
Prepared by: [PULIN] Lead Auditors:

- ONLY MYSELF !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Table of Contents

- Table of Contents
- Disclaimer
- Risk Classification
- Audit Details
 - Scope
- Protocol Summary
 - Roles
- Executive Summary
 - Issues Found
 - 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 `VaultGuardianTokens` and take over DAO, stealing DAO fees and maliciously setting parameters
 - * [H-4] In the constructor of `vaultshares`, the liquidityToken of Uniswap is obtained by calling the `getPair` function of its factory contract, but it hardcodes WETH and asset.
 - * [H-5] An attacker can impersonate the receiver and take all assets of a staker at `VaultShares`.
 - * [H-6] In the `addLiquidity` function of the adapter, an investment error in assets caused Uniswap to occupy an additional 25%, i.e., 37.5%, in the entire trading strategy.
 - * [H-7] The `VaultGuardianToken` has no burning function, meaning it cannot be destroyed. Even if the fund manager exits the project, they will still retain voting rights.
 - * [H-8] The `vaultshares` contract is one-time use. When a guardian exits the project, the contract is set to inactive. However, if the next fund manager wants to participate, there is no function to reactivate the contract.
 - 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
 - Info
 - * [I-1] The `IInvestableUniverseAdapter` interface has never been used.
 - * [I-2] The variable `GUARDIAN_FEE` has never been used.

Disclaimer

The YOUR_NAME_HERE 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

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

1 XXXX

Scope

```
1 ./src/
2 #-- abstract
3 |   #-- AStaticTokenData.sol
4 |   #-- AStaticUSDCData.sol
5 |   #-- AStaticWethData.sol
6 #-- dao
7 |   #-- VaultGuardianGovernor.sol
8 |   #-- VaultGuardianToken.sol
9 #-- interfaces
10 |  #-- IVaultData.sol
11 |  #-- IVaultGuardians.sol
12 |  #-- IVaultShares.sol
13 |  #-- InvestableUniverseAdapter.sol
14 #-- protocol
15 |  #-- VaultGuardians.sol
16 |  #-- VaultGuardiansBase.sol
17 |  #-- VaultShares.sol
18 |  #-- investableUniverseAdapters
19 |      #-- AaveAdapter.sol
20 |      #-- UniswapAdapter.sol
21 #-- vendor
22 |  #-- DataTypes.sol
23 |  #-- IPool.sol
24 |  #-- IUniswapV2Factory.sol
25 |  #-- IUniswapV2Router01.sol
```

Protocol Summary

This protocol allows users to deposit certain ERC20s into an ERC4626 vault managed by a human being, or a `vaultGuardian`. The goal of a `vaultGuardian` is to manage the vault in a way that maximizes the value of the vault for the users who have deposited money into the vault.

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	8
Medium	1
Low	2
Info	2
Gas	0
Total	13

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 `UniswapV2Router01` 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         (  
3 @>     amountOfTokenToSwap,  
4         0,  
5         s_pathArray,  
6 @>     address(this),  
7         block.timestamp  
8     );
```

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.

```
1 - function deposit(uint256 assets, address receiver) public override(  
    ERC4626, IERC4626) isActive returns (uint256) {  
2 + 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.

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

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.

```
1      function testConstantlyWithdrawAndDepositAndRedeem() public
2          hasGuardian {
3          //user deposit
4          weth.mint(mintAmount, user);
5          uint256 userWethBalanceBefore = weth.balanceOf(user); //100
6          ether
7          vm.startPrank(user);
8          weth.approve(address(wethVaultShares), mintAmount);
9          wethVaultShares.deposit(mintAmount, user);
10         //user redeem
11         uint256 maxShares = wethVaultShares.maxRedeem(user);
12         wethVaultShares.redeem(maxShares, user, user);
13         uint256 userWethBalanceAfter = weth.balanceOf(user); //52 ether
14         console2.log("userWethBalanceAfter", userWethBalanceAfter);
15         assert(userWethBalanceAfter < userWethBalanceBefore);
16     }
17 }
```

