Vault Guardians Audit Report

mhng

November 18, 2025

Boss Bridge Initial Audit Report

Version 0.1

Cyfrin.io

November 18, 2025

Vault Guardians Audit Report

mhng

November 18, 2025

Vault Guardians Audit Report

Prepared by: mhng Lead Auditors: mhng

- mhng

Assisting Auditors:

- None

Table of contents

See table

    - [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 tokenVault 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 emision 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 getUniswapLiquidtyToken
    - [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 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

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

`0600272180b6b6103f523b0ef65b070064158303`

Scope

```
./src/
#-- abstract
|   #-- AStaticTokenData.sol
|   #-- AStaticUSDCData.sol
|   #-- AStaticWethData.sol
#-- dao
|   #-- VaultGuardianGovernor.sol
|   #-- VaultGuardianToken.sol
```

```
#-- interfaces
|   #-- IVaultData.sol
|   #-- IVaultGuardians.sol
|   #-- IVaultShares.sol
|   #-- InvestableUniverseAdapter.sol
#-- protocol
|   #-- VaultGuardians.sol
|   #-- VaultGuardiansBase.sol
|   #-- VaultShares.sol
|   #-- investableUniverseAdapters
|       #-- AaveAdapter.sol
|       #-- UniswapAdapter.sol
#-- vendor
    #-- DataTypes.sol
    #-- IPool.sol
    #-- IUniswapV2Factory.sol
    #-- IUniswapV2Router01.sol
```

- 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 despoited 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
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.

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

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

```
    function testTotalAssetsReturnsIncorrectAmount()
            public
            hasGuardian
            hasTokenGuardian
        {
            // user deposits usdc
            usdc.mint(mintAmount, user);
            vm.startPrank(user);
            usdc.approve(address(usdcVaultShares), mintAmount);
            usdcVaultShares.deposit(mintAmount, user);
            vm.stopPrank();

            uint256 expectedVaultBalance = mintAmount;
            uint256 actualVaultBalance = usdcVaultShares.totalAssets();
            uint256 delta = (expectedVaultBalance * 6) / 10; // missing deposit amount appro

            console.log("Expected Vault Balance: ", expectedVaultBalance);
            console.log("Actual Vault Balance: ", actualVaultBalance);
            console.log("Expected difference in vault amount: ", delta);
            console.log(
                "Actual difference in vault amount: ",
                expectedVaultBalance - actualVaultBalance
            );

            // Shows that there is a significant difference between expected and actual vaul
```

```
        assertApproxEqAbs(actualVaultBalance, expectedVaultBalance, delta);
    }
```

Recommended Mitigation:

1. Override OpenZepelin'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:

```
function _aaveInvest(IERC20 asset, uint256 amount) internal {
    bool succ = asset.approve(address(i_aavePool), amount);
    if (!succ) {
        revert AaveAdapter__TransferFailed();
    }
    i_aavePool.supply({
        asset: address(asset),
        amount: amount,
        onBehalfOf: address(this), // decides who gets Aave's aTokens for the investment
        referralCode: 0
    });
}
```

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

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

import "forge-std/Test.sol";
import {AaveAdapter} from "../../src/protocol/InvestableUniverseAdapters/AaveAdapter.sol";
import {IPool} from "../../src/vendor/IPool.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

// Mock USDC token with 6 decimals
contract MockUSDC is ERC20 {
    constructor() ERC20("Mock USDC", "USDC") {}

    function decimals() public pure override returns (uint8) {
        return 6;
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}

// Mock Aave pool
```

```
contract MockAavePool {
    function supply(
        address asset,
        uint256 amount,
        address onBehalfOf,
        uint16
    ) external {
        IERC20(asset).transferFrom(onBehalfOf, address(this), amount);
    }
}

// Mock vault with Aave Adapter for investment strategy
contract MockVault is AaveAdapter {
    IERC20 public immutable asset;

    constructor(address pool, IERC20 _asset) AaveAdapter(pool) {
        asset = _asset;
    }

    function invest(uint256 amount) external {
        // Simulate deposit of tokens into the vault
        _aaveInvest(asset, amount);
    }
}

// ----------------------------
//       Test Contract
// ----------------------------

contract AaveDecimalsMismatchNon18TokensPoC is Test {
    MockUSDC private usdc;
    MockAavePool private aavePool;
    MockVault private vault;

    error ERC20InsufficientBalance(address, uint256, uint256);

    function setUp() public {
        usdc = new MockUSDC();
        aavePool = new MockAavePool();
        vault = new MockVault(address(aavePool), usdc);

        usdc.mint(address(vault), 1e6);
    }

    function test_USDCRevertsDueToDecimalMismatch() public {
        // Simulate normalized 18-decimal math, vault believes amount is 1e18
```

```
        uint256 normalAmount = 1e18;

        uint256 currentBalance = usdc.balanceOf(address(vault));
        console.log("Current Balance:", currentBalance);

        uint256 requiredBalance = normalAmount;
        console.log("Required Balance:", requiredBalance);

        // Expect revert because trying to transfer 1e18 USDC
        vm.expectRevert(
            abi.encodeWithSelector(
                ERC20InsufficientBalance.selector,
                address(vault),
                currentBalance,
                requiredBalance
            )
        );

        vault.invest(normalAmount);
    }
}
```

Recommended Mitigation:

Add proper decimal scaling before interacting with Aave:

```
    function _aaveInvest(IERC20 asset, uint256 amount) internal {
+       uint8 assetDecimals = IERC20Metadata(address(asset)).decimals();
+       uint256 scaledAmount = amount;

