Vault Guardians Audit Report

Prepared by KupiaSec

Version 1.0

Auditors

KupiaSec

November 25, 2023

Contents

1 About KupiaSec

KupiaSec is a team of Web3 security experts that operates with transparency and a meritocratic spirit.

KupiaSec executes the modified MPA model for the Private Audits, a.k.a. **Solo Audit by a Lead + Internal Competition + Mitigation Review**.

- Solo Audit by a Lead

  KupiaSec assigns a senior auditor as a Lead Auditor based on the protocol category. The Lead Auditor is responsible for the first phase and will be the main point of contact for the client. The Lead Auditor shares the analysis and findings with the team.

- Internal Competition

  KupiaSec assigns 5~7 assist auditors to conduct the second phase. The auditors compete to find the most issues and the best solutions. This phase ensures the protocol goes through a rigorous review process by "many eyes" in a competitive environment.

- Mitigation Review

  After the protocol team has fixed the issues, KupiaSec conducts a final review to ensure all the issues are resolved.

2 Disclaimer

The KupiaSec team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse 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.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Vault Guardians 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.

5 Audit Scope

Solidity source files in `src` folder were in the audit scope for all commit hashes.

Summary of Commits

Project Name	Vault Guardians
Repository	8-vault-guardians-audit
Initial Commit	b5c341de529d. . .
Mitigation Commit	N/A. . .

6 Executive Summary

KupiaSec executed a modified Multi-Phase Audit model, a.k.a. **Solo Audit by a Lead + Internal Competition + Mitigation Review**. According to the client's request, we skipped the third round - Mitigation Review.

Zhang conducted the audit as the Lead Auditor and 3 auditors competed in the second phase.

Execution Timeline

Phase-1: Audit by a Lead	Nov 1st - Nov 3rd
Phase-2: Internal Competition	Nov 6th - Nov 9th
Initial Report Delivery	Nov 9th, 2023
Phase-3: Mitigation Review	N/A
Final Report Delivery	Nov 25th, 2023

Issues Found

Critical Risk	0
High Risk	4
Medium Risk	9
Low Risk	4
Informational	3
Gas Optimizations	0
Total Issues	20

7 Findings

7.1 High

7.1.1 A user can become a guardian without paying a fee

Severity: High

Description: From a `GUARDIAN_FEE` param and comment, the protocol is meant to charge a registration fee for guardians, but its implementation is flawed.

Impact: The protocol wouldn't charge a fee as intended.

Proof of Concept: In `VaultGuardiansBase` contract, there is a GUARDIAN_FEE = 0.1 ether and it's set up to charge a fee mentioned in the note below.

```
     * @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 no implementation
     *
     * @param wethAllocationData the allocation data for the WETH vault
     */
    function becomeGuardian(AllocationData memory wethAllocationData) external returns (address) {}
```

However, the current implementation is missing the fee logic.

Recommendation: We should implement a fee logic.

7.1.2 In `VaultShares`, users can bypass the guardian fee mechanism using `ERC4626.mint()` instead of `deposit()`

Severity: High

Description: In `VaultShares.deposit()`, it mints additional shares for guardians but that logic doesn't exist in `mint()` as `ERC4626.mint()` hasn't been overridden. So users can use the `mint()` function to avoid those fees.

Impact: Users could break the protocol's fee mechanism using the `mint()` function.

Proof of Concept: Users can deposit funds to the vault and get shares using `deposit()`.

```
    function deposit(uint256 assets, address receiver)
        public
        override(ERC4626, IERC4626)
        isActive
        nonReentrant
        returns (uint256)
    {
        if (assets > maxDeposit(receiver)) {
            revert VaultShares__DepositMoreThanMax(assets, maxDeposit(receiver));
        }

        uint256 shares = previewDeposit(assets);
        _deposit(_msgSender(), receiver, assets, shares);

        _mint(i_guardian, shares / i_guardianAndDaoCut); //@audit bypass with mint()
        _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);

        _investFunds(assets);
        return shares;
    }
```

It mints additional shares for guardians, resulting in funds loss for normal users.

`ERC4626` contract has 2 functions - `deposit()` and `mint()` to be used to deposit funds to the vault.

So users can call ERC4626.mint() instead and bypass the fee mechanism.

Recommendation: We should add the same fee mechanism in `mint()` also.

7.1.3 `VaultGuardians` contract can't withdraw their shares from the vault

Severity: High

Description: While depositing funds to the guardian vault, it mints shares for `VaultGuardians` as a fee in `VaultShares.deposit()`. But there is no logic to withdraw these shares from the vault.

Impact: Accumulated shares for `VaultGuardians` will be locked in the vault forever.

