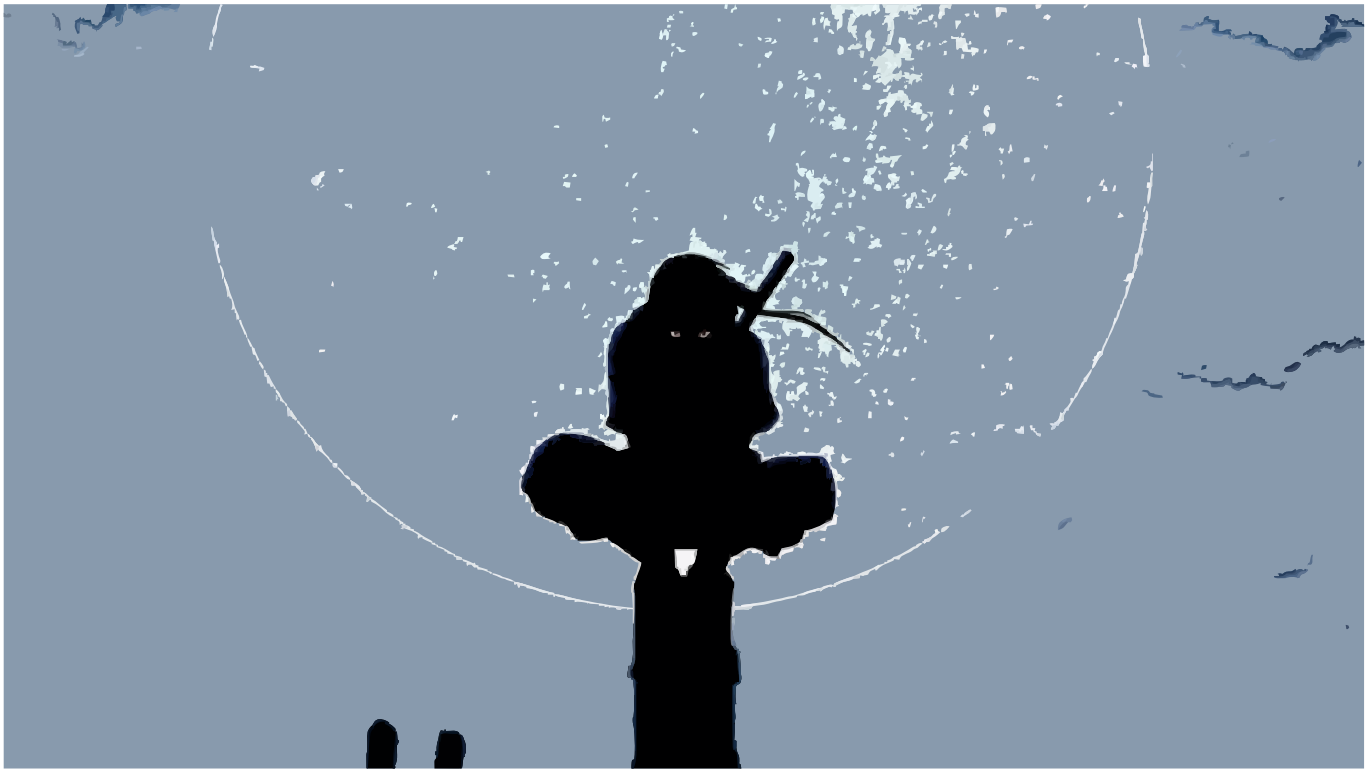

Vault Guardians Audit Report

Prepared by: mddragon18

Table of Contents

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.

Disclaimer

The mddragon18 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
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

The findings in the document are based on the commit hash :

```
Commit Hash: xxxxxxx
```

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
```

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	5
Medium	2
Low	3
Informational	0

Findings

Highs

[H-1] Allocated funds do not match `allocationData` due to double counting of tokens in `UniswapAdapter::_uniswapInvest`.

Description: The `_uniswapInvest` function is supposed to deposit a percentage of the vault's funds into uniswap pools, as uniswap pools must be provided liquidity in two tokens, half of the asset is swapped to WETH and along with it the asset is added to uniswap pools. The function misinterprets one of the return values of `addLiquidity` as the amount unspent in the transaction (`amounts[0]`) and add the `amountOfTokensToSwap` to it in the call to `addLiquidity` essentially doubling the deposit of the asset as the `amount[0]` is the input amount in a swap. https://github.com/Uniswap/v2-periphery/blob/0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f/contracts/libraries/UniswapV2Library.sol#L65C9-L65C31

▶ Code

