severe disruptions of protocol functionality

Description: OpenZeppelin's ERC4626 implementation of `totalAssets()` only checks for the vault balance, not taking into consideration invested amounts, which left the vault.

```
1 function totalAssets() public view virtual returns (uint256) {  
2     return _asset.balanceOf(address(this));  
3 }
```

Vault moves 50% of user funds out into: Aave (25%) + Uniswap LP (25%)

`totalAssets()` will return just 50% of the real correct value.

Impact:

Other functions rely on `totalAssets()` return value to calculate different return values:

- `convertToShares`
- `convertToAssets`
- `previewWithdraw`
- `previewRedeem`
- `maxWithdraw`
- `maxRedeem`
- `withdraw`
- `deposit`

Returning an incorrect value will have a severe economic and functional impact on the protocol:

1. Users receive too many shares on deposit, if `totalAssets()` is too small (because invested assets are ignored), then shares become extremely cheap → users get overminted.

2. Withdrawals and redeem become corrupted because they use `convertToShares` and `convertToAssets`, which use `totalAssets()`
3. Governance votes using share balances become inaccurate

Probably more...

Proof of Concept:

- Example scenario:

1. User deposits 100 ether
2. 50 ether gets automatically invested in Aave(25 ether) + Uniswap(25 ether)
3. `totalAssets()` is called as consequence of a protocol action
4. `totalAssets()` returns only 50% of the total amount since the vault only accounts for the holding allocation(50%)
5. Calculations are erroneous, state is incorrect and functionality is disrupted.

Proof of Code:

- Add this test in `test/unit/concrete/VaultGuardiansBaseTest.t.sol`:

Code

```
1      function testTotalAssetsReturnsIncorrectAmount()
2          public
3          hasGuardian
4          hasTokenGuardian
5      {
6          // user deposits usdc
7          usdc.mint(mintAmount, user);
8          vm.startPrank(user);
9          usdc.approve(address(usdcVaultShares), mintAmount);
10         usdcVaultShares.deposit(mintAmount, user);
11         vm.stopPrank();
12
13         uint256 expectedVaultBalance = mintAmount;
14         uint256 actualVaultBalance = usdcVaultShares.totalAssets();
15         uint256 delta = (expectedVaultBalance * 6) / 10; // missing
16                     deposit amount approx. 60% -> 25% Aave + 25% Uniswap +
17                     ~10%(fees + dilution)
18
19         console.log("Expected Vault Balance: ",
20                     expectedVaultBalance);
21         console.log("Actual Vault Balance: ", actualVaultBalance);
22         console.log("Expected difference in vault amount: ", delta);
23         ;
24         console.log(
25             "Actual difference in vault amount: ",
```

```
22         expectedVaultBalance - actualVaultBalance
23     );
24
25     // Shows that there is a significant difference between
26     // expected and actual vault amount, approx. 60%
27     assertApproxEqAbs(actualVaultBalance, expectedVaultBalance,
28                       delta);
29 }
```

Recommended Mitigation:

1. Override OpenZeppelin's ERC4626 `totalAssets()` implementation to include investment amounts
2. Implement a custom function to keep track of total assets in the vault and don't rely on OpenZeppelin's solution

[H-2] Missing decimal normalization causes reverts when supplying non 18 decimal tokens (USDC) to Aave**Description:**

In `_aaveInvest()`, the vault passes `amount` directly into Aave's `supply()` call:

```
1     function _aaveInvest(IERC20 asset, uint256 amount) internal {
2         bool succ = asset.approve(address(i_aavePool), amount);
3         if (!succ) {
4             revert AaveAdapter__TransferFailed();
5         }
6         i_aavePool.supply({
7             asset: address(asset),
8             amount: amount,
9             onBehalfOf: address(this), // decides who gets Aave's
10                aTokens for the investment. In this case, mint it to the
11                vault
12                referralCode: 0
13        });
14     }
```

The vault's logic assumes 18-decimal normalized math internally, but Aave expects native token units. This means tokens like USDC (6 decimals) are handled incorrectly:

Vault user deposits: 1e6 USDC (1.0 USDC normalized to 18 decimals = 1e18)

Adapter calls `supply(asset, 1e18, ...)`

Aave interprets that as 1,000,000,000,000 USDC (1e12 USDC), which far exceeds the user's balance.

Aave's internal ERC20 `transferFrom()` then reverts due to insufficient balance or allowance.

Impact:

- The vault cannot invest or rebalance when using 6-decimal tokens such as USDC(or USDT).
- Every `_aaveInvest()` call will revert on `i_aavePool.supply()`, blocking deposits, rebalances, or withdrawals that trigger divest/reinvest.
- This blocks all vault operations for such assets resulting in DoS.

Proof of Concept:

- Scenario step by step:

1. Vault uses USDC as its asset and a user tries to deposit 1 USDC (1e6), the vault may call:

```
_aaveInvest(IERC20(USDC), 1e18);
```

2. Aave receives:

```
supply(asset=USDC, amount=1e18)
```

3. Aave executes internal call:

```
USDC.transferFrom(vault, aUSDC, 1_000_000_000_000)
```

4. Call to transferFrom reverts with:

Error: ERC20: transfer amount exceeds balance

Proof of Code for calling redeem():

- Add this code in a new file in test/ProofOfCodes:

Code

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.20;
3
4 import "forge-std/Test.sol";
5 import {AaveAdapter} from "../src/protocol/
  InvestableUniverseAdapters/AaveAdapter.sol";
6 import {IPool} from "../src/vendor/IPool.sol";
7 import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
8 import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/
  extensions/IERC20Metadata.sol";
9 import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
10
11 // Mock USDC token with 6 decimals
12 contract MockUSDC is ERC20 {
13     constructor() ERC20("Mock USDC", "USDC") {}
```

```
14
15     function decimals() public pure override returns (uint8) {
16         return 6;
17     }
18
19     function mint(address to, uint256 amount) external {
20         _mint(to, amount);
21     }
22 }
23
24 // Mock Aave pool
25 contract MockAavePool {
26     function supply(
27         address asset,
28         uint256 amount,
29         address onBehalfOf,
30         uint16
31     ) external {
32         IERC20(asset).transferFrom(onBehalfOf, address(this), amount);
33     }
34 }
35
36 // Mock vault with Aave Adapter for investment strategy
37 contract MockVault is AaveAdapter {
38     IERC20 public immutable asset;
39
40     constructor(address pool, IERC20 _asset) AaveAdapter(pool) {
41         asset = _asset;
42     }
43
44     function invest(uint256 amount) external {
45         // Simulate deposit of tokens into the vault
46         _aaveInvest(asset, amount);
47     }
48 }
49
50 // -----
51 //      Test Contract
52 // -----
53
54 contract AaveDecimalsMismatchNon18TokensPoC is Test {
55     MockUSDC private usdc;
56     MockAavePool private aavePool;
57     MockVault private vault;
58
59     error ERC20InsufficientBalance(address, uint256, uint256);
60
61     function setUp() public {
62         usdc = new MockUSDC();
63         aavePool = new MockAavePool();
64         vault = new MockVault(address(aavePool), usdc);
```

```
65
66     usdc.mint(address(vault), 1e6);
67 }
68
69 function test_USDCRevertsDueToDecimalMismatch() public {
70     // Simulate normalized 18-decimal math, vault believes amount
71     // is 1e18
72     uint256 normalAmount = 1e18;
73
74     uint256 currentBalance = usdc.balanceOf(address(vault));
75     console.log("Current Balance:", currentBalance);
76
77     uint256 requiredBalance = normalAmount;
78     console.log("Required Balance:", requiredBalance);
79
80     // Expect revert because trying to transfer 1e18 USDC
81     vm.expectRevert(
82         abi.encodeWithSelector(
83             ERC20InsufficientBalance.selector,
84             address(vault),
85             currentBalance,
86             requiredBalance
87         )
88     );
89     vault.invest(normalAmount);
90 }
91 }
```

Recommended Mitigation:

Add proper decimal scaling before interacting with Aave:

```
1  function _aaveInvest(IERC20 asset, uint256 amount) internal {
2  +     uint8 assetDecimals = IERC20Metadata(address(asset)).decimals()
3  +     ;
4  +     uint256 scaledAmount = amount;
5  +
6  +     if (assetDecimals < 18) {
7  +         scaledAmount = amount / (10 ** (18 - assetDecimals));
8  +     } else if (assetDecimals > 18) {
9  +         scaledAmount = amount * (10 ** (assetDecimals - 18));
10 +     }
11
12     bool succ = asset.approve(address(i_aavePool), amount);
13     if (!succ) {
14         revert AaveAdapter__TransferFailed();
15     }
16     i_aavePool.supply({
17 +         asset: address(asset),
18 +         amount: scaledAmount,
```