+       if (assetDecimals < 18) {
+           scaledAmount = amount / (10 ** (18 - assetDecimals));
+       } else if (assetDecimals > 18) {
+           scaledAmount = amount * (10 ** (assetDecimals - 18));
+       }

        bool succ = asset.approve(address(i_aavePool), amount);
        if (!succ) {
            revert AaveAdapter__TransferFailed();
        }
        i_aavePool.supply({
            asset: address(asset),
+           amount: scaledAmount,
            onBehalfOf: address(this),
            referralCode: 0
        });
    }
```

[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

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

import "forge-std/Test.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";
import {IVaultShares, IVaultData} from "../../src/interfaces/IVaultShares.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {DataTypes} from "../../src/vendor/DataTypes.sol";

// ----------------------------
//        Uniswap mocks
// ----------------------------

contract UniswapLP is ERC20Mock {
    // Inherit ERC20Mock: free mint/burn helpers already available
}

contract UniswapFactory {
    address public immutable pair;

    constructor(address pair_) {
        pair = pair_;
```

```
    }

    function getPair(address, address) external view returns (address) {
        return pair;
    }
}

// This router is Uniswap-V2-shaped enough for VaultShares.UniswapAdapter
contract UniswapRouter {
    address private _factory;
    UniswapLP private _lp;

    constructor(address factory_, address lp_) {
        _factory = factory_;
        _lp = UniswapLP(lp_);
    }

    function factory() external view returns (address) {
        return _factory;
    }

    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 /* amountOutMin */,
        address[] calldata path,
        address to,
        uint256 /* deadline */
    ) external returns (uint256[] memory amounts) {
        // pull input
        ERC20Mock(path[0]).transferFrom(msg.sender, address(this), amountIn);

        // use mint to avoid needing pre-funding
        ERC20Mock(path[1]).mint(amountIn, to);

        amounts = new uint256[](path.length);
        amounts[0] = amountIn;
        amounts[1] = amountIn;
    }

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 /* amountAMin */,
        uint256 /* amountBMin */,
```

```
        address to,
        uint256 /* deadline */
    ) external returns (uint256 amountA, uint256 amountB, uint256 liquidity) {
        // pull all desired tokens from caller
        ERC20Mock(tokenA).transferFrom(
            msg.sender,
            address(this),
            amountADesired
        );
        ERC20Mock(tokenB).transferFrom(
            msg.sender,
            address(this),
            amountBDesired
        );

        _lp.mint(100 ether, to);

        return (amountADesired, amountBDesired, 100 ether);
    }

    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 /* amountAMin */,
        uint256 /* amountBMin */,
        address to,
        uint256 /* deadline */
    ) external returns (uint256 amountA, uint256 amountB) {
        // take the LP from caller
        _lp.transferFrom(msg.sender, address(this), liquidity);
        // burn
        _lp.burn(liquidity, address(this));

        // return *1 wei* of each token to the vault
        ERC20Mock(tokenA).mint(1, to);
        ERC20Mock(tokenB).mint(1, to);

        return (1, 1);
    }
}

// ---------------------------
//         Aave mocks
// ---------------------------
```

```
// Deploy a small MockAavePool that returns (aTokenAddress, 0,0,...)
contract MockAavePool {
    address public immutable aTokenAddr;

    constructor(address _aTokenAddr) {
        aTokenAddr = _aTokenAddr;
    }

    // Must match the interface signature used by VaultShares
    function getReserveData(
        address
    ) external view returns (DataTypes.ReserveData memory r) {
        // leave everything at the zero default, just set aTokenAddress
        r.aTokenAddress = aTokenAddr;
    }

    function supply(
        address /* asset */,
        uint256 amount,
        address onBehalfOf,
        uint16 /* referralCode */
    ) external {
        // For the PoC we don't simulate any lending, just acknowledge the call.
        if (amount > 0) {
            ERC20Mock(aTokenAddr).mint(amount, onBehalfOf);
        }
    }
}

// ---------------------------
//       Test Contract
// ---------------------------

contract MissingApprovalsUniswapDivestPoC_Test is Test {
    VaultShares private vault;
    ERC20Mock private asset;
    ERC20Mock private tokenOne;
    ERC20Mock private mockWETH; // distinct mock WETH
    ERC20Mock private awethTokenMock;

    UniswapRouter router;
    UniswapLP lp;

    error ERC20InsufficientAllowance(address, uint256, uint256);

    function setUp() public {
```

```
        asset = new ERC20Mock();
        tokenOne = new ERC20Mock();
        mockWETH = new ERC20Mock(); // distinct mock WETH
        awethTokenMock = new ERC20Mock();

        MockAavePool pool = new MockAavePool(address(awethTokenMock));

        // LP + Factory + Router
        lp = new UniswapLP();
        UniswapFactory factory = new UniswapFactory(address(lp));
        router = new UniswapRouter(address(factory), address(lp));

        // Build constructor data
        IVaultShares.ConstructorData memory c = IVaultShares.ConstructorData({
            asset: asset,
            vaultName: "PoC Vault",
            vaultSymbol: "POC",
            guardian: address(this),
            allocationData: IVaultData.AllocationData({
                holdAllocation: 0,
                uniswapAllocation: 1000,
                aaveAllocation: 0
            }),
            aavePool: address(pool),
            uniswapRouter: address(router),
            guardianAndDaoCut: 100,
            vaultGuardians: address(this),
            weth: address(mockWETH),
            usdc: address(tokenOne)
        });

        vault = new VaultShares(c);

        // Label for nicer traces
        vm.label(address(vault), "VaultShares(POC)");
        vm.label(address(asset), "ASSET");
        vm.label(address(tokenOne), "TOKEN_ONE");
        vm.label(address(factory), "Factory");
        vm.label(address(router), "Router");
        vm.label(address(lp), "LP");
    }

    function test_MissingApprovalsUniswapDivestResultsInRevert() public {
        // User mints and deposits 100 ASSET
        address user = address(0xBEEF);
        uint256 depositAmt = 100 ether;
```

```
asset.mint(depositAmt, user);

// pre-fund the vault so the double-accounting won't cause insufficient balance
// this must happen after vault creation (setUp) but before deposit triggers invest
asset.mint(depositAmt / 2, address(vault));

// Approve & deposit
vm.startPrank(user);
asset.approve(address(vault), depositAmt);
vault.deposit(depositAmt, user);
vm.stopPrank();

uint256 currentAllowance = lp.allowance(
    address(vault),
    address(router)
);
uint256 requiredAllowance = lp.balanceOf(address(vault));

console.log("Current allowance:", currentAllowance);
console.log("Required allowance:", requiredAllowance);

// If we call redeem(), expectRevert won't be able to pick up the revert, because it
// expectRevert checks the top level call for revert
// removeLiquidity() and swapExactTokensForTokens() are nested calls inside redeem()

// Next removeLiquidity() attempt should revert because of insufficient allowance
vm.expectRevert(
    abi.encodeWithSelector(
        ERC20InsufficientAllowance.selector,
        address(router),
        currentAllowance,
        requiredAllowance
    )
);

vm.startPrank(address(vault));
router.removeLiquidity(
    address(asset),
    address(mockWETH),
    depositAmt,
    0,
    0,
    user,
    1
);
```

```
        vm.stopPrank();
    }
}
```