```
(uint256 tokenAmount, uint256 counterPartyTokenAmount, uint256 liquidity) =
i_uniswapRouter.addLiquidity({
            tokenA: address(token),
            tokenB: address(counterPartyToken),

@>          amountADesired: amountOfTokenToSwap + amounts[0],
            amountBDesired: amounts[1],
            amountAMin: 0,
            amountBMin: 0,
            to: address(this),
            deadline: block.timestamp
        });
```

Impact: It makes the `allocationData` of the vault break by depositing more than desired amount of tokens into the uniswap pool which may cause losses to the vault and hence the user.

Proof of Concept: Add the following code to `test/unit/concrete/VaultShares.t.sol` :

▶ Code

```
function testIfFundsAreAllocatedProperly() public  hasGuardian {
        uint256 uniswapAllocationHalfWethSwapAmount =
(allocationData.uniswapAllocation * s_guardianStakePrice / 1000) / 2;
        uint256 holdBalanceRequired = allocationData.holdAllocation *
s_guardianStakePrice / 1000;
        uint256 actualHoldBalance =
weth.balanceOf(address(wethVaultShares));
        emit log_uint(holdBalanceRequired-actualHoldBalance);
        assertEq(holdBalanceRequired-
actualHoldBalance,uniswapAllocationHalfWethSwapAmount);
        assertNotEq(holdBalanceRequired,actualHoldBalance);
    }
```

Recommended Mitigation:

```
-       succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap
+ amounts[0]);
+       succ = token.approve(address(i_uniswapRouter), amounts[0]);
        if (!succ) {
            revert UniswapAdapter__TransferFailed();
        }

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

[H-2] 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:

```
    function swapExactTokensForTokens(
        uint256 amountIn,
@>      uint256 amountOutMin,
        address[] calldata path,
        address to,
@>      uint256 deadline
    )
```

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:

```
        uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens(
            amountOfTokenToSwap,
@>          0,
            s_pathArray,
            address(this),
@>          block.timestamp
        );
```

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.
2. This calls _uniswapInvest for a user to invest into Uniswap, and calls the router's swapExactTokensForTokens function.In the mempool, a malicious user could:

- Hold onto this transaction which makes the Uniswap swap
- Take a flashloan out
- Make a major swap on Uniswap, greatly changing the price of the assets
- 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 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-3] Lacks of checks in `VaultShares::withdraw` and `VaultShares::redeem` allow guardian to withdraw/redeem his stake while still being a guardian

Description: The protocol allows a guardian to create a vault after staking some amount of tokens and those tokens should are locked in the vault. But the lack of checks in withdraw and redeem allows the guardians to withdraw their stake while still being a guardian.

Impact: Guardian can easily withdraw his stake and break the invariance of the protocol that stake must be locked in untill the vault is not active and users have received all their funds.

Proof of Concept:
Add the following code to `test/unit/concrete/VaultShares.t.sol`

▶ Code

```
function testGuardianStakeInvarianceBreaks() public hasGuardian
userIsInvested {
        vm.startPrank(guardian);

wethVaultShares.approve(address(wethVaultShares),wethVaultShares.balanceOf(
guardian));

wethVaultShares.redeem(wethVaultShares.balanceOf(guardian),guardian,guardia
n);
        vm.stopPrank();
    }
```

Recommended Mitigation: Add checks so that guardian cannot withdraw his stake untill all users have withdrawn and the vault is inactive.

[H-4] `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

Proof of Concept:

▶ Code

```
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-5] Guardians can infinitely mint `VaultGuardianToken`s 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`.

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

```
function testDAOTakeover() public hasGuardian hasTokenGuardian {
        address malGuardian = makeAddr("MalGuardian");
        uint256 startingUsdcBalance = usdc.balanceOf(malGuardian);
        uint256 startingWethBalance = weth.balanceOf(malGuardian);

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

        weth.mint(mintAmount,malGuardian);
        uint256 startingMalVgBal = vg_token.balanceOf(malGuardian);
        uint256 startingRegVgBal = vg_token.balanceOf(guardian);

        vm.startPrank(malGuardian);
        weth.approve(address(vaultGuardians),type(uint256).max);
        for(uint256 i=0; i<10; i++) {
            address maliciousWethVaultShares =
vaultGuardians.becomeGuardian(allocationData);