```
18         onBehalfOf: address(this),
19         referralCode: 0
20     });
21 }
```

[H-3] Missing counterparty token approvals in _uniswapDivest result in redeem DoS and locked funds

Description:

The function `_uniswapDivest()` calls `removeLiquidity()` and `swapExactTokensForTokens()` without approving the Uniswap router to transfer the required LP or counterparty tokens. Both Uniswap functions rely on `transferFrom` under the hood, which reverts if the router has no allowance. As a result, any attempt to divest liquidity or redeem user funds will revert.

Impact:

The vault cannot divest or redeem positions because `_uniswapDivest()` always reverts ==> DoS + locked funds

Proof of Concept:

Without the approvals for LP and counterparty tokens, calling `redeem()` or `rebalanceFunds()`, which both call `_uniswapDivest` will result in a revert with `ERC20InsufficientAllowance`.

Proof of Code for calling `redeem()`:

- Add this code in a new file in `test/ProofOfCodes`:

Code

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.20;
3
4 import "forge-std/Test.sol";
5 import {VaultShares} from "../src/protocol/VaultShares.sol";
6 import {IVaultShares, IVaultData} from "../src/interfaces/
7     IVaultShares.sol";
8 import {ERC20Mock} from "../mocks/ERC20Mock.sol";
9 import {DataTypes} from "../src/vendor/DataTypes.sol";
10
11 // -----
12 //         Uniswap mocks
13 // -----
14
15 contract UniswapLP is ERC20Mock {
16     // Inherit ERC20Mock: free mint/burn helpers already available
17 }
```

```
17
18 contract UniswapFactory {
19     address public immutable pair;
20
21     constructor(address pair_) {
22         pair = pair_;
23     }
24
25     function getPair(address, address) external view returns (address)
26     {
27         return pair;
28     }
29 }
30 // This router is Uniswap-V2-shaped enough for VaultShares.
31 // UniswapAdapter
32 contract UniswapRouter {
33     address private _factory;
34     UniswapLP private _lp;
35
36     constructor(address factory_, address lp_) {
37         _factory = factory_;
38         _lp = UniswapLP(lp_);
39     }
40
41     function factory() external view returns (address) {
42         return _factory;
43     }
44
45     function swapExactTokensForTokens(
46         uint256 amountIn,
47         uint256 /* amountOutMin */,
48         address[] calldata path,
49         address to,
50         uint256 /* deadline */
51     ) external returns (uint256[] memory amounts) {
52         // pull input
53         ERC20Mock(path[0]).transferFrom(msg.sender, address(this),
54             amountIn);
55
56         // use mint to avoid needing pre-funding
57         ERC20Mock(path[1]).mint(amountIn, to);
58
59         amounts = new uint256[](path.length);
60         amounts[0] = amountIn;
61         amounts[1] = amountIn;
62     }
63
64     function addLiquidity(
65         address tokenA,
66         address tokenB,
```

```
65         uint256 amountADesired,
66         uint256 amountBDesired,
67         uint256 /* amountAMin */,
68         uint256 /* amountBMin */,
69         address to,
70         uint256 /* deadline */
71     ) external returns (uint256 amountA, uint256 amountB, uint256
liquidity) {
72         // pull all desired tokens from caller
73         ERC20Mock(tokenA).transferFrom(
74             msg.sender,
75             address(this),
76             amountADesired
77         );
78         ERC20Mock(tokenB).transferFrom(
79             msg.sender,
80             address(this),
81             amountBDesired
82         );
83
84         _lp.mint(100 ether, to);
85
86         return (amountADesired, amountBDesired, 100 ether);
87     }
88
89     function removeLiquidity(
90         address tokenA,
91         address tokenB,
92         uint256 liquidity,
93         uint256 /* amountAMin */,
94         uint256 /* amountBMin */,
95         address to,
96         uint256 /* deadline */
97     ) external returns (uint256 amountA, uint256 amountB) {
98         // take the LP from caller
99         _lp.transferFrom(msg.sender, address(this), liquidity);
100        // burn
101        _lp.burn(liquidity, address(this));
102
103        // return *1 wei* of each token to the vault
104        ERC20Mock(tokenA).mint(1, to);
105        ERC20Mock(tokenB).mint(1, to);
106
107        return (1, 1);
108    }
109 }
110
111 // -----
112 //           Aave mocks
113 // -----
114
```



```
115 // Deploy a small MockAavePool that returns (aTokenAddress, 0,0,...)
116 contract MockAavePool {
117     address public immutable aTokenAddr;
118
119     constructor(address _aTokenAddr) {
120         aTokenAddr = _aTokenAddr;
121     }
122
123     // Must match the interface signature used by VaultShares
124     function getReserveData(
125         address
126     ) external view returns (DataTypes.ReserveData memory r) {
127         // leave everything at the zero default, just set aTokenAddress
128         r.aTokenAddress = aTokenAddr;
129     }
130
131     function supply(
132         address /* asset */,
133         uint256 amount,
134         address onBehalfOf,
135         uint16 /* referralCode */
136     ) external {
137         // For the PoC we don't simulate any lending, just acknowledge
138         // the call.
139         if (amount > 0) {
140             ERC20Mock(aTokenAddr).mint(amount, onBehalfOf);
141         }
142     }
143
144     // -----
145     //      Test Contract
146     // -----
147
148     contract MissingApprovalsUniswapDivestPoC_Test is Test {
149         VaultShares private vault;
150         ERC20Mock private asset;
151         ERC20Mock private tokenOne;
152         ERC20Mock private mockWETH; // distinct mock WETH
153         ERC20Mock private awethTokenMock;
154
155         UniswapRouter router;
156         UniswapLP lp;
157
158         error ERC20InsufficientAllowance(address, uint256, uint256);
159
160         function setUp() public {
161             asset = new ERC20Mock();
162             tokenOne = new ERC20Mock();
163             mockWETH = new ERC20Mock(); // distinct mock WETH
164             awethTokenMock = new ERC20Mock();
```

```
165
166     MockAavePool pool = new MockAavePool(address(awethTokenMock));
167
168     // LP + Factory + Router
169     lp = new UniswapLP();
170     UniswapFactory factory = new UniswapFactory(address(lp));
171     router = new UniswapRouter(address(factory), address(lp));
172
173     // Build constructor data
174     IVaultShares.ConstructorData memory c = IVaultShares.
        ConstructorData({
175         asset: asset,
176         vaultName: "PoC Vault",
177         vaultSymbol: "POC",
178         guardian: address(this),
179         allocationData: IVaultData.AllocationData({
180             holdAllocation: 0,
181             uniswapAllocation: 1000,
182             aaveAllocation: 0
183         }),
184         aavePool: address(pool),
185         uniswapRouter: address(router),
186         guardianAndDaoCut: 100,
187         vaultGuardians: address(this),
188         weth: address(mockWETH),
189         usdc: address(tokenOne)
190     });
191
192     vault = new VaultShares(c);
193
194     // Label for nicer traces
195     vm.label(address(vault), "VaultShares(POC)");
196     vm.label(address(asset), "ASSET");
197     vm.label(address(tokenOne), "TOKEN_ONE");
198     vm.label(address(factory), "Factory");
199     vm.label(address(router), "Router");
200     vm.label(address(lp), "LP");
201 }
202
203 function test_MissingApprovalsUniswapDivestResultsInRevert() public
204 {
205     // User mints and deposits 100 ASSET
206     address user = address(0xBEEF);
207     uint256 depositAmt = 100 ether;
208
209     asset.mint(depositAmt, user);
210
211     // pre-fund the vault so the double-accounting won't cause
212     // insufficient balance
213     // this must happen after vault creation (setUp) but before
214     // deposit triggers invest()
```

```
212     asset.mint(depositAmt / 2, address(vault));
213
214     // Approve & deposit
215     vm.startPrank(user);
216     asset.approve(address(vault), depositAmt);
217     vault.deposit(depositAmt, user);
218     vm.stopPrank();
219
220     uint256 currentAllowance = lp.allowance(
221         address(vault),
222         address(router)
223     );
224     uint256 requiredAllowance = lp.balanceOf(address(vault));
225
226     console.log("Current allowance:", currentAllowance);
227     console.log("Required allowance:", requiredAllowance);
228
229     // If we call redeem(), expectRevert won't be able to pick up
230     // the revert, because it happens during a nested call
231     // expectRevert checks the top level call for revert
232     // removeLiquidity() and swapExactTokensForTokens() are nested
233     // calls inside redeem() which is the top call, that's why
234     // removeLiquidity() is simulated separately here
235
236     // Next removeLiquidity() attempt should revert because of
237     // insufficient allowance
238     vm.expectRevert(
239         abi.encodeWithSelector(
240             ERC20InsufficientAllowance.selector,
241             address(router),
242             currentAllowance,
243             requiredAllowance
244         )
245     );
246
247     vm.startPrank(address(vault));
248     router.removeLiquidity(
249         address(asset),
250         address(mockWETH),
251         depositAmt,
252         0,
253         0,
254         user,
255         1
256     );
257     vm.stopPrank();
258 }
```

Recommended Mitigation:

Add safe increase allowance for LP and Counterparty Token

```
1      function _uniswapDivest(IERC20 token, uint256 liquidityAmount)
2          internal returns (uint256 amountOfAssetReturned) {
3
4          // Approve uniswapRouter to transfer LP from VaultShares
5      +      if (
6      +          s_uniswapLP.allowance(address(this), address(
7      +          i_uniswapRouter)) == 0
8      +      ) {
9      +          s_uniswapLP.safeIncreaseAllowance(
10      +              address(i_uniswapRouter),
11      +              type(uint256).max
12      +          );
13      +      };
14
15      (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
16      i_uniswapRouter.removeLiquidity({
17      tokenA: address(token),
18      tokenB: address(counterPartyToken),
19      liquidity: liquidityAmount,
20      amountAMin: 0,
21      amountBMin: 0,
22      to: address(this),
23      deadline: block.timestamp
24      });
25      s_pathArray = [address(counterPartyToken), address(token)];
26
27      // Approve uniswapRouter to transfer counterPartyToken from
28      VaultShares
29      +      if (
30      +          counterPartyToken.allowance(address(this), address(
31      +          i_uniswapRouter)) == 0
32      +      ) {
33      +          counterPartyToken.safeIncreaseAllowance(
34      +              address(i_uniswapRouter),
35      +              type(uint256).max
36      +          );
37      +      };
38
39      uint256[] memory amounts = i_uniswapRouter.
40      swapExactTokensForTokens({
41      amountIn: counterPartyTokenAmount,
42      amountOutMin: 0,
43      path: s_pathArray,
44      to: address(this),
45      deadline: block.timestamp
46      });
```

```
43         emit UniswapDivested(tokenAmount, amounts[1]);
44         amountOfAssetReturned = amounts[1];
45     }
```

[H-4] Guardians have no restrictions on minting vgToken and can accumulate high amounts to take over DAO, steal DAO fees and set parameters for own interest

Description: Becoming a guardian is repaid by receiving Vault Guardian Tokens (vgTokens). Whenever a guardian successfully calls `VaultGuardiansBase::becomeGuardian` or `VaultGuardiansBase::becomeTokenGuardian`, `_becomeTokenGuardian` is executed, which mints the caller `i_vgToken`.

```
1     function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
2         private returns (address) {
3         s_guardians[msg.sender][token] = IVaultShares(address(
4             tokenVault));
5         i_vgToken.mint(msg.sender, s_guardianStakePrice);
6         emit GuardianAdded(msg.sender, token);
7         token.safeTransferFrom(msg.sender, address(this),
8             s_guardianStakePrice);
9         token.approve(address(tokenVault), s_guardianStakePrice);
10        tokenVault.deposit(s_guardianStakePrice, msg.sender);
11        return address(tokenVault);
12    }
```

Guardians are also free to quit their role at any time, calling the `VaultGuardianBase::quitGuardian` function. The combination of minting vgTokens, and freely being able to quit, results in users being able to farm vgTokens at any time.

Impact: Assuming the token has no monetary value, the malicious guardian could accumulate tokens until they can overtake the DAO. Then, they could execute any of these functions of the `VaultGuardians` contract:

```
1     "sweepErc20s(address)": "942d0ff9",
2     "transferOwnership(address)": "f2fde38b",
3     "updateGuardianAndDaoCut(uint256)": "9e8f72a4",
4     "updateGuardianStakePrice(uint256)": "d16fe105",
```

Proof of Concept:

1. User becomes WETH guardian and is minted vgTokens.
2. User quits, is given back original WETH allocation.
3. User becomes WETH guardian with the same initial allocation.
4. Repeat to keep minting vgTokens indefinitely.

Code

Place the following code into `VaultGuardiansBaseTest.t.sol`

```
1      function testDaoTakeover() public hasGuardian hasTokenGuardian {
2          address maliciousGuardian = makeAddr("maliciousGuardian");
3          uint256 startingVoterUsdcBalance = usdc.balanceOf(
4              maliciousGuardian);
5          uint256 startingVoterWethBalance = weth.balanceOf(
6              maliciousGuardian);
7          assertEq(startingVoterUsdcBalance, 0);
8          assertEq(startingVoterWethBalance, 0);
9
10         VaultGuardianGovernor governor = VaultGuardianGovernor(payable(
11             vaultGuardians.owner()));
12         VaultGuardianToken vgToken = VaultGuardianToken(address(
13             governor.token()));
14
15         // Flash loan the tokens, or just buy a bunch for 1 block
16         weth.mint(mintAmount, maliciousGuardian); // The same amount as
17             the other guardians
18         uint256 startingMaliciousVGTokenBalance = vgToken.balanceOf(
19             maliciousGuardian);
20         uint256 startingRegularVGTokenBalance = vgToken.balanceOf(
21             guardian);
22         console.log("Malicious vgToken Balance:\t",
23             startingMaliciousVGTokenBalance);
24         console.log("Regular vgToken Balance:\t",
25             startingRegularVGTokenBalance);
26
27         // Malicious Guardian farms tokens
28         vm.startPrank(maliciousGuardian);
29         weth.approve(address(vaultGuardians), type(uint256).max);
30         for (uint256 i; i < 10; i++) {
31             address maliciousWethSharesVault = vaultGuardians.
32                 becomeGuardian(allocationData);
33             IERC20(maliciousWethSharesVault).approve(
34                 address(vaultGuardians),
35                 IERC20(maliciousWethSharesVault).balanceOf(
36                     maliciousGuardian)
37             );
38             vaultGuardians.quitGuardian();
39         }
40         vm.stopPrank();
41
42         uint256 endingMaliciousVGTokenBalance = vgToken.balanceOf(
43             maliciousGuardian);
44         uint256 endingRegularVGTokenBalance = vgToken.balanceOf(
45             guardian);
46         console.log("Malicious vgToken Balance:\t",
47             endingMaliciousVGTokenBalance);
```

```
34         console.log("Regular vgToken Balance:\t",
35             endingRegularVGTokenBalance);
36     }
```

Recommended Mitigation:

1. Mint vgTokens on a vesting schedule after a user becomes a guardian.
2. Burn vgTokens when a guardian quits.

[H-5] Double accounting of swap amount leads to excess token spending and fund loss or DoS**Description:**

In `UniswapAdapter::_uniswapInvest`, the function uses `amountADesired = amountOfTokenToSwap + amounts[0]` when adding liquidity. However, `amounts[0]` already equals `amountOfTokenToSwap`, which was spent during the prior swap. This results in double accounting for the same amount of the base token.

```
1      (
2          uint256 tokenAmount,
3          uint256 counterPartyTokenAmount,
4          uint256 liquidity
5      ) = i_uniswapRouter.addLiquidity({
6          tokenA: address(token),
7          tokenB: address(counterPartyToken),
8          // Double accounting for amountOfTokenToSwap,
9              amountOfTokenToSwap + amounts[0]
10         amountADesired: amountOfTokenToSwap + amounts[0],
11         amountBDesired: amounts[1],
12         amountAMin: 0,
13         amountBMin: 0,
14         to: address(this),
15         deadline: block.timestamp
16     });
```

Impact:

The vault attempts to transfer 1.5× the intended investment amount.