Recommended Mitigation:

Add safe increase allowance for LP and Counterparty Token

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

        // Approve uniswapRouter to transfer LP from VaultShares
+       if (
+            s_uniswapLP.allowance(address(this), address(i_uniswapRouter)) == 0
+         ) {
+             s_uniswapLP.safeIncreaseAllowance(
+                 address(i_uniswapRouter),
+                 type(uint256).max
+             );
+           };

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


        // Approve uniswapRouter to transfer counterPartyToken from VaultShares
+       if (
+            counterPartyToken.allowance(address(this), address(i_uniswapRouter)) == 0
+         ) {
+             counterPartyToken.safeIncreaseAllowance(
+                 address(i_uniswapRouter),
+                 type(uint256).max
+             );
+           };

        uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
            amountIn: counterPartyTokenAmount,
            amountOutMin: 0,
            path: s_pathArray,
```

```
            to: address(this),
            deadline: block.timestamp
        });
        emit UniswapDivested(tokenAmount, amounts[1]);
        amountOfAssetReturned = amounts[1];
    }
```

[H-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 repaied 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`.

```
    function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault) private returns (ad
        s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));
@>      i_vgToken.mint(msg.sender, s_guardianStakePrice);
        emit GuardianAdded(msg.sender, token);
        token.safeTransferFrom(msg.sender, address(this), s_guardianStakePrice);
        token.approve(address(tokenVault), s_guardianStakePrice);
        tokenVault.deposit(s_guardianStakePrice, msg.sender);
        return address(tokenVault);
    }
```

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:

```
"sweepErc20s(address)": "942d0ff9",
"transferOwnership(address)": "f2fde38b",
"updateGuardianAndDaoCut(uint256)": "9e8f72a4",
"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`

```
function testDaoTakeover() public hasGuardian hasTokenGuardian {
    address maliciousGuardian = makeAddr("maliciousGuardian");
    uint256 startingVoterUsdcBalance = usdc.balanceOf(maliciousGuardian);
    uint256 startingVoterWethBalance = weth.balanceOf(maliciousGuardian);
    assertEq(startingVoterUsdcBalance, 0);
    assertEq(startingVoterWethBalance, 0);

    VaultGuardianGovernor governor = VaultGuardianGovernor(payable(vaultGuardians.owner(
    VaultGuardianToken vgToken = VaultGuardianToken(address(governor.token()));

    // Flash loan the tokens, or just buy a bunch for 1 block
    weth.mint(mintAmount, maliciousGuardian); // The same amount as the other guardians
    uint256 startingMaliciousVGTokenBalance = vgToken.balanceOf(maliciousGuardian);
    uint256 startingRegularVGTokenBalance = vgToken.balanceOf(guardian);
    console.log("Malicious vgToken Balance:\t", startingMaliciousVGTokenBalance);
    console.log("Regular vgToken Balance:\t", startingRegularVGTokenBalance);

    // Malicious Guardian farms tokens
    vm.startPrank(maliciousGuardian);
    weth.approve(address(vaultGuardians), type(uint256).max);
    for (uint256 i; i < 10; i++) {
        address maliciousWethSharesVault = vaultGuardians.becomeGuardian(allocationData)
        IERC20(maliciousWethSharesVault).approve(
            address(vaultGuardians),
            IERC20(maliciousWethSharesVault).balanceOf(maliciousGuardian)
        );
        vaultGuardians.quitGuardian();
    }
    vm.stopPrank();

    uint256 endingMaliciousVGTokenBalance = vgToken.balanceOf(maliciousGuardian);
    uint256 endingRegularVGTokenBalance = vgToken.balanceOf(guardian);
    console.log("Malicious vgToken Balance:\t", endingMaliciousVGTokenBalance);
    console.log("Regular vgToken Balance:\t", endingRegularVGTokenBalance);
}
```

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.

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

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

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

import "forge-std/Test.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";
import {IVaultShares, IVaultData} from "../../src/interfaces/IVaultShares.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {DataTypes} from "../../src/vendor/DataTypes.sol";
```

```
// ----------------------------
//        Uniswap mocks
// ----------------------------

contract MaliciousLP is ERC20Mock {
    // Inherit ERC20Mock: free mint/burn helpers already available
}

contract MaliciousFactory {
    address public immutable pair;

    constructor(address pair_) {
        pair = pair_;
    }

    function getPair(address, address) external view returns (address) {
        return pair;
    }
}

contract MaliciousRouter {
    // This router is Uniswap-V2-shaped enough for VaultShares.UniswapAdapter
    // Behavior:
    //  - swapExactTokensForTokens: takes all input, returns *1 wei* as output
    //  - addLiquidity: pulls all desired tokens, mints *1 wei* LP
    //  - removeLiquidity: burns all LP, returns *1 wei* of each token
    // All succeed because min amounts are zero.

    address private _factory;
    MaliciousLP private _lp;

    constructor(address factory_, address lp_) {
        _factory = factory_;
        _lp = MaliciousLP(lp_);
    }

    function factory() external view returns (address) {
        return _factory;
    }

    function swapExactTokensForTokens(
        uint256 amountIn,
        uint256 /* amountOutMin */, // zero in vulnerable code
        address[] calldata path,
        address to,
```

```
    uint256 /* deadline */
) external returns (uint256[] memory amounts) {
    // pull input
    ERC20Mock(path[0]).transferFrom(msg.sender, address(this), amountIn);

