Protocol Audit Report

Version 1.0

Vesko.io

October 18, 2024

VAULT-GUARDIANS Audit Report

Vesko.io

October 18, 2024

Prepared by: Vesko Lead Auditors: - Veselin Vachkov

Table of Contents

- Table of Contents
- Disclaimer
- Risk Classification
  - Audit Details
  - Scope
- Protocol Summary
  - Roles
- Executive Summary
  - Issues found
  - Findings
  - High
    * [H-1] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables frontrunners to steal profits
    * [H-2] `ERC4626::totalAssets` checks the balance of vault's underlying asset even when the asset is invested, resulting in incorrect values being returned
    * [H-3] Guardians can infinitely mint `VaultGuardianToken`s and take over DAO, stealing DAO fees and maliciously setting parameters
    * [H-4] Centralized Control of `VaultGuardianToken` and `VaultGuardians` Through Ownable

    * [H-5] Incorrect Emission of Old Stake Price in `VaultGuardians:updateGuardianStakePrice` Function
    * [H-6] Lack of Bound Checks and Incorrect Event Emission in `VaultGuardians:updateGuardianAndDaoCut` Function
- Medium
    * [M-1] Potentially incorrect voting period and delay in governor may affect governance
- Low
    * [L-1] Incorrect vault name and symbol
    * [L-2] Unassigned return value when divesting AAVE funds
- [L-3]: `public` functions not used internally could be marked `external`
- Informational
    * [I-1] Unused Import and Interface
    * [I-2] `nonReentrant` modifier should be placed before all other modifiers
    * [I-3] Return values are not checked
    * [I-4] Unused Errors in `VaultGuardiansBase` and `VaultGuardians` Contracts
- Gas
    * [G-1] Unused CONSTANT, also can be made immutable to save on gas in `VaultGuardiansBase`
    * [G-2] Non-Pure Function Declaration in the `UniswapV2Router01` Interface

Disclaimer

The Veselin's team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M

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

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

Audit Details

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
```

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.

Roles

There are 4 main roles associated with the system.

- *Vault Guardian DAO*: The org that takes a cut of all profits, controlled by the `VaultGuardianToken`. The DAO that controls a few variables of the protocol, including:
  - `s_guardianStakePrice`
  - `s_guardianAndDaoCut`
  - And takes a cut of the ERC20s made from the protocol
- *DAO Participants*: Holders of the `VaultGuardianToken` who vote and take profits on the protocol
- *Vault Guardians*: Strategists/hedge fund managers who have the ability to move assets in and out of the investable universe. They take a cut of revenue from the protocol.
- *Investors*: The users of the protocol. They deposit assets to gain yield from the investments of the Vault Guardians.

Executive Summary

The Vault Guardians project takes novel approaches to work ERC-4626 into a hedge fund of sorts, but makes some large mistakes on tracking balances and profits.

Issues found

Severity	Number of issues found
High	6
Medium	1
Low	3
Info	4
Total	2

Findings

High

[H-1] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables frontrunners to steal profits

Description: In `UniswapAdapter::_uniswapInvest` the protocol swaps half of an ERC20 token so that they can invest in both sides of a Uniswap pool. It calls the `swapExactTokensForTokens` function of the `UnisapV2Router01` contract , which has two input parameters to note:

```
1      function swapExactTokensForTokens(
2          uint256 amountIn,
3  @>      uint256 amountOutMin,
4          address[] calldata path,
5          address to,
6  @>      uint256 deadline
7      )
```

The parameter `amountOutMin` represents how much of the minimum number of tokens it expects to return. The `deadline` parameter represents when the transaction should expire.

As seen below, the `UniswapAdapter::_uniswapInvest` function sets those parameters to `0` and `block.timestamp`:

```
1      uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens
           (
2          amountOfTokenToSwap,
3  @>      0,
4          s_pathArray,
5          address(this),
6  @>      block.timestamp
7      );
```

Impact: This results in either of the following happening: - Anyone (e.g., a frontrunning bot) sees this transaction in the mempool, pulls a flashloan and swaps on Uniswap to tank the price before the swap happens, resulting in the protocol executing the swap at an unfavorable rate. - Due to the lack of a deadline, the node who gets this transaction could hold the transaction until they are able to profit from the guaranteed swap.

Proof of Concept:

1. User calls `VaultShares::deposit` with a vault that has a Uniswap allocation.
    1. This calls `_uniswapInvest` for a user to invest into Uniswap, and calls the router's `swapExactTokensForTokens` function.

2. In the mempool, a malicious user could:
    1. Hold onto this transaction which makes the Uniswap swap
    2. Take a flashloan out
    3. Make a major swap on Uniswap, greatly changing the price of the assets
    4. Execute the transaction that was being held, giving the protocol as little funds back as possible due to the `amountOutMin` value set to 0.

This could potentially allow malicious MEV users and frontrunners to drain balances.

Recommended Mitigation:

For the deadline issue, we recommend the following:

DeFi is a large landscape. For protocols that have sensitive investing parameters, add a custom parameter to the `deposit` function so the Vault Guardians protocol can account for the customizations of DeFi projects that it integrates with.

In the `deposit` function, consider allowing for custom data.

```
- function deposit(uint256 assets, address receiver) public override(
    ERC4626, IERC4626) isActive returns (uint256) {
+ function deposit(uint256 assets, address receiver, bytes customData)
    public override(ERC4626, IERC4626) isActive returns (uint256) {
```

This way, you could add a `deadline` to the Uniswap swap, and also allow for more DeFi custom integrations.

For the `amountOutMin` issue, we recommend one of the following:

1. Do a price check on something like a Chainlink price feed before making the swap, reverting if the rate is too unfavorable.
2. Only deposit 1 side of a Uniswap pool for liquidity. Don't make the swap at all. If a pool doesn't exist or has too low liquidity for a pair of ERC20s, don't allow investment in that pool.

Note that these recommendation require significant changes to the codebase.

[H-2] `ERC4626::totalAssets` checks the balance of vault's underlying asset even when the asset is invested, resulting in incorrect values being returned

Description: The `ERC4626::totalAssets` function checks the balance of the underlying asset for the vault using the `balanceOf` function.

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

However, the assets are invested in the investable universe (Aave and Uniswap) which means this will never return the correct value of assets in the vault.

Impact: This breaks many functions of the `ERC4626` contract: - `totalAssets` - `convertToShares` - `convertToAssets` - `previewWithdraw` - `withdraw` - `deposit`

All calculations that depend on the number of assets in the protocol would be flawed, severely disrupting the protocol functionality.

Proof of Concept:

Code

Add the following code to the `VaultSharesTest.t.sol` file.

```
function testWrongBalance() public {
    // Mint 100 ETH
    weth.mint(mintAmount, guardian);
    vm.startPrank(guardian);
    weth.approve(address(vaultGuardians), mintAmount);
    address wethVault = vaultGuardians.becomeGuardian(allocationData);
    wethVaultShares = VaultShares(wethVault);
    vm.stopPrank();

    // prints 3.75 ETH
    console.log(wethVaultShares.totalAssets());

    // Mint another 100 ETH
    weth.mint(mintAmount, user);
    vm.startPrank(user);
    weth.approve(address(wethVaultShares), mintAmount);
    wethVaultShares.deposit(mintAmount, user);
    vm.stopPrank();

    // prints 41.25 ETH
    console.log(wethVaultShares.totalAssets());
}
```

Recommended Mitigation: Do not use the OpenZeppelin implementation of the `ERC4626` contract. Instead, natively keep track of users total amounts sent to each protocol. Potentially have an automation tool or some incentivised mechanism to keep track of protocol's profits and losses, and take snapshots of the investable universe.

This would take a considerable re-write of the protocol.

[H-3] Guardians can infinitely mint `VaultGuardianTokens` and take over DAO, stealing DAO fees and maliciously setting parameters

Description: Becoming a guardian comes with the perk of getting minted Vault Guardian Tokens (vgTokens). Whenever a guardian successfully calls `VaultGuardiansBase::becomeGuardian` or `VaultGuardiansBase::becomeTokenGuardian`, `_becomeTokenGuardian` is executed, which mints the caller `i_vgToken`.

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

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

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

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

Proof of Concept:

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

Code

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