If balance is insufficient → the call reverts, halting deposits or investments.

If balance exists → the vault overspends user funds into the liquidity pool, causing a permanent loss of assets and inaccurate share accounting.

Proof of Concept:

Case 1. Investing 100 tokens, the vault spends 150 — proving double counting and extra spending/loss of funds.

Case 2. If the vault lacks the extra 50 tokens, the same path reverts due to insufficient balance, resulting in denial of service.

Proof of Code for Case 1:

Add this code in a new file in test/ProofOfCodes:

Code

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.20;
3
4 import "forge-std/Test.sol";
5 import {VaultShares} from "../../src/protocol/VaultShares.sol";
6 import {IVaultShares, IVaultData} from "../../src/interfaces/
  IVaultShares.sol";
7 import {ERC20Mock} from "../../mocks/ERC20Mock.sol";
8 import {DataTypes} from "../../src/vendor/DataTypes.sol";
9
10 // -----
11 //           Uniswap mocks
12 // -----
13
14 contract MaliciousLP is ERC20Mock {
15     // Inherit ERC20Mock: free mint/burn helpers already available
16 }
17
18 contract MaliciousFactory {
19     address public immutable pair;
20
21     constructor(address pair_) {
22         pair = pair_;
23     }
24
25     function getPair(address, address) external view returns (address)
26     {
27         return pair;
28     }
29 }
30
31 contract MaliciousRouter {
32     // This router is Uniswap-V2-shaped enough for VaultShares.
33     // Behavior:
34     // - swapExactTokensForTokens: takes all input, returns *1 wei* as
35     //   output
36     // - addLiquidity: pulls all desired tokens, mints *1 wei* LP
37     // - removeLiquidity: burns all LP, returns *1 wei* of each token
38     // All succeed because min amounts are zero.
39
40     address private _factory;
```



```
39     MaliciousLP private _lp;
40
41     constructor(address factory_, address lp_) {
42         _factory = factory_;
43         _lp = MaliciousLP(lp_);
44     }
45
46     function factory() external view returns (address) {
47         return _factory;
48     }
49
50     function swapExactTokensForTokens(
51         uint256 amountIn,
52         uint256 /* amountOutMin */, // zero in vulnerable code
53         address[] calldata path,
54         address to,
55         uint256 /* deadline */
56     ) external returns (uint256[] memory amounts) {
57         // pull input
58         ERC20Mock(path[0]).transferFrom(msg.sender, address(this),
59             amountIn);
60
61         // mint or transfer *1 wei* of output to `to`
62         // use mint to avoid needing pre-funding
63         ERC20Mock(path[1]).mint(1, to);
64
65         amounts = new uint256[](path.length);
66         amounts[0] = amountIn; // uniswap-style return
67         amounts[1] = 1;
68     }
69
70     function addLiquidity(
71         address tokenA,
72         address tokenB,
73         uint256 amountADesired,
74         uint256 amountBDesired,
75         uint256 amountAMin, // zero in vulnerable code
76         uint256 amountBMin, // zero in vulnerable code
77         address to,
78         uint256 /* deadline */
79     ) external returns (uint256 amountA, uint256 amountB, uint256
80         liquidity) {
81         // pull all desired tokens from caller
82         ERC20Mock(tokenA).transferFrom(
83             msg.sender,
84             address(this),
85             amountADesired
86         );
87         ERC20Mock(tokenB).transferFrom(
88             msg.sender,
89             address(this),
```

```
88         amountBDesired
89     );
90
91     // min checks are zero -> always satisfied
92     require(amountAMin == 0 && amountBMin == 0, "not the vulnerable
93         path");
94
95     // mint only 1 wei of LP to the vault (terrible deal)
96     _lp.mint(1, to);
97
98     return (amountADesired, amountBDesired, 1);
99 }
100
101 function removeLiquidity(
102     address tokenA,
103     address tokenB,
104     uint256 liquidity,
105     uint256 amountAMin, // zero in vulnerable code
106     uint256 amountBMin, // zero in vulnerable code
107     address to,
108     uint256 /* deadline */
109 ) external returns (uint256 amountA, uint256 amountB) {
110     // take the LP from caller
111     _lp.transferFrom(msg.sender, address(this), liquidity);
112     // burn
113     _lp.burn(liquidity, address(this));
114
115     // return *1 wei* of each token to the vault
116     ERC20Mock(tokenA).mint(1, to);
117     ERC20Mock(tokenB).mint(1, to);
118
119     require(amountAMin == 0 && amountBMin == 0, "not the vulnerable
120         path");
121
122     return (1, 1);
123 }
124
125 // -----
126 //           Aave mocks
127 // -----
128 // Deploy a small MockAavePool that returns (aTokenAddress, 0,0,...)
129 contract MockAavePool {
130     address public immutable aTokenAddr;
131
132     constructor(address _aTokenAddr) {
133         aTokenAddr = _aTokenAddr;
134     }
135
136     // Must match the interface signature used by VaultShares
```

```
137     function getReserveData(  
138         address  
139     ) external view returns (DataTypes.ReserveData memory r) {  
140         // leave everything at the zero default, just set aTokenAddress  
141         r.aTokenAddress = aTokenAddr;  
142     }  
143  
144     function supply(  
145         address /* asset */,  
146         uint256 amount,  
147         address onBehalfOf,  
148         uint16 /* referralCode */  
149     ) external {  
150         if (amount > 0) {  
151             ERC20Mock(aTokenAddr).mint(amount, onBehalfOf);  
152         }  
153     }  
154 }  
155  
156 // -----  
157 //             PoC test  
158 // -----  
159  
160 contract SlippageZeroMin_PoC_Test is Test {  
161     VaultShares private vault;  
162     ERC20Mock private asset;  
163     ERC20Mock private tokenOne;  
164     ERC20Mock private mockWETH; // distinct mock WETH  
165     ERC20Mock private awethTokenMock;  
166  
167     MaliciousLP lp;  
168     MaliciousFactory factory;  
169     MaliciousRouter router;  
170  
171     function setUp() public {  
172         asset = new ERC20Mock();  
173         tokenOne = new ERC20Mock();  
174         mockWETH = new ERC20Mock(); // distinct mock WETH  
175         awethTokenMock = new ERC20Mock();  
176  
177         MockAavePool pool = new MockAavePool(address(awethTokenMock));  
178  
179         // LP + Factory + Router (malicious)  
180         lp = new MaliciousLP();  
181         factory = new MaliciousFactory(address(lp));  
182         router = new MaliciousRouter(address(factory), address(lp));  
183  
184         // Build constructor data with:  
185         IVaultShares.ConstructorData memory c = IVaultShares.  
186             ConstructorData({  
                 asset: asset,
```

```
187         vaultName: "PoC Vault",
188         vaultSymbol: "POC",
189         guardian: address(this),
190         allocationData: IVaultData.AllocationData({
191             holdAllocation: 0,
192             uniswapAllocation: 1000, // 100% to Uniswap path (will
                hit vulnerable code)
193             aaveAllocation: 0
194         }),
195         aavePool: address(pool),
196         uniswapRouter: address(router),
197         guardianAndDaoCut: 100,
198         vaultGuardians: address(this),
199         weth: address(mockWETH),
200         usdc: address(tokenOne)
201     });
202
203     vault = new VaultShares(c);
204
205     // Label for nicer traces
206     vm.label(address(vault), "VaultShares(POC)");
207     vm.label(address(asset), "ASSET");
208     vm.label(address(tokenOne), "TOKEN_ONE");
209     vm.label(address(factory), "MaliciousFactory");
210     vm.label(address(router), "MaliciousRouter");
211     vm.label(address(lp), "MaliciousLP");
212 }
213
214 function test_Slippage_WithZeroMins_LosesValue() public {
215     // User mints and deposits 100 ASSET
216     address user = address(0xBEEF);
217     uint256 depositAmt = 100 ether;
218
219     asset.mint(depositAmt, user);
220
221     // pre-fund the vault so the double-accounting won't cause
        insufficient balance
222     // this must happen after vault creation (setUp) but before
        deposit triggers invest()
223     asset.mint(depositAmt / 2, address(vault));
224
225     // Approve & deposit
226     vm.startPrank(user);
227     asset.approve(address(vault), depositAmt);
228     uint256 shares = vault.deposit(depositAmt, user);
229     vm.stopPrank();
230
231     // Adding the uniswap approvals that are missing in the
        original code
232     vm.startPrank(address(vault));
233     lp.approve(address(router), type(uint256).max);
```

```
234     tokenOne.approve(address(router), type(uint256).max);
235     mockWETH.approve(address(router), type(uint256).max);
236     vm.stopPrank();
237
238     // Full redeem immediately - vulnerable paths:
239     // - invest: swap with amountOutMin=0, addLiquidity with
240       amountAMin/BMin=0
241     // - redeem: removeLiquidity with amountAMin/BMin=0, swap with
242       amountOutMin=0
243     vm.startPrank(user);
244     uint256 assetsOut = vault.redeem(vault.balanceOf(user), user,
245       user);
246     vm.stopPrank();
247
248     // Show the damage in logs
249     console.log("Deposit:", depositAmt);
250     console.log("Shares minted:", shares);
251     console.log("Assets out on full redeem:", assetsOut);
252
253     // Assert a large loss -> 10x
254     assertLt(
255       assetsOut,
256       depositAmt / 10,
257       "Expected significant loss due to zero slippage protections
258       "
259     );
260   }
261 }
```

Recommended Mitigation:

Use only the remaining base tokens for amountADesired:

```
1  +     uint256 tokenLeft = amount - amounts[0];
2
3      (
4         uint256 tokenAmount,
5         uint256 counterPartyTokenAmount,
6         uint256 liquidity
7      ) = i_uniswapRouter.addLiquidity({
8         tokenA: address(token),
9         tokenB: address(counterPartyToken),
10        // Use only the remaining base tokens after swap for
11          amountADesired
12      +     amountADesired: uint256 tokenLeft,
13        amountBDesired: amounts[1],
14        amountAMin: 0,
15        amountBMin: 0,
16        to: address(this),
17        deadline: block.timestamp
18      });
```

[H-6] No slippage protection in uniswap operations in UniswapAdapter can lead to loss of funds due to price changes**Description:**

The contract performs Uniswap swaps and liquidity operations (`swapExactTokensForTokens`, `addLiquidity`, and `removeLiquidity`) without specifying minimum acceptable output or input values. This lack of slippage control allows a transaction to execute at any unfavorable price, exposing the protocol to MEV, frontrunning, and large value loss due to market movement.

- Case 1: 0 value for `amountOutMin` in swaps

Found in:

`UniswapAdapter::_uniswapInvest -> swapExactTokensForTokens()` `UniswapAdapter::_uniswapDivest -> swapExactTokensForTokens()`

```
1 uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
2     amountIn: amountOfTokenToSwap,
3     amountOutMin: 0, // no protection
4     ...
5 });
```

Setting these to 0 means the transaction will execute even if the received output is 99% below expected. A small market move, a MEV sandwich, or a manipulated pair could drain value from the vault.

For example, when calling `swapExactTokensForTokens()` with `amountOutMin = 0`, there is no lower limit on the amount of returned tokens, accepting whatever value the market gives back.

Uniswap's price depends on pool reserves, so if the pool moves (price shifts between transaction submission and execution), or a MEV bot front-runs the transaction and manipulates the pool price, a large percentage of tokens can be lost.

- Case 2: 0 value for `amountAMin` / `amountBMin` in liquidity operations

Found in:

`UniswapAdapter::_uniswapInvest -> addLiquidity()` `UniswapAdapter::_uniswapDivest -> removeLiquidity()`