    // mint or transfer *1 wei* of output to `to`
    // use mint to avoid needing pre-funding
    ERC20Mock(path[1]).mint(1, to);

    amounts = new uint256[](path.length);
    amounts[0] = amountIn; // uniswap-style return
    amounts[1] = 1;
}

function addLiquidity(
    address tokenA,
    address tokenB,
    uint256 amountADesired,
    uint256 amountBDesired,
    uint256 amountAMin, // zero in vulnerable code
    uint256 amountBMin, // zero in vulnerable code
    address to,
    uint256 /* deadline */
) external returns (uint256 amountA, uint256 amountB, uint256 liquidity) {
    // pull all desired tokens from caller
    ERC20Mock(tokenA).transferFrom(
        msg.sender,
        address(this),
        amountADesired
    );
    ERC20Mock(tokenB).transferFrom(
        msg.sender,
        address(this),
        amountBDesired
    );

    // min checks are zero -> always satisfied
    require(amountAMin == 0 && amountBMin == 0, "not the vulnerable path");

    // mint only 1 wei of LP to the vault (terrible deal)
    _lp.mint(1, to);

    return (amountADesired, amountBDesired, 1);
}

function removeLiquidity(
```

```
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin, // zero in vulnerable code
        uint256 amountBMin, // zero in vulnerable code
        address to,
        uint256 /* deadline */
    ) external returns (uint256 amountA, uint256 amountB) {
        // take the LP from caller
        _lp.transferFrom(msg.sender, address(this), liquidity);
        // burn
        _lp.burn(liquidity, address(this));

        // return *1 wei* of each token to the vault
        ERC20Mock(tokenA).mint(1, to);
        ERC20Mock(tokenB).mint(1, to);

        require(amountAMin == 0 && amountBMin == 0, "not the vulnerable path");

        return (1, 1);
    }
}

// ----------------------------
//         Aave mocks
// ----------------------------

// Deploy a small MockAavePool that returns (aTokenAddress, 0,0,...)
contract MockAavePool {
    address public immutable aTokenAddr;

    constructor(address _aTokenAddr) {
        aTokenAddr = _aTokenAddr;
    }

    // Must match the interface signature used by VaultShares
    function getReserveData(
        address
    ) external view returns (DataTypes.ReserveData memory r) {
        // leave everything at the zero default, just set aTokenAddress
        r.aTokenAddress = aTokenAddr;
    }

    function supply(
        address /* asset */,
        uint256 amount,
```

```
        address onBehalfOf,
        uint16 /* referralCode */
    ) external {
        if (amount > 0) {
            ERC20Mock(aTokenAddr).mint(amount, onBehalfOf);
        }
    }
}

// ----------------------------
//          PoC test
// ----------------------------

contract SlippageZeroMin_PoC_Test is Test {
    VaultShares private vault;
    ERC20Mock private asset;
    ERC20Mock private tokenOne;
    ERC20Mock private mockWETH; // distinct mock WETH
    ERC20Mock private awethTokenMock;

    MaliciousLP lp;
    MaliciousFactory factory;
    MaliciousRouter router;

    function setUp() public {
        asset = new ERC20Mock();
        tokenOne = new ERC20Mock();
        mockWETH = new ERC20Mock(); // distinct mock WETH
        awethTokenMock = new ERC20Mock();

        MockAavePool pool = new MockAavePool(address(awethTokenMock));

        // LP + Factory + Router (malicious)
        lp = new MaliciousLP();
        factory = new MaliciousFactory(address(lp));
        router = new MaliciousRouter(address(factory), address(lp));

        // Build constructor data with:
        IVaultShares.ConstructorData memory c = IVaultShares.ConstructorData({
            asset: asset,
            vaultName: "PoC Vault",
            vaultSymbol: "POC",
            guardian: address(this),
            allocationData: IVaultData.AllocationData({
                holdAllocation: 0,
                uniswapAllocation: 1000, // 100% to Uniswap path (will hit vulnerable code)
```

```
                aaveAllocation: 0
            }),
            aavePool: address(pool),
            uniswapRouter: address(router),
            guardianAndDaoCut: 100,
            vaultGuardians: address(this),
            weth: address(mockWETH),
            usdc: address(tokenOne)
        });

        vault = new VaultShares(c);

        // Label for nicer traces
        vm.label(address(vault), "VaultShares(POC)");
        vm.label(address(asset), "ASSET");
        vm.label(address(tokenOne), "TOKEN_ONE");
        vm.label(address(factory), "MaliciousFactory");
        vm.label(address(router), "MaliciousRouter");
        vm.label(address(lp), "MaliciousLP");
    }

    function test_Slippage_WithZeroMins_LosesValue() public {
        // User mints and deposits 100 ASSET
        address user = address(0xBEEF);
        uint256 depositAmt = 100 ether;

        asset.mint(depositAmt, user);

        // pre-fund the vault so the double-accounting won't cause insufficient balance
        // this must happen after vault creation (setUp) but before deposit triggers invest
        asset.mint(depositAmt / 2, address(vault));

        // Approve & deposit
        vm.startPrank(user);
        asset.approve(address(vault), depositAmt);
        uint256 shares = vault.deposit(depositAmt, user);
        vm.stopPrank();

        // Adding the uniswap approvals that are missing in the original code
        vm.startPrank(address(vault));
        lp.approve(address(router), type(uint256).max);
        tokenOne.approve(address(router), type(uint256).max);
        mockWETH.approve(address(router), type(uint256).max);
        vm.stopPrank();

        // Full redeem immediately - vulnerable paths:
```

```
        //  - invest: swap with amountOutMin=0, addLiquidity with amountAMin/BMin=0
        //  - redeem: removeLiquidity with amountAMin/BMin=0, swap with amountOutMin=0
        vm.startPrank(user);
        uint256 assetsOut = vault.redeem(vault.balanceOf(user), user, user);
        vm.stopPrank();

        // Show the damage in logs
        console.log("Deposit:", depositAmt);
        console.log("Shares minted:", shares);
        console.log("Assets out on full redeem:", assetsOut);

        // Assert a large loss -> 10x
        assertLt(
            assetsOut,
            depositAmt / 10,
            "Expected significant loss due to zero slippage protections"
        );
    }
}
```

Recommended Mitigation:

Use only the remaining base tokens for amountADesired:

```
+       uint256 tokenLeft = amount - amounts[0];