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)
2          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 testMaliciousGuardianTakeOverTheDAO() public {
2         address maliciousGuardian = makeAddr("maliciousGuardian");
3         weth.mint(mintAmount, maliciousGuardian);
4
5         uint256 vgTokenBefore = vaultGuardianToken.balanceOf(
           maliciousGuardian);
6         vm.startPrank(maliciousGuardian);
7         weth.approve(address(vaultGuardians), mintAmount);
8         for (uint256 i = 0; i < 10; i++) {
9             address maliciousGuardianWethVault = vaultGuardians.
               becomeGuardian(
10                 allocationData
11             );
12             //quit

```



```

13         IERC20(maliciousGuardianWethVault).approve(
14             address(vaultGuardians),
15             1.001e19
16         );
17         vaultGuardians.quitGuardian();
18     }
19     uint256 vgTokenAfter = vaultGuardianToken.balanceOf(
20         maliciousGuardian);
21     console2.log("vgTokenBefore is :", vgTokenBefore); //0
22     console2.log("vgTokenAfter is :", vgTokenAfter); //10e18
23     assert(vgTokenAfter > vgTokenBefore);
24     vm.stopPrank();
25 }

```

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] In the constructor of vaultshares, the liquidityToken of Uniswap is obtained by calling the getPair function of its factory contract, but it hardcodes WETH and asset.

Description: If `wethVaultShares`, i.e., the vault created by WETH, then both parameters passed to `getPair` would be WETH, which would cause an error.

```

1     constructor(
2         ConstructorData memory constructorData
3     )
4     {
5         i_guardian = constructorData.guardian;
6         i_guardianAndDaoCut = constructorData.guardianAndDaoCut;
7         i_vaultGuardians = constructorData.vaultGuardians;
8         s_isActive = true;
9         updateHoldingAllocation(constructorData.allocationData);
10
11
12         i_aaveAToken = IERC20(
13             IPool(constructorData.aavePool)
14                 .getReserveData(address(constructorData.asset))
15                 .aTokenAddress
16         );
17
18         i_uniswapLiquidityToken = IERC20(
19             i_uniswapFactory.getPair(
20 @>                 address(constructorData.asset),

```

```
21         address(i_weth)
22     )
23 );
24 }
```

[H-5] An attacker can impersonate the receiver and take all assets of a staker at VaultShares.

```
1  function withdraw(
2      uint256 assets,
3      address receiver,
4      address owner
5  )
6      public
7      override(IERC4626, ERC4626)
8      divestThenInvest
9      nonReentrant
10     returns (uint256)
11  {
12      uint256 shares = super.withdraw(assets, receiver, owner);
13      return shares;
14  }
15  function redeem(
16      uint256 shares,
17      address receiver,
18      address owner
19  )
20      public
21      override(IERC4626, ERC4626)
22      divestThenInvest
23      nonReentrant
24     returns (uint256)
25  {
26      uint256 assets = super.redeem(shares, receiver, owner);
27      return assets;
28  }
```

Impact: The user's staked amount will be fully stolen simply by the attacker changing the receiver parameter in the implementation of redeem and withdraw.

Proof of Concept:

Code

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

```
1  function testAttackerFrookTheReceiverToUserRedeem() public
2      hasGuardian {
3      //user deposit
4      weth.mint(mintAmount, user);
```