```
1 (, , uint256 liquidity) = i_uniswapRouter.addLiquidity({
2     ...
3     amountAMin: 0, // no protection
4     amountBMin: 0, // no protection
5     ...
6 });
7
8 (, ) = i_uniswapRouter.removeLiquidity({
```

```
9      ...
10     amountAMin: 0, // no protection
11     amountBMin: 0, // no protection
12     ...
13  });
```

When the contract supplies or removes liquidity from a Uniswap pool with amountAMin and amountBMin set to 0, it effectively disables any price-slippage protection during these operations.

Uniswap uses these parameters to guarantee that liquidity is only added or removed within an acceptable price range.

If both are zero, the function call will succeed regardless of how far the pool's price has moved since the transaction was initiated.

Impact: Loss of vault funds -> An attacker or validator can manipulate price before execution, leading to vault assets being swapped at disadvantageous rates, draining value from user deposits.

Proof of Concept:

Exploit scenario: Swap with amountOutMin = 0 & Add liquidity with amountAMin / amountBMin = 0

Goal: Manipulate price so the vault swaps into an unfavorable ratio before addLiquidity, causing the vault to deposit assets at a bad price (lose value / mint fewer LP).

1. A user deposits weth into the vault which triggers _investFunds. Vault will attempt to swap half the deposited token for the counterparty and then call addLiquidity with amountAMin = 0 and amountBMin = 0.
2. Attacker front-run / manipulate price: Immediately before the vault's swap or addLiquidity will be included in a block, the attacker executes a large swap (or flash-loan + swap) on the same pair to move the pool price drastically (e.g., swap many USDC for WETH)
3. Vault executes swap with amountOutMin = 0, because amountOutMin is zero, the router will not revert — the vault accepts the unfavorable output amount.
4. Vault calls addLiquidity(amountAMin=0, amountBMin=0), with the skewed balances, the vault adds liquidity at the manipulated ratio and receives fewer LP tokens than expected or pairs tokens in a bad ratio — effectively losing value.
5. Vault deposits at manipulated price and loses value. No revert occurs because amountOutMin and amountAMin/amountBMin are zero.

- Proof of Code:

Add this code in a new file in test/ProofOfCodes:

Code

```
1 // SPDX-License-Identifier: MIT
```

```
2 pragma solidity ^0.8.20;
3
4 import "forge-std/Test.sol";
5 import {VaultShares} from "../src/protocol/VaultShares.sol";
6 import {IVaultShares, IVaultData} from "../src/interfaces/
  IVaultShares.sol";
7 import {ERC20Mock} from "../mocks/ERC20Mock.sol";
8 import {DataTypes} from "../src/vendor/DataTypes.sol";
9
10 // -----
11 //         Uniswap mocks
12 // -----
13
14 contract MaliciousLP is ERC20Mock {
15     // Inherit ERC20Mock: free mint/burn helpers already available
16 }
17
18 contract MaliciousFactory {
19     address public immutable pair;
20
21     constructor(address pair_) {
22         pair = pair_;
23     }
24
25     function getPair(address, address) external view returns (address)
26     {
27         return pair;
28     }
29 }
30
31 contract MaliciousRouter {
32     // This router is Uniswap-V2-shaped enough for VaultShares.
33     // Behavior:
34     // - swapExactTokensForTokens: takes all input, returns *1 wei* as
35     //   output
36     // - addLiquidity: pulls all desired tokens, mints *1 wei* LP
37     // - removeLiquidity: burns all LP, returns *1 wei* of each token
38     // All succeed because min amounts are zero.
39
40     address private _factory;
41     MaliciousLP private _lp;
42
43     constructor(address factory_, address lp_) {
44         _factory = factory_;
45         _lp = MaliciousLP(lp_);
46     }
47
48     function factory() external view returns (address) {
49         return _factory;
50     }
51 }
```



```
49
50     function swapExactTokensForTokens(
51         uint256 amountIn,
52         uint256 /* amountOutMin */, // zero in vulnerable code
53         address[] calldata path,
54         address to,
55         uint256 /* deadline */
56     ) external returns (uint256[] memory amounts) {
57         // pull input
58         ERC20Mock(path[0]).transferFrom(msg.sender, address(this),
59             amountIn);
60
61         // mint or transfer *1 wei* of output to `to`
62         // use mint to avoid needing pre-funding
63         ERC20Mock(path[1]).mint(1, to);
64
65         amounts = new uint256[](path.length);
66         amounts[0] = amountIn; // uniswap-style return
67         amounts[1] = 1;
68     }
69
70     function addLiquidity(
71         address tokenA,
72         address tokenB,
73         uint256 amountADesired,
74         uint256 amountBDesired,
75         uint256 amountAMin, // zero in vulnerable code
76         uint256 amountBMin, // zero in vulnerable code
77         address to,
78         uint256 /* deadline */
79     ) external returns (uint256 amountA, uint256 amountB, uint256
80         liquidity) {
81         // pull all desired tokens from caller
82         ERC20Mock(tokenA).transferFrom(
83             msg.sender,
84             address(this),
85             amountADesired
86         );
87         ERC20Mock(tokenB).transferFrom(
88             msg.sender,
89             address(this),
90             amountBDesired
91         );
92
93         // min checks are zero -> always satisfied
94         require(amountAMin == 0 && amountBMin == 0, "not the vulnerable
95             path");
96
97         // mint only 1 wei of LP to the vault (terrible deal)
98         _lp.mint(1, to);
```

```

97         return (amountADesired, amountBDesired, 1);
98     }
99
100     function removeLiquidity(
101         address tokenA,
102         address tokenB,
103         uint256 liquidity,
104         uint256 amountAMin, // zero in vulnerable code
105         uint256 amountBMin, // zero in vulnerable code
106         address to,
107         uint256 /* deadline */
108     ) external returns (uint256 amountA, uint256 amountB) {
109         // take the LP from caller
110         _lp.transferFrom(msg.sender, address(this), liquidity);
111         // burn
112         _lp.burn(liquidity, address(this));
113
114         // return *1 wei* of each token to the vault
115         ERC20Mock(tokenA).mint(1, to);
116         ERC20Mock(tokenB).mint(1, to);
117
118         require(amountAMin == 0 && amountBMin == 0, "not the vulnerable
119             path");
120
121         return (1, 1);
122     }
123 }
124 // -----
125 //           Aave mocks
126 // -----
127
128 // Deploy a small MockAavePool that returns (aTokenAddress, 0,0,...)
129 contract MockAavePool {
130     address public immutable aTokenAddr;
131
132     constructor(address _aTokenAddr) {
133         aTokenAddr = _aTokenAddr;
134     }
135
136     // Must match the interface signature used by VaultShares
137     function getReserveData(
138         address
139     ) external view returns (DataTypes.ReserveData memory r) {
140         // leave everything at the zero default, just set aTokenAddress
141         r.aTokenAddress = aTokenAddr;
142     }
143
144     function supply(
145         address /* asset */,
146         uint256 amount,
```

```
147     address onBehalfOf,
148     uint16 /* referralCode */
149 ) external {
150     if (amount > 0) {
151         ERC20Mock(aTokenAddr).mint(amount, onBehalfOf);
152     }
153 }
154 }
155
156 // -----
157 //             PoC test
158 // -----
159
160 contract SlippageZeroMin_PoC_Test is Test {
161     VaultShares private vault;
162     ERC20Mock private asset;
163     ERC20Mock private tokenOne;
164     ERC20Mock private mockWETH; // distinct mock WETH
165     ERC20Mock private awethTokenMock;
166
167     function setUp() public {
168         asset = new ERC20Mock();
169         tokenOne = new ERC20Mock();
170         mockWETH = new ERC20Mock(); // distinct mock WETH
171         awethTokenMock = new ERC20Mock();
172
173         MockAavePool pool = new MockAavePool(address(awethTokenMock));
174
175         // LP + Factory + Router (malicious)
176         MaliciousLP lp = new MaliciousLP();
177         MaliciousFactory factory = new MaliciousFactory(address(lp));
178         MaliciousRouter router = new MaliciousRouter(
179             address(factory),
180             address(lp)
181         );
182
183         // Build constructor data with:
184         IVaultShares.ConstructorData memory c = IVaultShares.
185             ConstructorData({
186                 asset: asset,
187                 vaultName: "PoC Vault",
188                 vaultSymbol: "POC",
189                 guardian: address(this),
190                 allocationData: IVaultData.AllocationData({
191                     holdAllocation: 0,
192                     uniswapAllocation: 1000, // 100% to Uniswap path (will
193                         hit vulnerable code)
194                     aaveAllocation: 0
195                 }),
196                 aavePool: address(pool),
197                 uniswapRouter: address(router),
```

```
196         guardianAndDaoCut: 100,
197         vaultGuardians: address(this),
198         weth: address(mockWETH),
199         usdc: address(tokenOne)
200     });
201
202     vault = new VaultShares(c);
203
204     // Label for nicer traces
205     vm.label(address(vault), "VaultShares(POC)");
206     vm.label(address(asset), "ASSET");
207     vm.label(address(tokenOne), "TOKEN_ONE");
208     vm.label(address(factory), "MaliciousFactory");
209     vm.label(address(router), "MaliciousRouter");
210     vm.label(address(lp), "MaliciousLP");
211 }
212
213 function test_Slippage_WithZeroMins_LosesValue() public {
214     // User mints and deposits 100 ASSET
215     address user = address(0xBEEF);
216     uint256 depositAmt = 100 ether;
217
218     asset.mint(depositAmt, user);
219
220     // pre-fund the vault so the double-accounting won't cause
221     // insufficient balance
222     // this must happen after vault creation (setUp) but before
223     // deposit triggers invest()
224     asset.mint(depositAmt / 2, address(vault));
225
226     // Approve & deposit
227     vm.startPrank(user);
228     asset.approve(address(vault), depositAmt);
229     uint256 shares = vault.deposit(depositAmt, user);
230     vm.stopPrank();
231
232     // Full redeem immediately - vulnerable paths:
233     // - invest: swap with amountOutMin=0, addLiquidity with
234     //   amountAMin/BMin=0
235     // - redeem: removeLiquidity with amountAMin/BMin=0, swap with
236     //   amountOutMin=0
237     vm.startPrank(user);
238     uint256 assetsOut = vault.redeem(vault.balanceOf(user), user,
239     user);
240     vm.stopPrank();
241
242     // Show the damage in logs
243     console.log("Deposit:", depositAmt);
244     console.log("Shares minted:", shares);
245     console.log("Assets out on full redeem:", assetsOut);
246 }
```

```
242         // Assert a large loss -> 10x
243         assertLt(
244             assetsOut,
245             depositAmt / 10,
246             "Expected significant loss due to zero slippage protections
                "
247         );
248     }
249 }
```

Recommended Mitigation:

Set a real slippage tolerance -> Allow the user (or the vault logic) to specify a percentage tolerance for price movement.

- Case 1:

Compute `amountOutMin` dynamically before swapping.

Add a new function to dynamically calculate `amountOutMin` based on user entered slippage tolerance `slippageBps`:

```
1 + function _getAmountOutMin(
2 +     IUniswapV2Router01 router,
3 +     address[] memory path,
4 +     uint256 amountIn,
5 +     uint256 slippageBps // e.g., 100 = 1%
6 + ) internal view returns (uint256) {
7 +     uint256[] memory amountsOut = router.getAmountsOut(amountIn,
8 +     path);
9 +     uint256 expectedOut = amountsOut[amountsOut.length - 1];
10 +     return (expectedOut * (10_000 - slippageBps)) / 10_000;
11 }
```

Use this value to assign it to `amountOutMinin` for `swapExactTokensForTokens` in `_uniswapInvest` and `uniswapDivest`:

```
1 + function _uniswapInvest(IERC20 token, uint256 amount, uint256
2 +     slippageBps) internal {
3 +     IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
4 +         i_weth;
5 +     uint256 amountOfTokenToSwap = amount / 2;
6 +     s_pathArray = [address(token), address(counterPartyToken)];
7 +
8 +     bool succ = token.approve(address(i_uniswapRouter),
9 +         amountOfTokenToSwap);
10 +     if (!succ) {
11         revert UniswapAdapter__TransferFailed();
12     }
13 }
```