IERC20(maliciousWethVaultShares).approve(address(vaultGuardians),IERC20(mal
iciousWethVaultShares).balanceOf(malGuardian));
            vaultGuardians.quitGuardian();
        }

        vm.stopPrank();

        uint256 endingMalVgBal = vg_token.balanceOf(malGuardian);

        emit log_uint(startingMalVgBal);
        emit log_uint(startingRegVgBal);
        emit log_uint(endingMalVgBal);
        emit log_uint(vg_token.balanceOf(guardian));
    }
```

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

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

Mediums

[M-1] `VaultGuardianGovernor` possibly uses incorrect units for `votingDelay` and `votingPeriod` causing unexpectedly short or long periods

Description: The `VaultGuardianGovernor` contract is the core governing contract of the DAO that controls `VaultGuardians` , the DAO has defined some `votingDelay()` and `votingPeriod()` as `1 days` and `7 days` respectively. The openzeppelin implementation of governor contract by default measures the `clock()` or time in units of `block.number`. The values given seem to represent the `block.timestamp` but they are misinterpreted as `86400` blocks and `86400*7` blocks instead of time.

▶ Code

```
    function votingDelay() public pure override returns (uint256) {
@>        return 1 days;
    }

    function votingPeriod() public pure override returns (uint256) {
@>        return 7 days;
    }
```

Impact: This can lead to incorrect amount of time and delays that need to pass for proposals to start voting or voting to end , causing unexpected behaviour.

Proof of Concept: Add the following code to `test/unit/concrete/VaultGuardianDao.t.sol` :

▶ Code

```
    function testDaoChangeStakePrice() public hasGuardian {
        vaultGuardianGovernor.CLOCK_MODE();     //
mode=blocknumber&from=default
        // Initial setup
        uint256 initialStakePrice = vaultGuardians.getGuardianStakePrice();
        uint256 newStakePrice = 12e18;

        // Delegate voting power to guardian
        vm.startPrank(guardian);
        vaultGuardianToken.delegate(guardian);
        vm.stopPrank();

        // Create proposal
        bytes memory encodedCall =
abi.encodeCall(vaultGuardians.updateGuardianStakePrice, newStakePrice);
```

```
        address[] memory targets = new address[](1);
        uint256[] memory values = new uint256[](1);
        bytes[] memory calldatas = new bytes[](1);
        targets[0] = address(vaultGuardians);
        values[0] = 0;
        calldatas[0] = encodedCall;
        string memory description = "Update guardian stake price";

        // Submit proposal
        vm.prank(guardian);
        uint256 proposalId = vaultGuardianGovernor.propose(targets, values,
calldatas, description);

        // Wait for voting delay but instead of vm.warp we have to use
vm.roll as the CLOCK_MODE is block.number
        vm.roll(block.number + vaultGuardianGovernor.votingDelay() + 1);

        // Cast vote
        vm.prank(guardian);
        vaultGuardianGovernor.castVote(proposalId, 0);

        // Wait for voting period but instead of vm.warp we have to use
vm.roll as the CLOCK_MODE is block.number
        vm.roll(block.number + vaultGuardianGovernor.votingPeriod() + 1);

        // Execute proposal
        vm.prank(guardian);
        vaultGuardianGovernor.execute(targets, values, calldatas,
keccak256(bytes(description)));

        // Verify stake price was updated
        assertEq(vaultGuardians.getGuardianStakePrice(), newStakePrice);
        assertNotEq(vaultGuardians.getGuardianStakePrice(),
initialStakePrice);
    }
```

Recommended Mitigation:

1. Override openzeppelin implementation to use `block.timestamp`.
2. Or Change the code as follows:

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

    function votingPeriod() public pure override returns (uint256) {
-       return 7 days;
+       return 50400;     //7 days in blocks
```

```
    }
```

[M-2] `UniswapAdapter::_uniswapDivest` lacks neccesary approvals to LP tokens to call `removeLiquidity` on the uniswap router

Description: The `_uniswapDivest` function is called to withdraw funds back into the vault. To do so , the uniswap pool must burn the LP's (vault in our case) LP tokens to give us the funds back. The uniswap documentation clearly state that before `removeLiquidity` is called , the LP tokens must be approved by the LP to be spent by the router contract. But the `_uniswapDivest` does not do so before withdrawing. The tests pass in our test suite because the uniswap router mock does not represent the uniswap router exactly, and stores the LP tokens in the router itself.