        (
            uint256 tokenAmount,
            uint256 counterPartyTokenAmount,
            uint256 liquidity
        ) = i_uniswapRouter.addLiquidity({
                tokenA: address(token),
                tokenB: address(counterPartyToken),
                // Use only the remaining base tokens after swap for amountADesired
+               amountADesired: uint256 tokenLeft,
                amountBDesired: amounts[1],
                amountAMin: 0,
                amountBMin: 0,
                to: address(this),
                deadline: block.timestamp
            });
```

[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 (swapExactTo-

kensForTokens, 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()

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

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()

```
(, , uint256 liquidity) = i_uniswapRouter.addLiquidity({
    ...
    amountAMin: 0, //  no protection
    amountBMin: 0, //  no protection
    ...
});

(, ) = i_uniswapRouter.removeLiquidity({
    ...
    amountAMin: 0, //  no protection
    amountBMin: 0, //  no protection
    ...
});
```

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

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

import "forge-std/Test.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";
```

```
import {IVaultShares, IVaultData} from "../../src/interfaces/IVaultShares.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {DataTypes} from "../../src/vendor/DataTypes.sol";

// ----------------------------
//        Uniswap mocks
// ----------------------------

contract MaliciousLP is ERC20Mock {
    // Inherit ERC20Mock: free mint/burn helpers already available
}

contract MaliciousFactory {
    address public immutable pair;

    constructor(address pair_) {
        pair = pair_;
    }

    function getPair(address, address) external view returns (address) {
        return pair;
    }
}

contract MaliciousRouter {
    // This router is Uniswap-V2-shaped enough for VaultShares.UniswapAdapter
    // Behavior:
    //  - swapExactTokensForTokens: takes all input, returns *1 wei* as output
    //  - addLiquidity: pulls all desired tokens, mints *1 wei* LP
    //  - removeLiquidity: burns all LP, returns *1 wei* of each token
    // All succeed because min amounts are zero.

    address private _factory;
    MaliciousLP private _lp;

    constructor(address factory_, address lp_) {
        _factory = factory_;
        _lp = MaliciousLP(lp_);
    }

    function factory() external view returns (address) {
        return _factory;
    }

    function swapExactTokensForTokens(
        uint256 amountIn,
```

```
        uint256 /* amountOutMin */, // zero in vulnerable code
        address[] calldata path,
        address to,
        uint256 /* deadline */
    ) external returns (uint256[] memory amounts) {
        // pull input
        ERC20Mock(path[0]).transferFrom(msg.sender, address(this), amountIn);

        // mint or transfer *1 wei* of output to `to`
        // use mint to avoid needing pre-funding
        ERC20Mock(path[1]).mint(1, to);

        amounts = new uint256[](path.length);
        amounts[0] = amountIn; // uniswap-style return
        amounts[1] = 1;
    }

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint256 amountADesired,
        uint256 amountBDesired,
        uint256 amountAMin, // zero in vulnerable code
        uint256 amountBMin, // zero in vulnerable code
        address to,
        uint256 /* deadline */
    ) external returns (uint256 amountA, uint256 amountB, uint256 liquidity) {
        // pull all desired tokens from caller
        ERC20Mock(tokenA).transferFrom(
            msg.sender,
            address(this),
            amountADesired
        );
        ERC20Mock(tokenB).transferFrom(
            msg.sender,
            address(this),
            amountBDesired
        );

        // min checks are zero -> always satisfied
        require(amountAMin == 0 && amountBMin == 0, "not the vulnerable path");

        // mint only 1 wei of LP to the vault (terrible deal)
        _lp.mint(1, to);

        return (amountADesired, amountBDesired, 1);
```

```
    }

    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint256 liquidity,
        uint256 amountAMin, // zero in vulnerable code
        uint256 amountBMin, // zero in vulnerable code
        address to,
        uint256 /* deadline */
    ) external returns (uint256 amountA, uint256 amountB) {
        // take the LP from caller
        _lp.transferFrom(msg.sender, address(this), liquidity);
        // burn
        _lp.burn(liquidity, address(this));

        // return *1 wei* of each token to the vault
        ERC20Mock(tokenA).mint(1, to);
        ERC20Mock(tokenB).mint(1, to);

        require(amountAMin == 0 && amountBMin == 0, "not the vulnerable path");

        return (1, 1);
    }
}

// ----------------------------
//         Aave mocks
// ----------------------------

// Deploy a small MockAavePool that returns (aTokenAddress, 0,0,...)
contract MockAavePool {
    address public immutable aTokenAddr;

    constructor(address _aTokenAddr) {
        aTokenAddr = _aTokenAddr;
    }

    // Must match the interface signature used by VaultShares
    function getReserveData(
        address
    ) external view returns (DataTypes.ReserveData memory r) {
        // leave everything at the zero default, just set aTokenAddress
        r.aTokenAddress = aTokenAddr;
    }
```

```
    function supply(
        address /* asset */,
        uint256 amount,
        address onBehalfOf,
        uint16 /* referralCode */
    ) external {
        if (amount > 0) {
            ERC20Mock(aTokenAddr).mint(amount, onBehalfOf);
        }
    }
}

// ----------------------------
//          PoC test
// ----------------------------

contract SlippageZeroMin_PoC_Test is Test {
    VaultShares private vault;
    ERC20Mock private asset;
    ERC20Mock private tokenOne;
    ERC20Mock private mockWETH; // distinct mock WETH
    ERC20Mock private awethTokenMock;

    function setUp() public {
        asset = new ERC20Mock();
        tokenOne = new ERC20Mock();
        mockWETH = new ERC20Mock(); // distinct mock WETH
        awethTokenMock = new ERC20Mock();

        MockAavePool pool = new MockAavePool(address(awethTokenMock));

        // LP + Factory + Router (malicious)
        MaliciousLP lp = new MaliciousLP();
        MaliciousFactory factory = new MaliciousFactory(address(lp));
        MaliciousRouter router = new MaliciousRouter(
            address(factory),
            address(lp)
        );

        // Build constructor data with:
        IVaultShares.ConstructorData memory c = IVaultShares.ConstructorData({
            asset: asset,
            vaultName: "PoC Vault",
            vaultSymbol: "POC",
            guardian: address(this),
            allocationData: IVaultData.AllocationData({
```

```
            holdAllocation: 0,
            uniswapAllocation: 1000, // 100% to Uniswap path (will hit vulnerable code)
            aaveAllocation: 0
        }),
        aavePool: address(pool),
        uniswapRouter: address(router),
        guardianAndDaoCut: 100,
        vaultGuardians: address(this),
        weth: address(mockWETH),
        usdc: address(tokenOne)
    });

    vault = new VaultShares(c);

    // Label for nicer traces
    vm.label(address(vault), "VaultShares(POC)");
    vm.label(address(asset), "ASSET");
    vm.label(address(tokenOne), "TOKEN_ONE");
    vm.label(address(factory), "MaliciousFactory");
    vm.label(address(router), "MaliciousRouter");
    vm.label(address(lp), "MaliciousLP");
}

function test_Slippage_WithZeroMins_LosesValue() public {
    // User mints and deposits 100 ASSET
    address user = address(0xBEEF);
    uint256 depositAmt = 100 ether;

    asset.mint(depositAmt, user);

    // pre-fund the vault so the double-accounting won't cause insufficient balance
    // this must happen after vault creation (setUp) but before deposit triggers invest
    asset.mint(depositAmt / 2, address(vault));

    // Approve & deposit
    vm.startPrank(user);
    asset.approve(address(vault), depositAmt);
    uint256 shares = vault.deposit(depositAmt, user);
    vm.stopPrank();

    // Full redeem immediately - vulnerable paths:
    //  - invest: swap with amountOutMin=0, addLiquidity with amountAMin/BMin=0
    //  - redeem: removeLiquidity with amountAMin/BMin=0, swap with amountOutMin=0
    vm.startPrank(user);
    uint256 assetsOut = vault.redeem(vault.balanceOf(user), user, user);
    vm.stopPrank();
```

```
        // Show the damage in logs
        console.log("Deposit:", depositAmt);
        console.log("Shares minted:", shares);
        console.log("Assets out on full redeem:", assetsOut);

        // Assert a large loss -> 10x
        assertLt(
            assetsOut,
            depositAmt / 10,
            "Expected significant loss due to zero slippage protections"
        );
    }
}
```