```

11      // Compute slippage-protected min output
12 +    uint256 minOut = _getAmountOutMin(i_uniswapRouter, s_pathArray,
    half, slippageBps);
13
14      uint256[] memory amounts = i_uniswapRouter.
    swapExactTokensForTokens({
15          amountIn: amountOfTokenToSwap,
16 +        amountOutMin: minOut,
17          path: s_pathArray,
18          to: address(this),
19          deadline: block.timestamp
20      });
21
22      ...
23  }

```

- Case 2:

Add a new function to dynamically calculate `amountAMin` / `amountBMin` based on user entered slippage tolerance `slippageBps`, like in the swaps example:

```

1 +  function _calculateLiquidityMin(
2 +      uint256 amountADesired,
3 +      uint256 amountBDesired,
4 +      uint256 slippageBps // e.g., 100 = 1%
5 +  ) internal pure returns (uint256 amountAMin, uint256 amountBMin) {
6 +      amountAMin = (amountADesired * (10_000 - slippageBps)) / 10
    _000;
7 +      amountBMin = (amountBDesired * (10_000 - slippageBps)) / 10
    _000;
8 +  }

```

Use these values to assign them to `amountAMin` / `amountBMin` for `addLiquidity` / `removeLiquidity` in `_uniswapInvest` and `uniswapDivest`:

```

1 +  (uint256 tokenAMin, uint256 tokenBMin) = _calculateLiquidityMin(
2 +      desiredA,
3 +      desiredB,
4 +      100 // 1% slippage tolerance
5 +  );
6
7  (
8      uint256 tokenAmount,
9      uint256 counterPartyTokenAmount,
10     uint256 liquidity
11 ) = i_uniswapRouter.addLiquidity({
12     tokenA: address(token),
13     tokenB: address(counterPartyToken),
14     amountADesired: desiredA,
15     amountBDesired: desiredB,

```

```
16 +         amountAMin: tokenAMin,  
17 +         amountBMin: tokenBMin,  
18         to: address(this),  
19         deadline: block.timestamp  
20     });
```

Same for `removeLiquidity`.

[H-7] Uniswap LP token can be address (0) when vault asset is weth(weth/weth pair), causing functional failure and potential token loss.

Description: In `VaultShares` in constructor, is fetched the uniswap LP token.

```
1 i_uniswapLiquidityToken = IERC20(  
2     i_uniswapFactory.getPair(address(token), address(counterPartyToken))  
3 );
```

There was no validation that `getPair()` actually returned a valid deployed pair. If the vault's token and `counterPartyToken` are identical (e.g., both WETH), Uniswap V2's `getPair()` returns `address(0)` since no such pool can exist.

This means the vault would set:

```
1 i_uniswapLiquidityToken = IERC20(address(0));
```

and continue execution without reverting. Later, when the vault attempts to call `i_uniswapLiquidityToken.balanceOf()`, or Transfer LP tokens, or Redeem liquidity, the transaction reverts with:

```
1 call to non-contract address 0x0000000000000000000000000000000000000000
```

Now the vault becomes permanently unusable (funds could be stuck).

Impact:

Vault logic dependent on LP token calls fails completely. Vault cannot invest, withdraw, or divest. Assets may become stuck or lost if the vault's state becomes unrecoverable.

The issue breaks core functionality and potentially leads to permanent asset loss or vault lockup.

Proof of Concept:

Reproduction:

1. Vault initializes with WETH as asset.
2. `_uniswapInvest()` calls `getPair(WETH, WETH)` → returns `address(0)`.
3. Vault sets `i_uniswapLiquidityToken = IERC20(address(0))`.

4. On withdrawal or reinvestment, vault reverts with: "call to non-contract address 0x00"
5. Vault becomes permanently stuck.

- Proof of Code:

Add this test in WethFork.t.sol in test/fork

Mock Uniswap Factory getPair() always returns a valid address, unlike real mainnet or forked scenario where address(0) is returned.

Code

```
1 function testWethVaultGeneratesUniswapAddress0LP() public hasGuardian {
2     vm.startPrank(user);
3     wETH.deposit{value: mintAmount}(); // convert ETH -> WETH
4     wETH.approve(address(wethVaultShares), mintAmount);
5     wethVaultShares.deposit(mintAmount, user);
6
7     uniswapLiquidityToken = wethVaultShares.i_uniswapLiquidityToken
8         ();
9
10    console.log(
11        "Uniswap LP token address:",
12        address(uniswapLiquidityToken)
13    );
14    assertEq(address(uniswapLiquidityToken), address(0));
15 }
```

Recommended Mitigation:

The current design only allows weth as counter pair token.

Proposed solution:

1. Add a dynamic detection for uniswap counter pair:

- asset: weth -> counter pair: usdc
- asset: not weth -> counter pair: weth

```
1 // Dynamic LP counter-asset selection
2 + if (constructorData.asset == constructorData.weth) {
3     // asset is WETH, so use USDC as the counter asset for Uniswap
4     // LP
5     i_uniswapPairToken = IERC20(constructorData.usdc);
6 + } else {
7     // otherwise, standard case - use WETH
8     i_uniswapPairToken = IERC20(constructorData.weth);
9 + }
```



```
10     i_uniswapLiquidityToken = IERC20(  
11         i_uniswapFactory.getPair(  
12             address(constructorData.asset),  
13             // *** use a variable instead of hardcoding i_weth ***  
14 +         address(i_uniswapPairToken)  
15         )  
16     );
```

Other solutions, but less ideal:

2. Swap weth to usdc -> adds more complexity and moving parts
3. Redistribution of uniswap allocation into aave allocation -> breaks investment diversification
4. Prohibit weth vaults -> restrictive to users

[H-8] nonReentrant is not the First Modifier in VaultShares which makes the functions vulnerable to reentrancy attacks

Description:

Found in: `VaultShares::deposit` `VaultShares::rebalanceFunds` `VaultShares::withdraw` `VaultShares::redeem`

The placement of the `divestThenInvest` modifier before the `nonReentrant` modifier in the function signature creates a critical window for exploitation.

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

Impact: Loss or manipulation of funds

This could allow an attacker to withdraw their share twice or call the divestment logic to retrieve tokens that they shouldn't have access to, leading to a direct loss of user funds from the vault.

A reentrant call could interrupt the process, leading to a situation where the contract's internal accounting is calculated based on incomplete or manipulated data.

Proof of Concept:

- Mechanism of Attack

The Solidity compiler executes modifiers in the order they are listed.

1. `isActive` runs (a simple check, no external call, safe).
2. `divestThenInvest` runs:

- It executes the Divest logic, which includes external calls to third-party protocols (Uniswap and Aave) via `_uniswapDivest` and `_aaveDivest`.
 - Since divestment from these protocols often involves the target protocol sending tokens back to the vault, these are external calls that transfer control flow.
3. Reentrancy Window Opens: An attacker (who controls one of the addresses or a token hook involved in the divestment) can trigger a malicious fallback or hook function in their contract during the divestment phase.
 4. Bypass: The attacker's malicious code executes and immediately calls `rebalanceFunds()` again. Since the execution hasn't reached the `nonReentrant` modifier yet, the reentrancy lock has not been set.
 5. Exploitation: The recursive call to `rebalanceFunds()` proceeds and executes the `divestThenInvest` logic a second time, potentially causing unauthorized state changes, double-claiming of funds, or manipulation of the fund's internal accounting before the first call can complete its state updates.

Recommended Mitigation:

Always place the `nonReentrant` modifier first in the function declaration.

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

[H-9] Using `block.timestamp` for swap deadline offers no protection**Description:**

Found in:

```
UniswapAdapter::_uniswapInvest -> swapExactTokensForTokens(), addLiquidity()  
UniswapAdapter::_uniswapDivest -> swapExactTokensForTokens(), removeLiquidity()
```

The purpose of the deadline parameter in a swap function (like on Uniswap) is to specify a fixed future time after which the transaction will fail.

When deadline: `block.timestamp` is used, the deadline is set to the exact moment the transaction is included in the block.

The swap router contract usually checks:

```
1 require(deadline >= block.timestamp, 'EXPIRED');
```

Impact: Loss of funds(MEV Attacks), transactions stuck forever

The primary function of the swap deadline is to protect the user from slippage and Miner Extractable Value (MEV) attacks, specifically the sandwich attack. By nullifying the deadline check:

Enables Sandwich Attacks: A block producer (validator) can see the user's transaction in the mempool and hold it indefinitely. They can wait for the price to move favorably, execute a transaction just before the user's swap (front-running) to drive the price against the user, and a transaction just after (back-running) to pocket the difference. The user is guaranteed a bad price, and the block producer extracts maximum value.

Allows Arbitrary Delay/Censorship: The validator can delay the transaction for hours or even days, allowing huge price swings that exceed the user's intended slippage tolerance, leading to massive and unintended loss of funds.

The direct result of this vulnerability is the theft or extraction of value from the user, which places it firmly in the high-severity category.

Proof of Concept: If the user passes a fixed future time (e.g., `block.timestamp + 300` seconds from the frontend), the check prevents the transaction from being executed after that time.

If the user passes `block.timestamp` from within the smart contract, the condition becomes `require(block.timestamp >= block.timestamp)`, which will always be true.

The deadline check is completely bypassed, meaning the transaction never expires due to time and offers zero protection against high slippage or attacks by malicious block producers.

Recommended Mitigation:

1. Use a future deadline window Instead of:

```
1 deadline: block.timestamp
```

use something like:

```
1 deadline: block.timestamp + 300 // 5 minutes
```

or make it configurable.

That way: Your transaction won't be stuck forever (it expires in 5 minutes). A validator can't hold it indefinitely for manipulation.

2. Use MEV-resistant submission For high-value transactions: Send the transaction via Flashbots Protect / MEV-Blocker RPC or similar RPC relayers. These systems submit your transaction directly to block builders privately (not via the public mempool), preventing front-running and sandwiching.

Medium

[M-1] Using unsafe ERC20 approve operation can result in stolen assets

Description: The ERC20 approve() function sets the spender's allowance to a specific amount — but if there was already an allowance, it overwrites it. This creates a race condition known as the ERC20 approve front-running issue: If someone can front-run your transaction between two approvals (e.g., approve(100) → approve(200)), they might use the old allowance before it's updated.

Used in:

```
UniswapAdapter::_uniswapInvestAaveAdapter::_aaveInvestVaultGuardiansBase  
::_becomeTokenGuardian
```

Impact: Loss of funds

Proof of Concept:

You currently have allowance = 100. You try to change it to 200. Before your new approval is mined, the spender calls transferFrom(..., 100) — using the old allowance. Your transaction sets allowance to 200 again → they can now pull 200 more. This risk exists any time you use approve() to modify a nonzero allowance directly.

Recommended Mitigation:

Option 1 — Use SafeERC20.safeIncreaseAllowance OpenZeppelin's SafeERC20 has helper functions that safely add to the existing allowance instead of overwriting it: asset.safeIncreaseAllowance(address(i_aavePool), amount); This increases the allowance by amount rather than replacing it, and reverts on failure. You don't even need to check a boolean return — the SafeERC20 wrapper handles that internally.

Option 2 — Use safeApprove(0) before setting new allowance If you prefer the raw approve() semantics, the safe pattern is: asset.safeApprove(address(i_aavePool), 0); asset.safeApprove(address(i_aavePool), amount); This clears the previous allowance first (per ERC20 recommendations), then sets the new one. It ensures no residual approvals exist in between.

In this specific context (Aave and Uniswap), the spender is a trusted protocol — it will only pull tokens during your supply()/addLiquidity() call, so the race condition isn't a real-world exploit risk.

However, it's still a bad pattern to replicate elsewhere, for example VaultGuardiansBase.sol. If your vault ever interacts with other contracts dynamically (like different pools or adapters), it becomes dangerous. So it's still best practice to always use SafeERC20 helpers for approvals.

[M-2] Centralization Risk

Description: These functions are using an access control library that put the power to make changes in the hands of a single entity, which is a centralization issue.

5 Found Instances

- Found in src/dao/VaultGuardianToken.sol Line: 9