▶ Code

```
function _uniswapDivest(IERC20 token, uint256 liquidityAmount) internal
returns (uint256 amountOfAssetReturned) {
        IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
        //@audit-medium: No approval
        (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
i_uniswapRouter.removeLiquidity({
            tokenA: address(token),
            tokenB: address(counterPartyToken),
            liquidity: liquidityAmount,
            amountAMin: 0,
            amountBMin: 0,
            to: address(this),
            deadline: block.timestamp
        });
        s_pathArray = [address(counterPartyToken), address(token)];
        //@audit-medium: No approval
        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];
    }
```

Impact: Any withdraw attempts will be blocked from uniswap pools

Recommended Mitigation:

1. Provide neccesary approvals before calling `removeLiquidity`

```
    function _uniswapDivest(IERC20 token, uint256 liquidityAmount) internal
returns (uint256 amountOfAssetReturned) {
        IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
+
i_uniswapLiquidityToken.approve(address(i_uniswapRouter),type(uint256).max)
;
        (uint256 tokenAmount, uint256 counterPartyTokenAmount) =
i_uniswapRouter.removeLiquidity({
            tokenA: address(token),
            tokenB: address(counterPartyToken),
            liquidity: liquidityAmount,
            amountAMin: 0,
            amountBMin: 0,
            to: address(this),
            deadline: block.timestamp
        });
        s_pathArray = [address(counterPartyToken), address(token)];
+
counterPartyToken.approve(address(i_uniswapRouter),counterPartyTokenAmount)
;
        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];
    }
```

Lows

[L-1] Incorrect metadata supplied by `VaultGuardiansBase::becomeTokenGuardian()` for `tokenTwo` vaults

Description: The `becomeTokenGuardian` function is supposed to create a new vault when a new guardian calls this function. The function deploys a new `VaultShares` contract by supplying the parameters in form of a struct. The function when making a new contract for `tokenTwo` provides the name and symbols of `tokenOne`.

```
    else if (address(token) == address(i_tokenTwo)) {
            tokenVault =
            new VaultShares(IVaultShares.ConstructorData({
```

```
                asset: token,
      @>        vaultName: TOKEN_ONE_VAULT_NAME,        //@audit-low:
  Should be TOKEN_TWO*
                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: Leads to incorrect names and event emissions leading to confusion.

Recommended Mitigation:

```
else if (address(token) == address(i_tokenTwo)) {
            tokenVault =
            new VaultShares(IVaultShares.ConstructorData({
                asset: token,
-               vaultName: TOKEN_ONE_VAULT_NAME,
-               vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
+               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-2] `VaultGuardianBase::becomeGuardian` is not payable to receive a `0.5 ETH` fee as specfied by the natspec of the function leading to loss in fees.

Description: The `becomeGuardian` function is supposed to take a `0.5 ETH` fee along with the stake of `s_guardianStakePrice` to become a guardian. But the function is not payable to accept any fees nor does it check for any fee received.

▶ Code

```
    * @notice allows a user to become a guardian
@>      * @notice they have to send an ETH amount equal to the fee, and a
WETH amount equal to the stake price
     * //@audit-low: Function is not payable to accept a 0.5 ETH fee as in
GUARDIAN_FEE as required by the docs.
     * @param wethAllocationData the allocation data for the WETH vault
     */
@>     function becomeGuardian(AllocationData memory wethAllocationData)
external returns (address) {
        VaultShares wethVault =
        new VaultShares(IVaultShares.ConstructorData({
            asset: i_weth,
            vaultName: WETH_VAULT_NAME,
            vaultSymbol: WETH_VAULT_SYMBOL,
            guardian: msg.sender,
            allocationData: wethAllocationData,
            aavePool: i_aavePool,
            uniswapRouter: i_uniswapV2Router,
            guardianAndDaoCut: s_guardianAndDaoCut,
            vaultGuardians: address(this),
            weth: address(i_weth),
            usdc: address(i_tokenOne)
        }));
        return _becomeTokenGuardian(i_weth, wethVault);
    }
```

Impact: This leads to loss of potential fees to the protocol.

Recommended Mitigation:

```
-     function becomeGuardian(AllocationData memory wethAllocationData)
external returns (address) {
+     function becomeGuardian(AllocationData memory wethAllocationData)
external payable returns (address) {
+           require(msg.value >= GUARDIAN_FEE);
```

[L-3] `AaveAdapter::_aaveDivest` function declares a return variable but does not assign the variable

Description: The `_aaveDivest` function calls the aave pool to withdraw the vault's investment in the pool back into the vault. It call the `withdraw` function which returns a value of amount of assets returned , but the return value is ignored.

Impact: This is not a vulnerability but will cause problems if the aave protocol updates in the future.

▶ Code

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

Recommended Mitigation:

```
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)
        });
    }
```