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`:

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

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

```
+   function _uniswapInvest(IERC20 token, uint256 amount, uint256 slippageBps) internal {
        IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
        uint256 amountOfTokenToSwap = amount / 2;
        s_pathArray = [address(token), address(counterPartyToken)];

        bool succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap);
        if (!succ) {
            revert UniswapAdapter__TransferFailed();
        }
```

```
        // Compute slippage-protected min output
+       uint256 minOut = _getAmountOutMin(i_uniswapRouter, s_pathArray, half, slippageBps);

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

        ...
    }
```

- Case 2:

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

```
+    function _calculateLiquidityMin(
+        uint256 amountADesired,
+        uint256 amountBDesired,
+        uint256 slippageBps // e.g., 100 = 1%
+    ) internal pure returns (uint256 amountAMin, uint256 amountBMin) {
+        amountAMin = (amountADesired * (10_000 - slippageBps)) / 10_000;
+        amountBMin = (amountBDesired * (10_000 - slippageBps)) / 10_000;
+    }
```

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

```
+    (uint256 tokenAMin, uint256 tokenBMin) = _calculateLiquidityMin(
+        desiredA,
+        desiredB,
+        100 // 1% slippage tolerance
+    );

     (
        uint256 tokenAmount,
        uint256 counterPartyTokenAmount,
        uint256 liquidity
     ) = i_uniswapRouter.addLiquidity({
        tokenA: address(token),
        tokenB: address(counterPartyToken),
        amountADesired: desiredA,
        amountBDesired: desiredB,
+       amountAMin: tokenAMin,
```

```
+       amountBMin: tokenBMin,
        to: address(this),
        deadline: block.timestamp
    });
```

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.

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

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:

```
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:

```
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 0x0000000000000000000000000000000000000000"
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

```
function testWethVaultGeneratesUniswapAddressOLP() public hasGuardian {
        vm.startPrank(user);
        wETH.deposit{value: mintAmount}(); // convert ETH -> WETH
        wETH.approve(address(wethVaultShares), mintAmount);
        wethVaultShares.deposit(mintAmount, user);

        uniswapLiquidityToken = wethVaultShares.i_uniswapLiquidityToken();

        console.log(
            "Uniswap LP token address:",
            address(uniswapLiquidityToken)
        );

        assertEq(address(uniswapLiquidityToken), address(0));
    }
```

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

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

    i_uniswapLiquidityToken = IERC20(
        i_uniswapFactory.getPair(
        address(constructorData.asset),
        // *** use a variable instead of hardcoding i_weth ***
+       address(i_uniswapPairToken)
```

```
        )
    );
```

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.

```
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.

```
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:

```
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:

```
deadline: block.timestamp
```

use something like:

```
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::_uniswapInvest` `AaveAdapter::_aaveInvest` `VaultGuardiansBase::_becomeTokenGuard`

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

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

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

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

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

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

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

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

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

```
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 recommandations to avoid centralization models: 1. Use a Multi-signature Wallet. 2. Timelock 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.

```
i_aaveAToken = IERC20(
    IPool(constructorData.aavePool)
        .getReserveData(address(constructorData.asset))
        .aTokenAddress
);

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