```
1 contract VaultGuardianToken is ERC20, ERC20Permit, ERC20Votes,
  Ownable {
```

- Found in src/dao/VaultGuardianToken.sol Line: 21

```
1 function mint(address to, uint256 amount) external onlyOwner {
```

- Found in src/protocol/VaultGuardians.sol Line: 40

```
1 contract VaultGuardians is Ownable, VaultGuardiansBase {
```

- Found in src/protocol/VaultGuardians.sol Line: 71

```
1 function updateGuardianStakePrice(uint256 newStakePrice)
  external onlyOwner {
```

- Found in src/protocol/VaultGuardians.sol Line: 82

```
1 function updateGuardianAndDaoCut(uint256 newCut) external
  onlyOwner {
```

Impact: Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Recommended Mitigation: Consider some of the following recommendations to avoid centralization models: 1. Use a Multi-signature Wallet. 2. Time-lock Contract. 3. Governance by DAO. 4. Use of AccessControl library from OpenZeppelin, which restricts the access to a group of addresses instead a single one.

[M-3] Missing validation for address(0) in VaultShares constructor

Description: In `VaultShares` constructor `aToken` and `Uniswap` pair are assigned by making external calls to `aave getReserveData`, respectively `uniswap factory`.

- Aave: `aTokenAddress == address(0)`

If `getReserveData()` returns an empty struct because that asset isn't supported by Aave, then `aTokenAddress` will be `address(0)`.

Consequence: Later, when your vault tries to deposit to Aave via `_aaveInvest()`, it will interact with `address(0)`, causing a revert (or worse, if unchecked, a failed call without revert detection).

Impact: Users won't lose funds, but deposits/rebalances will revert unexpectedly or result in locked vaults.

- Uniswap: `getPair() == address(0)`

If the Uniswap factory doesn't have a pair for asset/WETH (e.g., `asset == WETH`), it will return `address(0)`.

Consequence: `_uniswapInvest()` and `_uniswapDivest()` may call the router or LP token assuming a valid pair exists. If not handled, that can cause: reverts during deposits or withdrawals, or invalid assumptions about invested balances.

Impact: No funds can be drained (since `address(0)` has no code), but the vault may get stuck, or deposits may fail silently.

```
1 i_aaveAToken = IERC20(  
2     IPool(constructorData.aavePool)  
3     .getReserveData(address(constructorData.asset))  
4     .aTokenAddress  
5 );  
6  
7 i_uniswapLiquidityToken = IERC20(  
8     i_uniswapFactory.getPair(  
9         address(constructorData.asset),  
10        address(i_weth)  
11    )  
12 );
```

Impact: Vaults can be deployed with unsupported or invalid investment configurations, leading to stuck funds or failed deposits.

Proof of Concept: This proves that a vault can be deployed with `i_aaveAToken` or `i_uniswapLiquidityToken` = `address(0)`

- Proof of Code:

Add this test in `test/ProofOfCodes`

Code

```
1 // SPDX-License-Identifier: MIT  
2 pragma solidity ^0.8.20;  
3
```

```
4 import "forge-std/Test.sol";
5 import {VaultShares} from "../src/protocol/VaultShares.sol";
6 import {IVaultShares, IERC4626, IVaultData} from "../src/interfaces/
  IVaultShares.sol";
7 import {DataTypes} from "../src/vendor/DataTypes.sol";
8 import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
9 import {Base_Test} from "../Base.t.sol";
10 import {ERC20Mock} from "../mocks/ERC20Mock.sol";
11
12 // Mocks returning address(0)
13 contract MockAavePool is Base_Test {
14     function getReserveData(
15         address
16     ) external pure returns (DataTypes.ReserveData memory) {
17         // Create a blank configuration map
18         DataTypes.ReserveConfigurationMap memory emptyConfig =
19             DataTypes
20                 .ReserveConfigurationMap({data: 0});
21
22         return
23             DataTypes.ReserveData({
24                 configuration: emptyConfig,
25                 liquidityIndex: 0,
26                 currentLiquidityRate: 0,
27                 variableBorrowIndex: 0,
28                 currentVariableBorrowRate: 0,
29                 currentStableBorrowRate: 0,
30                 lastUpdateTimestamp: 0,
31                 id: 0,
32                 aTokenAddress: address(0),
33                 stableDebtTokenAddress: address(0),
34                 variableDebtTokenAddress: address(0),
35                 interestRateStrategyAddress: address(0),
36                 accruedToTreasury: 0,
37                 unbacked: 0,
38                 isolationModeTotalDebt: 0
39             });
40     }
41
42     contract MockUniswapFactory {
43         function getPair(address, address) external pure returns (address)
44         {
45             return address(0);
46         }
47     }
48
49     contract MockUniswapRouterZero {
50         address private _factory;
51
52         constructor(address factory_) {
```

```
52     _factory = factory_;
53 }
54
55 function factory() external view returns (address) {
56     return _factory;
57 }
58 }
59
60 contract VaultSharesZeroAddressPoC is Base_Test {
61     function testDeployVaultWithZeroAddresses() public {
62         // Deploy mocks used for this PoC (independent of Base_Test's
63         // network config)
64         MockAavePool poolZero = new MockAavePool();
65         MockUniswapFactory factoryZero = new MockUniswapFactory();
66         MockUniswapRouterZero routerZero = new MockUniswapRouterZero(
67             address(factoryZero)
68         );
69
70         // Deploy simple ERC20 mocks for asset/weth/usdc
71         ERC20Mock mockAsset = new ERC20Mock();
72         ERC20Mock mockWETH = new ERC20Mock();
73         ERC20Mock mockUSDC = new ERC20Mock();
74
75         // Make sure the deployer (this) is the vaultGuardians to
76         // satisfy onlyVaultGuardians in constructor
77         IVaultShares.ConstructorData memory cdata = IVaultShares
78             .ConstructorData({
79                 asset: mockAsset,
80                 vaultName: "ZeroAddressVault",
81                 vaultSymbol: "ZAV",
82                 aavePool: address(poolZero),
83                 uniswapRouter: address(routerZero),
84                 weth: address(mockWETH),
85                 usdc: address(mockUSDC),
86                 guardian: address(this),
87                 guardianAndDaoCut: 100, // arbitrary
88                 vaultGuardians: address(this), // important:
89                 // constructor calls updateHoldingAllocation() which is
90                 // onlyVaultGuardians
91                 allocationData: IVaultData.AllocationData({
92                     holdAllocation: 500,
93                     uniswapAllocation: 250,
94                     aaveAllocation: 250
95                 })
96             });
97
98         // Deploy VaultShares (constructor will query aavePoolZero and
99         // factoryZero)
100         VaultShares vault = new VaultShares(cdata);
101
102         console.log("Vault deployed at:", address(vault));
```



```
98
99     // Vulnerable state: the constructor saved zero addresses
    returned by external calls
100     address aToken = vault.getAaveAToken();
101     address lpToken = vault.getUniswapLiquidityToken();
102
103     console.log("aToken stored:", aToken);
104     console.log("uniswap LP token stored:", lpToken);
105
106     assertEq(aToken, address(0), "expected aToken to be address(0)"
    );
107     assertEq(
108         lpToken,
109         address(0),
110         "expected uniswap LP token to be address(0)"
111     );
112 }
113 }
```

Recommended Mitigation: Validate addresses and revert if configuration is invalid.

```
1  constructor(
2      ConstructorData memory constructorData
3  )
4      ERC4626(constructorData.asset)
5      ERC20(constructorData.vaultName, constructorData.vaultSymbol)
6      AaveAdapter(constructorData.aavePool)
7      UniswapAdapter(
8          constructorData.uniswapRouter,
9          constructorData.weth,
10         constructorData.usdc
11     )
12     {
13         i_guardian = constructorData.guardian;
14         i_guardianAndDaoCut = constructorData.guardianAndDaoCut;
15         i_vaultGuardians = constructorData.vaultGuardians;
16         s_isActive = true;
17         updateHoldingAllocation(constructorData.allocationData);
18
19         i_aaveAToken = IERC20(
20             IPool(constructorData.aavePool)
21                 .getReserveData(address(constructorData.asset))
22                 .aTokenAddress
23         );
24
25         // New check for address(0)
26 +         if (address(i_aaveAToken) == address(0)) {
27 +             revert("Invalid Aave aToken: unsupported asset");
28 +         }
29
30         i_uniswapLiquidityToken = IERC20(
```

```
31         i_uniswapFactory.getPair(  
32             address(constructorData.asset),  
33             address(i_weth)  
34         )  
35     );  
36  
37     // New check for address(0)  
38 +     if (address(i_uniswapLiquidityToken) == address(0)) {  
39 +         revert("Uniswap pair not found");  
40 +     }  
41  
42     i_uniswapLiquidityToken = IERC20(i_uniswapLiquidityToken);  
43 }
```

[M-4] Anyone can force rebalances causing value leakage and fund instability

Description:

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

The `rebalanceFunds()` function is declared as public and can be called by any address, because access is not restricted, for example, only to guardians. This function fully divests and re-invests all vault assets through external protocol interactions (Uniswap and Aave).

Because it lacks access control, anyone can repeatedly trigger full rebalance cycles. Each rebalance incurs swap fees, gas costs, and potential slippage, negatively affecting vault performance.

Also, this function is already affected by the separate high-severity issue [H-8] “nonReentrant is not the first modifier”, which allows reentrancy during the `divestThenInvest` phase — amplifying the impact of this unrestricted entrypoint.

Impact:

Attackers or bots can repeatedly call `rebalanceFunds()` and cause cumulative losses.

Recommended Mitigation:

Restrict access using `onlyGuardian` or a designated keeper role. Move `nonReentrant` to be the first modifier, in order to guard against reentrancy.

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

[M-5] Potentially incorrect voting period and delay in governor may affect governance

Description: The `VaultGuardianGovernor` contract, based on OpenZeppelin Contract's Governor, implements two functions to define the voting delay (`votingDelay`) and period (`votingPeriod`). The contract intends to define a voting delay of 1 day, and a voting period of 7 days. It does it by returning the value 1 `days` from `votingDelay` and 7 `days` from `votingPeriod`. In Solidity these values are translated to number of seconds.

However, the `votingPeriod` and `votingDelay` functions, by default, are expected to return number of blocks. Not the number seconds.

Impact: Voting period and delay will be far off what the developers intended, which could potentially affect the intended governance mechanics.

Recommended Mitigation:

Consider updating the functions as follows:

```
1 function votingDelay() public pure override returns (uint256) {
2 -   return 1 days;
3 +   return 7200; // 1 day
4 }
5
6 function votingPeriod() public pure override returns (uint256) {
7 -   return 7 days;
8 +   return 50400; // 1 week
9 }
```

[M-6] Low quorum threshold can lead to governance takeover**Description:**

The `VaultGuardianGovernor` contract sets the quorum requirement to only 4% of total votes:

```
1 GovernorVotesQuorumFraction(4);
```

This means that any proposal reaching just 4% participation is considered valid. In scenarios where token distribution is uneven or voter participation is low, a small minority (or a single large holder) could unilaterally pass proposals, including malicious ones.

Impact: Can affect protocol safety. A 4% quorum threshold makes it feasible for a small coalition or single actor to pass governance actions that alter protocol-critical parameters.

Proof of Concept:

If a tokenholder controls 5% of the governance token supply, they can:

1. Submit a proposal (if proposal threshold allows it).
2. Vote it through with minimal participation.
3. Execute it, gaining control over protocol behavior.

Recommended Mitigation:

Increase quorum fraction to at least 10–20% depending on token distribution and participation expectations.

Introduce a proposal threshold to prevent low-stake or spam proposals.

Low**[L-1] Unused event in VaultGuardians****Description:**