```
1      function testDaoTakeover() public hasGuardian hasTokenGuardian {
2          address maliciousGuardian = makeAddr("maliciousGuardian");
```

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

Recommended Mitigation: There are a few options to fix this issue:

1. Mint vgTokens on a vesting schedule after a user becomes a guardian.
2. Burn vgTokens when a guardian quits.
3. Simply don't allocate vgTokens to guardians. Instead, mint the total supply on contract deployment.

[H-4] Centralized Control of `VaultGuardianToken` and `VaultGuardians` Through Ownable

Description: The `VaultGuardianToken` contract inherits from `Ownable`, which grants ownership to a single address (the contract deployer). This centralization creates a significant risk in the context of a Decentralized Autonomous Organization (DAO), where the intention is to distribute control among token holders rather than concentrating it in a single entity. The mint function, restricted to the owner, enables the owner to arbitrarily inflate the supply of `VaultGuardianToken` (VGT), undermining the trust and value of the token.

```
1   import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
2   contract VaultGuardianToken is ERC20, ERC20Permit, ERC20Votes,
       Ownable{...}
```

Impact: If a malicious owner (or an exploited owner) were to mint an excessive amount of VGT, they could significantly influence the governance decisions made within the DAO. This could lead to the following issues:

- Token Inflation: The value of existing tokens would diminish as the total supply increases without a corresponding increase in demand. Governance Manipulation: The owner could gain undue influence over the DAO's governance by accumulating voting power through minted tokens, effectively negating the decentralized nature of the organization.

Proof of Concept: - The deployer (owner) of the `VaultGuardianToken` contract calls the mint function to create an arbitrary amount of VGT. The owner then votes on governance proposals, potentially swaying decisions in favor of their interests. The overall integrity of the DAO is compromised due to centralized control.

Recommended Mitigation: To align the VaultGuardianToken contract with the principles of decentralization and secure governance, consider the following options: - Remove the `Ownable` inheritance: This would eliminate centralized control over the minting function, ensuring that all VGT holders have a say in governance. - Implement a `multi-signature wallet`: If ownership is necessary, consider using a multi-signature wallet to require consensus among multiple parties before minting tokens. - Introduce a decentralized minting mechanism: Allow VGT to be minted based on community proposals or through mechanisms such as staking, rather than solely at the discretion of the owner.

[H-5] Incorrect Emission of Old Stake Price in `VaultGuardians:updateGuardianStakePrice` Function

Description: The `updateGuardianStakePrice` function emits an event that records the old stake price, but it uses the updated value of `s_guardianStakePrice` instead of the previous value. As a result, the event emits the new value for both the old and new stake prices, which leads to misleading information being logged.

Impact: - Misleading Event Logs: Users and other contracts that rely on the emitted events for tracking state changes will receive incorrect information regarding the stake price history, which can lead to confusion and erroneous assumptions about the contract state.

- Potential Exploits: If other functions or external actors rely on the historical values of stake prices for decision-making, this inconsistency could potentially be exploited to gain an advantage or manipulate outcomes.

Proof of Concept: To illustrate the issue, consider the following sequence of events: - The initial stake price is set to 100. - The owner calls updateGuardianStakePrice(150). - The emitted event shows both values as 150, misleading observers to think that the previous value was 150 rather than the correct 100.

Recommended Mitigation:

```
function updateGuardianStakePrice(uint256 newStakePrice) external
    onlyOwner {
     uint256 oldStakePrice = s_guardianStakePrice;  // Store the old
        stake price
     s_guardianStakePrice = newStakePrice;
     emit VaultGuardians__UpdatedStakePrice(oldStakePrice, newStakePrice
        ); // Emit the correct old stake price
}
```

[H-6] Lack of Bound Checks and Incorrect Event Emission in `VaultGuardians:updateGuardianAndDaoCut` Function

Description: The `updateGuardianAndDaoCut()` function allows the contract owner to set a new cut value that impacts the distribution of rewards between the guardians and the DAO. However, there are two significant issues with the function.

Impact: - Potential for Invalid or Extreme Values: Without range checks, the owner could set values for ‘`s_guardianAndDaoCut` that are unrealistic, such as 0% or 100%, which could undermine the contract’s fairness and disrupt the reward distribution mechanism.

- Incorrect Emission of Old Cut Value: Similar to the issue found in the `updateGuardianStakePrice` function, the `updateGuardianAndDaoCut()` function uses the new value of `s_guardianAndDaoCut` when emitting the event. As a result, the event logs the new value twice instead of the old and new values, leading to inaccurate tracking of changes.

Proof of Concept: - The owner sets `s_guardianAndDaoCut` to an extreme value like `100000` without any restrictions, which could lead to skewed reward distribution. - When calling the function, the emitted event shows the new cut value twice, rather than logging the correct old value.

Recommended Mitigation:

```
function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
    require(newCut <= 100 && newCut >= 0, "Cut value must be between 0 and 100"); // Ensure the value is within bounds
    uint256 oldCut = s_guardianAndDaoCut;  // Store the old cut value
    s_guardianAndDaoCut = newCut;
    emit VaultGuardians__UpdatedGuardianAndDaoCut(oldCut, newCut); // Emit the correct old and new values
}
```

Medium

[M-1] 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.

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

```
9 }
```

Low

[L-1] Incorrect vault name and symbol

When new vaults are deployed in the `VaultGuardianBase::becomeTokenGuardian` function, symbol and vault name are set incorrectly when the `token` is equal to `i_tokenTwo`. Consider modifying the function as follows, to avoid errors in off-chain clients reading these values to identify vaults.

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

Also, add a new test in the `VaultGuardiansBaseTest.t.sol` file to avoid reintroducing this error, similar to what's done in the test `testBecomeTokenGuardianTokenOneName`.

[L-2] Unassigned return value when divesting AAVE funds

The `AaveAdapter::_aaveDivest` function is intended to return the amount of assets returned by AAVE after calling its `withdraw` function. However, the code never assigns a value to the named return variable `amountOfAssetReturned`. As a result, it will always return zero.

While this return value is not being used anywhere in the code, it may cause problems in future changes. Therefore, update the `_aaveDivest` function as follows:

```
function _aaveDivest(IERC20 token, uint256 amount) internal returns (
    uint256 amountOfAssetReturned) {
-       i_aavePool.withdraw({
```

```
+       amountOfAssetReturned = i_aavePool.withdraw({
            asset: address(token),
            amount: amount,
            to: address(this)
        });
}
```

[L-3]: `public` functions not used internally could be marked `external`

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

9 Found Instances

- Found in src/dao/VaultGuardianGovernor.sol Line: 17

```
    function votingDelay() public pure override returns (uint256)
        {
```

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

```
    function votingPeriod() public pure override returns (uint256)
        {
```

- Found in src/dao/VaultGuardianGovernor.sol Line: 27

```
    function quorum(uint256 blockNumber)
```

- Found in src/dao/VaultGuardianToken.sol Line: 17

```
    function nonces(address ownerOfNonce) public view override(
        ERC20Permit, Nonces) returns (uint256) {
```

- Found in src/protocol/VaultShares.sol Line: 116

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

- Found in src/protocol/VaultShares.sol Line: 141

```
    function deposit(uint256 assets, address receiver)
```

- Found in src/protocol/VaultShares.sol Line: 182

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

- Found in src/protocol/VaultShares.sol Line: 190

```
1     function withdraw(uint256 assets, address receiver, address
          owner)
```

- Found in src/protocol/VaultShares.sol Line: 207

```
1     function redeem(uint256 shares, address receiver, address
          owner)
```

Informational

[I-1] Unused Import and Interface

Description: The contract contains an unused import statement and an interface that are not utilized in the codebase. - `VaultShares solidit : import {DataTypes} from "../vendor/DataTypes.sol";` -