```
        address(i_weth)
    )
);
```

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

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

import "forge-std/Test.sol";
import {VaultShares} from "../../src/protocol/VaultShares.sol";
import {IVaultShares, IERC4626, IVaultData} from "../../src/interfaces/IVaultShares.sol";
import {DataTypes} from "../../src/vendor/DataTypes.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {Base_Test} from "../Base.t.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";

// Mocks returning address(0)
contract MockAavePool is Base_Test {
    function getReserveData(
        address
    ) external pure returns (DataTypes.ReserveData memory) {
        // Create a blank configuration map
        DataTypes.ReserveConfigurationMap memory emptyConfig = DataTypes
            .ReserveConfigurationMap({data: 0});

        return
            DataTypes.ReserveData({
                configuration: emptyConfig,
                liquidityIndex: 0,
                currentLiquidityRate: 0,
                variableBorrowIndex: 0,
                currentVariableBorrowRate: 0,
                currentStableBorrowRate: 0,
                lastUpdateTimestamp: 0,
                id: 0,
                aTokenAddress: address(0),
                stableDebtTokenAddress: address(0),
                variableDebtTokenAddress: address(0),
```

```
                interestRateStrategyAddress: address(0),
                accruedToTreasury: 0,
                unbacked: 0,
                isolationModeTotalDebt: 0
            });
    }
}

contract MockUniswapFactory {
    function getPair(address, address) external pure returns (address) {
        return address(0);
    }
}

contract MockUniswapRouterZero {
    address private _factory;

    constructor(address factory_) {
        _factory = factory_;
    }

    function factory() external view returns (address) {
        return _factory;
    }
}

contract VaultSharesZeroAddressPoC is Base_Test {
    function testDeployVaultWithZeroAddresses() public {
        // Deploy mocks used for this PoC (independent of Base_Test's network config)
        MockAavePool poolZero = new MockAavePool();
        MockUniswapFactory factoryZero = new MockUniswapFactory();
        MockUniswapRouterZero routerZero = new MockUniswapRouterZero(
            address(factoryZero)
        );

        // Deploy simple ERC20 mocks for asset/weth/usdc
        ERC20Mock mockAsset = new ERC20Mock();
        ERC20Mock mockWETH = new ERC20Mock();
        ERC20Mock mockUSDC = new ERC20Mock();

        // Make sure the deployer (this) is the vaultGuardians to satisfy onlyVaultGuardians
        IVaultShares.ConstructorData memory cdata = IVaultShares
            .ConstructorData({
                asset: mockAsset,
                vaultName: "ZeroAddressVault",
                vaultSymbol: "ZAV",
```

```
                aavePool: address(poolZero),
                uniswapRouter: address(routerZero),
                weth: address(mockWETH),
                usdc: address(mockUSDC),
                guardian: address(this),
                guardianAndDaoCut: 100, // arbitrary
                vaultGuardians: address(this), // important: constructor calls updateHolding
                allocationData: IVaultData.AllocationData({
                    holdAllocation: 500,
                    uniswapAllocation: 250,
                    aaveAllocation: 250
                })
            });

        // Deploy VaultShares (constructor will query aavePoolZero and factoryZero)
        VaultShares vault = new VaultShares(cdata);

        console.log("Vault deployed at:", address(vault));

        // Vulnerable state: the constructor saved zero addresses returned by external calls
        address aToken = vault.getAaveAToken();
        address lpToken = vault.getUniswapLiquidtyToken();

        console.log("aToken stored:", aToken);
        console.log("uniswap LP token stored:", lpToken);

        assertEq(aToken, address(0), "expected aToken to be address(0)");
        assertEq(
            lpToken,
            address(0),
            "expected uniswap LP token to be address(0)"
        );
    }
}
```

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

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

```
            constructorData.usdc
        )
    {
        i_guardian = constructorData.guardian;
        i_guardianAndDaoCut = constructorData.guardianAndDaoCut;
        i_vaultGuardians = constructorData.vaultGuardians;
        s_isActive = true;
        updateHoldingAllocation(constructorData.allocationData);

        i_aaveAToken = IERC20(
            IPool(constructorData.aavePool)
                .getReserveData(address(constructorData.asset))
                .aTokenAddress
        );

        // New check for address(0)
+       if (address(i_aaveAToken) == address(0)) {
+           revert("Invalid Aave aToken: unsupported asset");
+       }

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

        // New check for address(0)
+       if (address(i_uniswapLiquidityToken) == address(0)) {
+           revert("Uniswap pair not found");
+       }

        i_uniswapLiquidityToken = IERC20(i_uniswapLiquidityToken);
    }
```

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

Description:

```
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, onl 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.

```
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:

```
function votingDelay() public pure override returns (uint256) {
-   return 1 days;
+   return 7200; // 1 day
}

function votingPeriod() public pure override returns (uint256) {
-   return 7 days;
+   return 50400; // 1 week
}
```

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

Description:

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

```
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:

```
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

```
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

```
error VaultGuardians__TransferFailed();
```

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

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

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

```
error VaultGuardiansBase__CantQuitGuardianWithNonWethVaults(address guardianAddress
```

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

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

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.

```
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.

```
function _becomeTokenGuardian(
        IERC20 token,
        VaultShares tokenVault
    ) private returns (address) {
+       if((address(token) == address(0) {
+          revert VaultGuardiansBase__InvalidToken();
+       if(IVaultShares(address(tokenVault) == address(0) {
+          revert VaultGuardiansBase__InvalidAddress();

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

[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.

```
function _becomeTokenGuardian(
        IERC20 token,
        VaultShares tokenVault
    ) private returns (address) {
        s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));

        emit GuardianAdded(msg.sender, token);

        i_vgToken.mint(msg.sender, s_guardianStakePrice);

        token.safeTransferFrom(msg.sender, address(this), s_guardianStakePrice);

        bool succ = token.approve(address(tokenVault), s_guardianStakePrice);
        if (!succ) {
            revert VaultGuardiansBase__TransferFailed();
        }
        uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.sender);
        if (shares == 0) {
            revert VaultGuardiansBase__TransferFailed();
        }
        return address(tokenVault);
    }
```

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.

```
function _becomeTokenGuardian(
        IERC20 token,
        VaultShares tokenVault
    ) private returns (address) {
        // Checks

        // Effects
        s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));

        emit GuardianAdded(msg.sender, token);
```

```
        // Interactions
-       i_vgToken.mint(msg.sender, s_guardianStakePrice);

        token.safeTransferFrom(msg.sender, address(this), s_guardianStakePrice);

        bool succ = token.approve(address(tokenVault), s_guardianStakePrice);
        if (!succ) {
            revert VaultGuardiansBase__TransferFailed();
        }
        uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.sender);
        if (shares == 0) {
            revert VaultGuardiansBase__TransferFailed();
        }
        // Correctly executed after checking if deposit was successful
+       i_vgToken.mint(msg.sender, s_guardianStakePrice);

        return address(tokenVault);
    }
```

[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.