```

4      uint256 userWethBalanceBefore = weth.balanceOf(user); //100
      ether
5      vm.startPrank(user);
6      weth.approve(address(wethVaultShares), mintAmount);
7      wethVaultShares.deposit(mintAmount, user);
8      //attacker redeem
9      uint256 maxShares = wethVaultShares.maxRedeem(user);
10     wethVaultShares.redeem(maxShares, attacker, user);
11     uint256 userWethBalanceAfter = weth.balanceOf(user); //0 ether
12     console2.log("attacker balance after ", weth.balanceOf(attacker
        )); //52 ether
13     console2.log("userWethBalanceAfter", userWethBalanceAfter);
14
15     vm.stopPrank();
16 }
17
18 function testAttackerFrokTheReceiverToUserWithdraw() public
    hasGuardian {
19     //user deposit
20     weth.mint(mintAmount, user);
21     uint256 userWethBalanceBefore = weth.balanceOf(user); //100
        ether
22     vm.startPrank(user);
23     weth.approve(address(wethVaultShares), mintAmount);
24     wethVaultShares.deposit(mintAmount, user);
25     //attacker redeem
26     uint256 maxAssets = wethVaultShares.maxWithdraw(user);
27     wethVaultShares.withdraw(maxAssets, attacker, user);
28     uint256 userWethBalanceAfter = weth.balanceOf(user); //0 ether
29     console2.log("attacker balance after ", weth.balanceOf(attacker
        )); //52 ether
30     console2.log("userWethBalanceAfter", userWethBalanceAfter);
31
32     vm.stopPrank();
33 }

```

Recommended Mitigation: It is recommended to add a check for whether the receiver has shares before starting.

```

1 function redeem(
2     uint256 shares,
3     address receiver,
4     address owner
5 )
6     public
7     override(IERC4626, ERC4626)
8     divestThenInvest
9     nonReentrant
10    returns (uint256)
11 {
12 +    if(...){...revert}

```

```

13         uint256 assets = super.redeem(shares, receiver, owner);
14         return assets;
15     }
16     function withdraw(
17         uint256 assets,
18         address receiver,
19         address owner
20     )
21     public
22     override(IERC4626, ERC4626)
23     divestThenInvest
24     nonReentrant
25     returns (uint256)
26     {
27 +     if(...){...revert}
28         uint256 shares = super.withdraw(assets, receiver, owner);
29         return shares;
30     }

```

[H-6] In the addLiquidity function of the adapter, an investment error in assets caused Uniswap to occupy an additional 25%, i.e., 37.5%, in the entire trading strategy.

Description:

```

1     function _uniswapInvest(IERC20 token, uint256 amount) internal {
2         IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
           i_weth;
3
4         uint256 amountOfTokenToSwap = amount / 2;
5
6         s_pathArray = [address(token), address(counterPartyToken)];
7         bool succ = token.approve(
8             address(i_uniswapRouter),
9             amountOfTokenToSwap
10        );
11        if (!succ) {
12            revert UniswapAdapter__TransferFailed();
13        }
14        uint256[] memory amounts = i_uniswapRouter.
           swapExactTokensForTokens({
15            amountIn: amountOfTokenToSwap,
16            amountOutMin: 0,
17            path: s_pathArray,
18            to: address(this),
19            deadline: block.timestamp
20        });
21        succ = counterPartyToken.approve(address(i_uniswapRouter),
           amounts[1]);
22        if (!succ) {

```

```

23         revert UniswapAdapter__TransferFailed();
24     }
25     succ = token.approve(
26         address(i_uniswapRouter),
27         amountOfTokenToSwap + amounts[0]
28     );
29     if (!succ) {
30         revert UniswapAdapter__TransferFailed();
31     }
32     (
33         uint256 tokenAmount,
34         uint256 counterPartyTokenAmount,
35         uint256 liquidity
36     ) = i_uniswapRouter.addLiquidity({
37         tokenA: address(token),
38         tokenB: address(counterPartyToken),
39 @>         amountADesired: amountOfTokenToSwap + amounts[0],
40         amountBDesired: amounts[1],
41         amountAMin: 0,
42         amountBMin: 0,
43         to: address(this),
44         deadline: block.timestamp
45     });
46     emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
47         liquidity);
47     }

```

Impact: If the investment strategy is Hold: 50%, Uniswap: 25%, Aave: 25%, it would result in Uniswap receiving an additional 25% of the investment. This is because amounts[0] represents a 12.5% share, so 12.5% + 12.5% + 12.5% equals 37.5%.

Recommended Mitigation:

```

1     (
2         uint256 tokenAmount,
3         uint256 counterPartyTokenAmount,
4         uint256 liquidity
5     ) = i_uniswapRouter.addLiquidity({
6         tokenA: address(token),
7         tokenB: address(counterPartyToken),
8 -         amountADesired: amountOfTokenToSwap + amounts[0],
9 +         amountADesired: amountOfTokenToSwap ,
10        amountBDesired: amounts[1],
11        amountAMin: 0,
12        amountBMin: 0,
13        to: address(this),
14        deadline: block.timestamp
15    });
16    emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
17        liquidity);
17    }

```

[H-7] The VaultGuardianToken has no burning function, meaning it cannot be destroyed. Even if the fund manager exits the project, they will still retain voting rights.

Description:as you can see ,there is no burning function in the `VaultGuardianToken` contract. This means that the tokens cannot be destroyed, and if the fund manager exits the project, they will still retain voting rights.

```
1
2 contract VaultGuardianToken is ERC20, ERC20Permit, ERC20Votes, Ownable
3 {
4     constructor()
5         ERC20("VaultGuardianToken", "VGT")
6         ERC20Permit("VaultGuardianToken")
7         Ownable(msg.sender)
8     {}
9
10    function _update(
11        address from,
12        address to,
13        uint256 value
14    ) internal override(ERC20, ERC20Votes) {
15        super._update(from, to, value);
16    }
17
18    function nonces(
19        address ownerOfNonce
20    ) public view override(ERC20Permit, Nonces) returns (uint256) {
21        return super.nonces(ownerOfNonce);
22    }
23
24    function mint(address to, uint256 amount) external onlyOwner {
25        _mint(to, amount);
26    }
27 }
```

Impact:Even if the fund manager exits the project, the retention of tokens means they still have equivalent voting rights, which could suppress the activity of current fund managers.

Recommended Mitigation:Add a burn function to the original contract and set conditions to automatically burn the VGT tokens of a VaultGuardian or significantly reduce their voting power after they exit, thereby decreasing their voting weight.

[H-8] The vaultshares contract is one-time use. When a guardian exits the project, the contract is set to inactive. However, if the next fund manager wants to participate, there is no function to reactivate the contract.

Description: Suppose a guardian creates a USDC contract and invests according to their strategy. Once they exit, the contract is set to inactive. However, if a second guardian wants to participate, they can only create a new contract rather than investing based on the original one, significantly reducing liquidity. If users stake in the first contract and it becomes inactive, they can only redeem or withdraw and then invest in the second contract. The redundant fees and operational complexity may discourage users from continuing to participate.

Recommended Mitigation: It is recommended to implement measures in the VaultGuardian contract by adding a new function that allows other guardians to join an existing vaultshares contract and autonomously allocate investment strategies.

Medium

[M-1] Potentially incorrect voting period and delay in governor may affect governance

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. This means that the voting period and delay will be far off what the developers intended, which could potentially affect the intended governance mechanics.

Consider updating the functions as follows:

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

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.

```
1  else if (address(token) == address(i_tokenTwo)) {
2      tokenVault =
3      new VaultShares(IVaultShares.ConstructorData({
4          asset: token,
5      -      vaultName: TOKEN_ONE_VAULT_NAME,
6      +      vaultName: TOKEN_TWO_VAULT_NAME,
7      -      vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
8      +      vaultSymbol: TOKEN_TWO_VAULT_SYMBOL,
9          guardian: msg.sender,
10         allocationData: allocationData,
11         aavePool: i_aavePool,
12         uniswapRouter: i_uniswapV2Router,
13         guardianAndDaoCut: s_guardianAndDaoCut,
14         vaultGuardian: address(this),
15         weth: address(i_weth),
16         usdc: address(i_tokenOne)
17     }));
```

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:

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

```
7         });  
8     }
```

Info

[I-1] The `IInvestableUniverseAdapter` interface has never been used.

Recommended Mitigation:

```
1 // SPDX-License-Identifier: MIT  
2 pragma solidity 0.8.20;  
3  
4 import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";  
5  
6 -interface IInvestableUniverseAdapter {  
7 -     // function invest(IERC20 token, uint256 amount) external;  
8 -     // function divest(IERC20 token, uint256 amount) external;  
9 }
```

[I-2] The variable `GUARDIAN_FEE` has never been used.

Recommended Mitigation:

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