```
1 interface IVaultGuardians {}
```

```
1 interface IVaultGuardians {}
```

```
1 import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
2
3 interface IInvestableUniverseAdapter {}
```

```
1 interface IVaultGuardians {}
```

Impact: - Code Bloat: Unused imports and interfaces increase the size of the code unnecessarily, making it harder to read and maintain. - Potential Confusion: Developers may be misled into thinking these components are required for the contract's functionality, leading to confusion during future development.

Recommended Mitigation: - Remove Unused Imports and Interfaces: Eliminate any import statements and interfaces that are not utilized within the contract to enhance code clarity and maintainability.

[I-2] `nonReentrant` modifier should be placed before all other modifiers

Description: The `nonReentrant` modifier should be placed before all other modifiers in `VaultShares` contract, in `deposit`, `rebalanceFunds`, and `withdraw` functions. This is a best practice to ensure protection against reentrancy attacks, especially when other modifiers might alter the state or call external contracts.

Impact: - Potential reentrancy attack.

Recommended Mitigation: - Move the nonReentrant modifier to the beginning of the modifier list in all affected functions

[I-3] Return values are not checked

Description: The return values from the divestment functions `VaultShares:_uniswapDivest` and `VaultShares:_aaveDivest` are not checked.

Impact: If these functions fail or return unexpected values, the caller may remain unaware, potentially leading to unhandled exceptions or inconsistent state.

Recommended Mitigation: Use require statements to validate that divestments are successful before proceeding to the reinvestment step.

```
1 if (uniswapLiquidityTokensBalance > 0) {
2   require(amountUniswapDivested_uniswapDivest(IERC20(asset()),
       uniswapLiquidityTokensBalance) > 0, "Uniswap divestment failed or
       returned 0");
3 }
4 if (aaveAtokensBalance > 0) {
5   require(_aaveDivest(IERC20(asset()), aaveAtokensBalance) > 0, "Aave
       divestment failed or returned 0");
6 }
```

[I-4] Unused Errors in `VaultGuardiansBase` and `VaultGuardians` Contracts

Description: The `VaultGuardiansBase` contract contains several error definitions that are not utilized in the contract logic. Specifically, the following errors are defined but never thrown:

- VaultGuardiansBase:VaultGuardiansBase__NotEnoughWeth(uint256 amount, uint256 amount-Needed);
- VaultGuardiansBase:VaultGuardiansBase__NotAGuardian(address guardianAddress, IERC20 token);
- VaultGuardiansBase:VaultGuardiansBase__CantQuitGuardianWithNonWethVaults(address guardianAddress);
- VaultGuardiansBase:VaultGuardiansBase__FeeTooSmall(uint256 fee, uint256 requiredFee);
- VaultGuardians:VaultGuardians__TransferFailed();

These errors suggest potential failure conditions that are either unaccounted for in the contract logic or not adequately enforced, which could lead to unexpected behaviors during execution.

Impact: Leaving error checks unused could result in vulnerabilities, as the contract may proceed with operations that should be restricted. For instance:

- Security Risks: Unchecked conditions can lead to unauthorized actions or state changes that compromise the integrity of the contract.

Recommended Mitigation: Refactor Logic: If certain error checks are no longer relevant, remove them to maintain clean and efficient code.

Gas

[G-1] Unused CONSTANT, also can be made immutable to save on gas in `VaultGuardiansBase`

Description:

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

[G-2] Non-Pure Function Declaration in the `UniswapV2Router01` Interface

Description: The `IUniswapV2Router01` interface includes functions that are declared as view but could be defined as pure. This includes functions like `IUniswapV2Router01:factory()` and `IUniswapV2Router01:WETH()`.

Impact: - Function Overhead: Declaring functions as view when they could be pure may lead to unnecessary overhead in terms of gas costs, as view functions can still access blockchain state. - Clarity: Using pure clearly indicates that the function does not read or modify the state, making the code more intuitive for developers.

Recommended Mitigation: - Change View to Pure: Update the declarations of `IUniswapV2Router01:factory()` and `IUniswapV2Router01:WETH()` to pure.