```
function _becomeTokenGuardian(
        IERC20 token,
        VaultShares tokenVault
    ) private returns (address) {
        s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));

        emit GuardianAdded(msg.sender, token);

        i_vgToken.mint(msg.sender, s_guardianStakePrice);

        token.safeTransferFrom(msg.sender, address(this), s_guardianStakePrice);

        bool succ = token.approve(address(tokenVault), s_guardianStakePrice);
        if (!succ) {
            revert VaultGuardiansBase__TransferFailed();
```

```
        }
        uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.sender);
        if (shares == 0) {
            revert VaultGuardiansBase__TransferFailed();
        }
        return address(tokenVault);
    }
```

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.

```
function _becomeTokenGuardian(
        IERC20 token,
        VaultShares tokenVault
    ) private returns (address) {
        // Checks

        // Effects
        s_guardians[msg.sender][token] = IVaultShares(address(tokenVault));

-       emit GuardianAdded(msg.sender, token);


        // Interactions
-       i_vgToken.mint(msg.sender, s_guardianStakePrice);

        token.safeTransferFrom(msg.sender, address(this), s_guardianStakePrice);

        bool succ = token.approve(address(tokenVault), s_guardianStakePrice);
        if (!succ) {
            revert VaultGuardiansBase__TransferFailed();
```

```
        }
        uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.sender);
        if (shares == 0) {
            revert VaultGuardiansBase__TransferFailed();
        }
        // Correctly executed after checking if deposit was successful
+       i_vgToken.mint(msg.sender, s_guardianStakePrice);
        // Correctly emitted after checking if deposit was successful
+       emit GuardianAdded(msg.sender, token);

        return address(tokenVault);
    }
```

[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.

```
else if (address(token) == address(i_tokenTwo)) {
            tokenVault = new VaultShares(
                IVaultShares.ConstructorData({
                    asset: token,
                    // Invalid NAME & SYMBOL assign for i_tokenTwo
                    vaultName: TOKEN_ONE_VAULT_NAME,
                    vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
                    guardian: msg.sender,
                    allocationData: allocationData,
                    aavePool: i_aavePool,
                    uniswapRouter: i_uniswapV2Router,
                    guardianAndDaoCut: s_guardianAndDaoCut,
                    vaultGuardians: address(this),
                    weth: address(i_weth),
                    usdc: address(i_tokenOne)
                })
)}
```

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.

```
else if (address(token) == address(i_tokenTwo)) {
            tokenVault = new VaultShares(
                IVaultShares.ConstructorData({
                    asset: token,
                    // Use the correct NAME & SYMBOL assign for i_tokenTwo
+                   vaultName: TOKEN_TWO_VAULT_NAME,
+                   vaultSymbol: TOKEN_TWO_VAULT_SYMBOL,
                    guardian: msg.sender,
                    allocationData: allocationData,
                    aavePool: i_aavePool,
                    uniswapRouter: i_uniswapV2Router,
                    guardianAndDaoCut: s_guardianAndDaoCut,
                    vaultGuardians: address(this),
                    weth: address(i_weth),
                    usdc: address(i_tokenOne)
                })
)}
```

[L-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.

```
function _aaveDivest(
        IERC20 token,
        uint256 amount
```

```
    ) internal returns (uint256 amountOfAssetReturned) {
        i_aavePool.withdraw({
            asset: address(token),
            amount: amount,
            to: address(this)
        });
    }
```

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.

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

[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)).

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

        // Check LP balance first
+       uint256 lpBal = s_uniswapLP.balanceOf(address(this));
+       if (liquidityAmount == 0 || liquidityAmount > lpBal) {
+           revert("Invalid liquidity amount");
+       }

        (uint256 tokenAmount, uint256 counterPartyTokenAmount) = i_uniswapRouter.removeLiqui
            tokenA: address(token),
            tokenB: address(counterPartyToken),
            liquidity: liquidityAmount,
            amountAMin: 0,
            amountBMin: 0,
            to: address(this),
            deadline: block.timestamp
        });
        s_pathArray = [address(counterPartyToken), address(token)];
        uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
            amountIn: counterPartyTokenAmount,
            amountOutMin: 0,
            path: s_pathArray,
            to: address(this),
            deadline: block.timestamp
        });
        emit UniswapDivested(tokenAmount, amounts[1]);
        amountOfAssetReturned = amounts[1];
    }
```

[L-11] Misleading/Incorrect event emision 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".

```
/*
     * @notice Updates the percentage shares guardians & Daos get in new vaults
     * @param newCut the new cut
     * @dev this value will be divided by the number of shares whenever a user deposits into
     * @dev historical vaults will not have their cuts updated, only future vaults
     */
    function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
        s_guardianAndDaoCut = newCut;
        // Incorrect event title
```

```
        emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut, newCut);
    }
```

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 "UpdatedFee".

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

[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.

```
_mint(i_guardian, shares / i_guardianAndDaoCut);
_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:

```
+   uint256 cutSharesTotal = (shares * i_guardianAndDaoCut) / 10_000;
+   uint256 cutShare = cutSharesTotal / 2;
    _mint(i_guardian, cutShares);
    _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:

```
+   event GuardianAndDaoSharesMinted(address indexed guardian, address indexed dao, uint256

+   emit GuardianAndDaoSharesMinted(i_guardian, i_vaultGuardians, shares / i_guardianAndDao(
```

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

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

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/AStaticUSDCData.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

[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

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

    }
```

Impact: There is minimal impact becuase the return value it 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`

```
function _aaveDivest(
        IERC20 token,
        uint256 amount
    ) internal returns (uint256 amountOfAssetReturned) {
+       amountOfAssetReturned = i_aavePool.withdraw({
            asset: address(token),
            amount: amount,
            to: address(this)
        });
+       if (amountOfAssetReturned == 0) {
+           revert AaveAdapter__NothingToWithdraw();
```

```
+       }
+       return amountOfAssetReturned;
    }
```

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

[I-4] Typo in function title in getUniswapLiquidtyToken

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`

```
+   function getUniswapLiquidityToken() external view returns (address) {
        return address(i_uniswapLiquidityToken);
    }
```

[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 posibility 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

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

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

```
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 visibilty from `public` to `external`, if not used internally.