```
1 event VaultGuardians__UpdatedFee(uint256 oldFee, uint256 newFee);
```

Impact: Can increase contract size and complexity.

Recommended Mitigation: Consider using or removing the unused event.

[L-2] Unused State Variable in VaultGuardiansBase

Description: Declared variable is not used anywhere in the contract logic

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

Recommended Mitigation: Consider removing it.

[L-3] Unused error in VaultGuardians, VaultGuardiansBase**Description:**

4 Found Instances

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

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

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

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

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

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

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

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

Impact: Can increase contract size and complexity.

Recommended Mitigation: Consider using or removing the unused errors.

[L-4] PUSH0 Opcode

Description: The primary concern identified in the smart contracts relates to the Solidity compiler version used, specifically pragma solidity 0.8.20;. This version, along with every version after 0.8.19, introduces the use of the PUSH0 opcode. This opcode is not universally supported across all Ethereum Virtual Machine (EVM)-based Layer 2 (L2) solutions. For instance, ZKSync, one of the targeted platforms for this protocol's deployment, does not currently support the PUSH0 opcode.

The consequence of this incompatibility is that contracts compiled with Solidity versions higher than 0.8.19 may not function correctly or fail to deploy on certain L2 solutions.

Impact: The impact of using a Solidity compiler version that includes the PUSH0 opcode is significant for a protocol intended to operate across multiple EVM-based chains. Chains that do not support this opcode will not be able to execute the contracts as intended, resulting in a range of issues from minor malfunctions to complete deployment failures. This limitation directly affects the protocol's goal of wide compatibility and interoperability, potentially excluding it from deployment on key L2 solutions like ZKsync.

Recommended Mitigation: To mitigate this issue and ensure broader compatibility with various EVM-based L2 solutions, it is recommended to downgrade the Solidity compiler version used in the smart contracts to 0.8.19. This version does not utilize the PUSH0 opcode and therefore maintains compatibility with a wider range of L2 solutions, including ZKsync.

[L-5] No address(0) check in VaultGuardiansBase _becomeTokenGuardian for tokenVault and token

Description: These values are passed as parameters, generated by the calling functions (becomeGuardian and becomeTokenGuardian).

The functions (becomeGuardian and becomeTokenGuardian) are actually calling _becomeTokenGuardian and generate a non-zero address via new VaultShares(...) and user input for token. So here the contract logic and the **private** visibility help in guarding against assigning an address(0), but there are some edge cases when this could happen, like if the contract is upgraded and new functions or new logic is added.

```
1 s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));
```

Impact: The primary impact of the missing address(0) check in _becomeTokenGuardian is a potential for Corrupted State leading to a localized Denial of Service (DoS), which is only possible if the contract is upgraded, or if another internal function is added that fails to validate the input.

Recommended Mitigation: Add address(0) check before using the values in the `s_guardian` mapping entry.

```
1 function _becomeTokenGuardian(  
2     IERC20 token,  
3     VaultShares tokenVault  
4 ) private returns (address) {  
5 +     if((address(token) == address(0)) {  
6 +         revert VaultGuardiansBase__InvalidToken();  
7 +     if(IVaultShares(address(tokenVault)) == address(0)) {  
8 +         revert VaultGuardiansBase__InvalidAddress();  
9  
10         s_guardians[msg.sender][token] = IVaultShares(address(  
11             tokenVault));  
12         //...  
13     }
```

[L-6] The guardian receives its vgToken before the vault creation is confirmed introducing a risk of reentrancy

Description: In order to become a vault guardian a user has to deposit a stake amount `s_guardianStakePrice`. In exchange for this stake amount the vault guardian receives the same amount of `vgToken`, which they use to get their stake amount back when they quit being a vault guardian.

Here, in `VaultGuardiansBase::_becomeTokenGuardian`, the code doesn't follow the Check-Effect-Interaction(CEI) principle and the vault guardian receives the `vgToken` before the `tokenVault` creation is confirmed successful.

```
1 function _becomeTokenGuardian(  
2     IERC20 token,  
3     VaultShares tokenVault  
4 ) private returns (address) {  
5     s_guardians[msg.sender][token] = IVaultShares(address(  
6         tokenVault));  
7  
8     emit GuardianAdded(msg.sender, token);  
9  
10    i_vgToken.mint(msg.sender, s_guardianStakePrice);  
11  
12    token.safeTransferFrom(msg.sender, address(this),  
13        s_guardianStakePrice);  
14  
15    bool succ = token.approve(address(tokenVault),  
16        s_guardianStakePrice);  
17    if (!succ) {  
18        revert VaultGuardiansBase__TransferFailed();  
19    }  
20    uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.  
21        sender);  
22    if (shares == 0) {  
23        revert VaultGuardiansBase__TransferFailed();  
24    }  
25    return address(tokenVault);  
26 }
```

Impact: Potential loss of funds

Because the function doesn't follow CEI, it could pose a reentrancy risk, but the function is called internally by `VaultGuardiansBase` and the `vgToken` does not include a reentrancy vector, the Early Reward Minting vulnerability is mitigated for the purpose of a reentrancy attack.

Therefore its impact is reduced from a Critical reentrancy to a Low vulnerability.

However it is still a logic flaw (reward -> deposit -> check) and not good practice in function design.

Recommended Mitigation: Consider following the CEI pattern for operations flow. The `vgToken` mint should be moved after the `deposit` check was performed.

```
1 function _becomeTokenGuardian(  
2     IERC20 token,  
3     VaultShares tokenVault  
4 ) private returns (address) {  
5     // Checks  
6
```

```
7      // Effects
8      s_guardians[msg.sender][token] = IVaultShares(address(
9          tokenVault));
10     emit GuardianAdded(msg.sender, token);
11
12
13     // Interactions
14 -    i_vgToken.mint(msg.sender, s_guardianStakePrice);
15
16     token.safeTransferFrom(msg.sender, address(this),
17         s_guardianStakePrice);
18
19     bool succ = token.approve(address(tokenVault),
20         s_guardianStakePrice);
21     if (!succ) {
22         revert VaultGuardiansBase__TransferFailed();
23     }
24     uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.
25         sender);
26     if (shares == 0) {
27         revert VaultGuardiansBase__TransferFailed();
28     }
29     // Correctly executed after checking if deposit was successful
30 +    i_vgToken.mint(msg.sender, s_guardianStakePrice);
31
32     return address(tokenVault);
33 }
```

[L-7] A GuardianAdded event is emitted before the vault creation is confirmed leading to a false positive in the off-chain information flow

Description: In order to become a vault guardian a user has to deposit a stake amount `s_guardianStakePrice`. The event `GuardianAdded` is emitted before checking if the deposit was actually successful. This can lead to inconsistent information for the event readers.

Here, in `VaultGuardiansBase::_becomeTokenGuardian`, the code doesn't follow the Check-Effect-Interaction(CEI) principle and the event is emitted before the state actually changes.

```
1 function _becomeTokenGuardian(
2     IERC20 token,
3     VaultShares tokenVault
4 ) private returns (address) {
5     s_guardians[msg.sender][token] = IVaultShares(address(
6         tokenVault));
7
8     emit GuardianAdded(msg.sender, token);
9 }
```



```
9         i_vgToken.mint(msg.sender, s_guardianStakePrice);
10
11         token.safeTransferFrom(msg.sender, address(this),
12                                 s_guardianStakePrice);
13
14         bool succ = token.approve(address(tokenVault),
15                                   s_guardianStakePrice);
16         if (!succ) {
17             revert VaultGuardiansBase__TransferFailed();
18         }
19         uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.
20                                             sender);
21         if (shares == 0) {
22             revert VaultGuardiansBase__TransferFailed();
23         }
24         return address(tokenVault);
25     }
```

Impact: While the state on-chain is correctly reverted, the misinformation can negatively affect systems relying on logs: front-ends, analytics and other smart contracts.

Proof of Concept:

1. The event is emitted, signaling to the outside world (off-chain indexers, UI dashboards, explorers) that msg.sender has successfully become a guardian.
2. The transaction continues to the deposit call, which fails (e.g., due to insufficient allowance, a vault error, or a low-level call failure).
3. The transaction executes the revert statement, undoing all state changes (including the s_guardians mapping update and the vgToken mint).
4. Result: External listeners will see an event log for GuardianAdded in a transaction that ultimately failed and reverted.

This creates a confusing and incorrect picture of the contract's state.

Recommended Mitigation: Consider following the CEI pattern for operations flow. The event emission must be moved to the final section of the function, after all checks and interactions have been successfully completed.

```
1 function _becomeTokenGuardian(
2     IERC20 token,
3     VaultShares tokenVault
4 ) private returns (address) {
5     // Checks
6
7     // Effects
8     s_guardians[msg.sender][token] = IVaultShares(address(
9         tokenVault));
```

```

10 -     emit GuardianAdded(msg.sender, token);
11
12
13     // Interactions
14 -     i_vgToken.mint(msg.sender, s_guardianStakePrice);
15
16     token.safeTransferFrom(msg.sender, address(this),
17         s_guardianStakePrice);
18
19     bool succ = token.approve(address(tokenVault),
20         s_guardianStakePrice);
21     if (!succ) {
22         revert VaultGuardiansBase__TransferFailed();
23     }
24     uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.
25         sender);
26     if (shares == 0) {
27         revert VaultGuardiansBase__TransferFailed();
28     }
29     // Correctly executed after checking if deposit was successful
30     i_vgToken.mint(msg.sender, s_guardianStakePrice);
31     // Correctly emitted after checking if deposit was successful
32     emit GuardianAdded(msg.sender, token);
33
34     return address(tokenVault);
35 }

```

[L-8] Incorrect Name and Symbol when creating tokenVault for i_TokenTwo

Description: In `VaultGuardianBase::becomeTokenGuardian` when an user calls `becomeTokenGuardian` a new tokenVault is created for the token the guardian has chosen. There are two accepted tokens, `i_tokenOne` and `i_tokenTwo`, with different name and symbol. But the function assigns `TOKEN_ONE_VAULT_NAME` & `TOKEN_ONE_VAULT_SYMBOL` for both token vaults, which is misleading to off-chain indexers.

```

1 else if (address(token) == address(i_tokenTwo)) {
2     tokenVault = new VaultShares(
3         IVaultShares.ConstructorData({
4             asset: token,
5             // Invalid NAME & SYMBOL assign for i_tokenTwo
6             vaultName: TOKEN_ONE_VAULT_NAME,
7             vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
8             guardian: msg.sender,
9             allocationData: allocationData,
10            aavePool: i_aavePool,
11            uniswapRouter: i_uniswapV2Router,
12            guardianAndDaoCut: s_guardianAndDaoCut,
13            vaultGuardians: address(this),
14            weth: address(i_weth),
15            usdc: address(i_tokenOne)

```

```
16         })
17     }}
```

Impact: The core problem is that the VaultShares contract, which represents the user's staked position, will use this name and symbol when created.

1. User Confusion and Loss of Trust (Primary Impact) Misleading Information:

- A user who stakes LINK (represented by `i_tokenTwo`) will receive a receipt or see a token in their wallet named "USDC Vault Shares" (using `TOKEN_ONE_VAULT_NAME` and `TOKEN_ONE_VAULT_SYMBOL`).
- Trust and Error: This misalignment can cause panic, as the user might believe their funds were accidentally staked into the wrong asset vault, leading to support issues and a significant loss of trust in the protocol.

2. Broken Off-Chain Systems and Integration

- DEX/Aggregator Misidentification: If the protocol integrates with any DEX, portfolio tracker, or DeFi aggregator, those services rely on the token symbol and name to correctly identify the underlying asset. If the LINK vault shares are labeled as USDC vault shares, the tracking systems will display incorrect portfolio values and holdings.
- Indexing Issues: Off-chain services that index all deployed vaults will see two distinct vault addresses (one for USDC, one for LINK) but with identical metadata, making it impossible to programmatically distinguish between them without reading the vault's internal asset variable.

Recommended Mitigation: Use the correct name & symbol for `i_tokenTwo`.

```
1  else if (address(token) == address(i_tokenTwo)) {
2      tokenVault = new VaultShares(
3          IVaultShares.ConstructorData({
4              asset: token,
5              // Use the correct NAME & SYMBOL assign for
6              // i_tokenTwo
7              vaultName: TOKEN_TWO_VAULT_NAME,
8              vaultSymbol: TOKEN_TWO_VAULT_SYMBOL,
9              guardian: msg.sender,
10             allocationData: allocationData,
11             aavePool: i_aavePool,
12             uniswapRouter: i_uniswapV2Router,
13             guardianAndDaoCut: s_guardianAndDaoCut,
14             vaultGuardians: address(this),
15             weth: address(i_weth),
16             usdc: address(i_tokenOne)
17         })
18     })
```

[L-9] Return value not assigned to named return variable in _aaveDivest

Description: In `AaveAdapter _aaveDivest` calls `withdraw` on the Aave pool to retrieve the invested amount. This withdraw call returns a `uint256` value, the received amount. `_aaveDivest` has declared a return variable `amountOfAssetReturned`, but the Aave pool withdraw call return value it is never assigned to it. `_aaveDivest` will always return 0 in this case.

```
1 function _aaveDivest(  
2     IERC20 token,  
3     uint256 amount  
4 ) internal returns (uint256 amountOfAssetReturned) {  
5     i_aavePool.withdraw({  
6         asset: address(token),  
7         amount: amount,  
8         to: address(this)  
9     });  
10 }
```

Impact: The consequence of this is that the calling function receives a 0. Since the return value is not used by the caller, the integrity of the protocol's logic is not affected. It's purely a broken function implementation.

Recommended Mitigation: Assign the withdraw return value to the defined named return variable. Explicit return is optional due to the named return param, but it is good practice for clarity.

```
1 function _aaveDivest(  
2     IERC20 token,  
3     uint256 amount  
4 ) internal returns (uint256 amountOfAssetReturned) {  
5 +     amountOfAssetReturned = i_aavePool.withdraw({  
6         asset: address(token),  
7         amount: amount,  
8         to: address(this)  
9     });  
10 +     return amountOfAssetReturned;  
11 }
```

[L-10] Missing check for sufficient liquidity before calling removeLiquidity()**Description:**

`_uniswapDivest()` calls `i_uniswapRouter.removeLiquidity()` using `liquidityAmount` without verifying that the vault holds at least that many LP tokens. In the current implementation, `divestThenInvest()` always passes the vault's full LP balance, preventing this from triggering. However, if `_uniswapDivest` is reused elsewhere or modified, it could cause unexpected reverts during divestment if the balance check is missing.

Impact:

Potential for failed divestment operations or locked liquidity in edge cases(rounding or future refactors).

Recommended Mitigation:

Add an explicit check in `_uniswapDivest` to ensure `liquidityAmount <= i_uniswapLiquidityToken.balanceOf(address(this))`

```
1 function _uniswapDivest(IERC20 token, uint256 liquidityAmount) internal
  returns (uint256 amountOfAssetReturned) {
2     IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
      i_weth;
3
4     // Check LP balance first
5 +   uint256 lpBal = s_uniswapLP.balanceOf(address(this));
6 +   if (liquidityAmount == 0 || liquidityAmount > lpBal) {
7 +       revert("Invalid liquidity amount");
8 +   }
9
10    (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
      i_uniswapRouter.removeLiquidity({
11        tokenA: address(token),
12        tokenB: address(counterPartyToken),
13        liquidity: liquidityAmount,
14        amountAMin: 0,
15        amountBMin: 0,
16        to: address(this),
17        deadline: block.timestamp
18    });
19    s_pathArray = [address(counterPartyToken), address(token)];
20    uint256[] memory amounts = i_uniswapRouter.
      swapExactTokensForTokens({
21        amountIn: counterPartyTokenAmount,
22        amountOutMin: 0,
23        path: s_pathArray,
24        to: address(this),
25        deadline: block.timestamp
26    });
27    emit UniswapDivested(tokenAmount, amounts[1]);
28    amountOfAssetReturned = amounts[1];
29 }
```

[L-11] Misleading/Incorrect event emission in `updateGuardianAndDaoCut`

Description: In `VaultGuardians::updateGuardianAndDaoCut` the event emitted doesn't correctly describe the function action. The function updates the Guardian and Dao fee, but the event is stating "UpdatedStakePrice".

```
1  /*
2      * @notice Updates the percentage shares guardians & Daos get in
        new vaults
3      * @param newCut the new cut
4      * @dev this value will be divided by the number of shares whenever
        a user deposits into a vault
5      * @dev historical vaults will not have their cuts updated, only
        future vaults
6      */
7  function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner
    {
8      s_guardianAndDaoCut = newCut;
9      // Incorrect event title
10     emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut,
        newCut);
11 }
```

Impact: Causes misleading logs and confusion for off-chain monitoring.

Recommended Mitigation:

Consider using a correctly describing event title, like the already existing and unused event “Updated-Fee”.

```
1  function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
2      s_guardianAndDaoCut = newCut;
3  -     emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut,
        newCut);
4  +     event VaultGuardians__UpdatedFee(s_guardianAndDaoCut, newCut);
5  }
```

[L-12] Ambiguous guardian and DAO cut calculation can lead to misleading configuration

Description:

In `VaultShares::deposit` the guardian and DAO fee is calculated using `shares / i_guardianAndDaoCut`, where `i_guardianAndDaoCut` is set to 1000. This treats the value as a divisor rather than a percentage or basis-point ratio. The naming (“cut”) suggests a proportional fee, which can confuse or lead to incorrect future updates. Additionally, the same fraction is minted twice — once for the guardian and once for the DAO — doubling the total effective fee.

```
1  _mint(i_guardian, shares / i_guardianAndDaoCut);
2  _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);
```

Impact: The total protocol cut is ambiguous and may be misconfigured or misinterpreted, leading to unintended fee rates and inconsistent accounting across vaults.

Proof of Concept:

With `i_guardianAndDaoCut = 1000` => each receives 0.1% of user shares, totaling 0.2%.

Recommended Mitigation:

Clarify the intended meaning of `i_guardianAndDaoCut`. If it is meant to represent a fee in basis points, use:

```
1 + uint256 cutSharesTotal = (shares * i_guardianAndDaoCut) / 10_000;  
2 + uint256 cutShare = cutSharesTotal / 2;  
3   _mint(i_guardian, cutShares);  
4   _mint(i_vaultGuardians, cutShares);
```

Otherwise, rename it to `guardianAndDaoDivisor` and document that it is used as a divisor for clarity.

[L-13] Missing event emission on user deposit**Description:**

The `deposit` function in `VaultShares` doesn't emit an event when the operation is completed, indicating that the deposit happened.. This function changes the state by minting vault shares to the user and shares cut to the DAO and guardian. Events should be emitted when state is changed.

While OpenZeppelin's ERC4626 emits a Deposit event for the main user deposit, the additional minting to the guardian and DAO happens silently.

Impact:

Off-chain systems and indexers cannot accurately track vault inflows or fee distributions, reducing transparency and potentially breaking accounting or analytics that rely on emitted events.

Recommended Mitigation:

Emit a dedicated event when shares are minted to the guardian and DAO, on deposit operations:

```
1 + event GuardianAndDaoSharesMinted(address indexed guardian, address  
   indexed dao, uint256 guardianShares, uint256 daoShares);  
2  
3 + emit GuardianAndDaoSharesMinted(i_guardian, i_vaultGuardians,  
   shares / i_guardianAndDaoCut, shares / i_guardianAndDaoCut);
```

This improves transparency and facilitates accurate monitoring of fee allocations.

Informational

[I-1] Unused interfaces `IInvestableUniverseAdapter` and `IVaultGuardians` with commented-out functions

Description:

1. The interface `IInvestableUniverseAdapter` defines commented-out function stubs and is not referenced anywhere in the codebase.
2. The interface `IVaultGuardians` has an empty body and is not referenced anywhere in the codebase.

Impact: No runtime impact, can increase audit complexity and create confusion

Recommended Mitigation: Consider removing unused files and data

[I-2] Insufficient test coverage

File	% Lines	% Statements	% Branches	% Funcs
src/abstract/AStaticTokenData.sol	100.00% (4/4)	100.00% (2/2)	100.00% (0/0)	100.00% (2/2)
src/abstract/AStaticUSDData.sol	100.00% (4/4)	100.00% (2/2)	100.00% (0/0)	100.00% (2/2)
src/abstract/AStaticWethData.sol	100.00% (4/4)	100.00% (2/2)	100.00% (0/0)	100.00% (2/2)
src/dao/VaultGuardianGovernor.sol	0.00% (0/6)	0.00% (0/4)	100.00% (0/0)	0.00% (0/3)
src/dao/VaultGuardianToken.sol	66.67% (4/6)	50.00% (2/4)	100.00% (0/0)	66.67% (2/3)
src/protocol/VaultGuardians.sol	100.00% (10/10)	100.00% (8/8)	100.00% (0/0)	100.00% (3/3)
src/protocol/VaultGuardiansBase.sol	95.65% (66/69)	93.55% (58/62)	81.82% (9/11)	100.00% (16/16)
src/protocol/VaultShares.sol	91.55% (65/71)	91.80% (56/61)	50.00% (4/8)	94.74% (18/19)
src/protocol/investableUniverseAdapters/AaveAdapter.sol	66.67% (6/9)	71.43% (5/7)	0.00% (0/1)	66.67% (2/3)
src/protocol/investableUniverseAdapters/UniswapAdapter.sol	61.54% (16/26)	61.29% (19/31)	0.00% (0/3)	66.67% (2/3)
Total	85.65% (179/209)	84.15% (154/183)	56.52% (13/23)	87.50% (49/56)

Figure 1: alt text

Recommended Mitigation: Aim to get test coverage up to over 90% for all files and improve Branch testing.

[I-3] No check on return value in `_aaveDivest`

Description: In `AaveAdapter`, `_aaveDivest` defines a named return value which is never assigned (see L-9) and never checked

```

1 function _aaveDivest(
2     IERC20 token,
3     uint256 amount
4 ) internal returns (uint256 amountOfAssetReturned) {
5     i_aavePool.withdraw({

```



```
6         asset: address(token),
7         amount: amount,
8         to: address(this)
9     });
10
11 }
```

Impact: There is minimal impact because the return value is not used by other functions and Aave's `withdraw` is expected to revert on failure, mitigating the most severe risk.

Recommended Mitigation: Add a check on the return value of `_aaveDivest`

```
1 function _aaveDivest(
2     IERC20 token,
3     uint256 amount
4 ) internal returns (uint256 amountOfAssetReturned) {
5 +     amountOfAssetReturned = i_aavePool.withdraw({
6         asset: address(token),
7         amount: amount,
8         to: address(this)
9     });
10 +     if (amountOfAssetReturned == 0) {
11 +         revert AaveAdapter__NothingToWithdraw();
12 +     }
13 +     return amountOfAssetReturned;
14 }
```

Alternatively, remove the unused return value from the function signature to avoid confusion.

[I-4] Typo in function title in `getUniswapLiquidityToken`

Description:

In `VaultShares` the function title contains a misspelled word (missing an "i"): `Liquidty`, instead of `Liquidity`

Impact: No runtime impact, can create confusion

Recommended Mitigation: Consider correcting the function title to `getUniswapLiquidityToken`

```
1 + function getUniswapLiquidityToken() external view returns (address)
2     {
3         return address(i_uniswapLiquidityToken);
4     }
```

[I-5] Missing mechanism for adding new investment platforms**Description:**

The README file states that the project can add new investment strategies, but there is no function or mechanism to allow this.

“The protocol is upgradeable so that if any of the platforms in the investable universe change, or we want to add more, we can do so.”

In the current version, there is only the possibility to update the investment strategy/allocation.

Impact: Misleading description of the protocol abilities.

Recommended Mitigation: Consider removing the part that states that the protocol can add new investment platforms or implement a mechanism that allows new investment platforms to be safely added.

Gas**[G-1] Functions marked `public` are not used internally**

Description: A function is declared as `public` when it can be declared as external. The `public` visibility forces the compiler to generate code that copies all function arguments from calldata to memory (specifically, into the free memory pointer region) when the function is called.

2 Found Instances

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

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

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

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

This copying operation generates a significant gas consumption for every function call. When the function is only intended to be called by external accounts (EOAs) or other contracts, and not internally by other functions within the same contract.

Impact: Using `public` leads to waste of gas, directly increasing the transaction cost for the user.

Recommended Mitigation: Change visibility from `public` to `external`, if not used internally.