Proof of Concept: In `VaultShares.deposit()`, it mints some shares for the `VaultGuardians` contract.

```
    function deposit(uint256 assets, address receiver)
        public
        override(ERC4626, IERC4626)
        isActive
        nonReentrant
        returns (uint256)
    {
        if (assets > maxDeposit(receiver)) {
            revert VaultShares__DepositMoreThanMax(assets, maxDeposit(receiver));
        }

        uint256 shares = previewDeposit(assets);
        _deposit(_msgSender(), receiver, assets, shares);

        _mint(i_guardian, shares / i_guardianAndDaoCut);
        _mint(i_vaultGuardians, shares / i_guardianAndDaoCut); //@audit fund lock

        _investFunds(assets);
        return shares;
    }
```

There are 2 ways to withdraw shares from the vault.

- `VaultGuardians` contract should calls `VaultShares.withdraw()/redeem()` directly but there is no such logic.
- Otherwise, the `VaultGuardians` contract should approve its vault shares to another address(like the owner) and he withdraws instead. But currently, there is no approval logic in `VaultGuardians`.

As a result, those shares will be locked in the vault forever.

Recommendation: We should add a function to withdraw accumulated shares from guardian vaults in `VaultGuardians`.

```
function withdrawDAOShares(IVaultShares vault) external return (uint256) {
    uint256 maxRedeemable = vault.maxRedeem(address(this)); //accmulated shares for VaultGuardians
    uint256 numberOfAssetsReturned = vault.redeem(maxRedeemable, owner(), address(this)); //transfer to
    ↪  the owner
    return numberOfAssetsReturned;
}
```

7.1.4 Lack of token approvals in `UniswapAdapter._uniswapDivest()`

Severity: High

Description: In `_uniswapDivest()`, it doesn't approve before calling `i_uniswapRouter.removeLiquidity()/swapExactTokens`
As a result, this function will always revert.

Impact: User funds will be locked forever as `_uniswapDivest()` always reverts.

Proof of Concept: `_uniswapDivest()` removes liquidity from Uniswap and swaps to the vault asset.

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

    (uint256 tokenAmount, uint256 counterPartyTokenAmount) = i_uniswapRouter.removeLiquidity({
    ↪   //@audit no approval
        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({ //@audit no approval
        amountIn: counterPartyTokenAmount,
        amountOutMin: 0,
        path: s_pathArray,
        to: address(this),
        deadline: block.timestamp
    });
    emit UniswapDivested(tokenAmount, amounts[1]);
    amountOfAssetReturned = amounts[1];
}
```

But it doesn't approve so it will revert while transferring the tokens to the router in removeLiquidity() and swapExactTokensForTokens().

Recommendation: We should approve the transfer first like _uniswapInvest() before calling `removeLiquidity()/swapExactTokensForTokens()`.

7.2 Medium

7.2.1 `VaultGuardiansBase.s_guardianStakePrice` is meaningless for `USDC`

Severity: Medium

Description: USDC token has 6 decimals and `s_guardianStakePrice = 10 ether` means 1e13 USDC which is higher than the current total supply = 22 billion.

Impact: Users couldn't use the `USDC` vault due to the very high `StakePrice`.

Proof of Concept: In `VaultGuardiansBase`, `s_guardianStakePrice` is set to `10 ether` by default. This amount will be used as an initial stake price in `_becomeTokenGuardian()` for all WETH/USDC/LINK tokens.

```
    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); //@audit 1e13 USDC
        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);
    }
```

It might be appropriate for WETH/LINK as these tokens have 18 decimals but it's extremely high for USDC of 6 decimals.

Recommendation: I think we should use different stake prices for each token.

7.2.2 `_becomeTokenGuardian()` might overwrite an already existing vault

Severity: Medium

Description: When users register to become guardians, it doesn't validate if they've created a vault for the token already.

Impact: If users register a vault for the same token again by fault, they can't control the previous one.

Proof of Concept: When users call `becomeGuardian()/becomeTokenGuardian()` to be guardians, `_becomeTokenGuardian()` stores the vault for `user/token`.

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

But if a user has a vault already, it just replaces the vault without handling the previous one. As a result, he couldn't manage the previous vault anymore. (like quitting from the token vault).

Recommendation: `_becomeTokenGuardian()` should revert if a user has the same token vault already.

7.2.3 `VaultShares.i_uniswapLiquidityToken` will be address(0) for `weth`

Severity: Medium

Description: In the constructor of `VaultShares`, `i_uniswapLiquidityToken` is set wrongly if `constructorData.asset == i_weth`.

Impact: The vault couldn't interact with Uniswap because `i_uniswapLiquidityToken = address(0)`.

Proof of Concept: In the constructor, `i_uniswapLiquidityToken` is set using `uniswapFactory.getPair()`.

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

If `constructorData.asset == i_weth` for weth vault, `i_uniswapFactory.getPair(i_weth, i_weth)` will be called and it will return address(0) because it's impossible to create a pair of same tokens.

Originally, this `weth` vault should work with `(WETH, USDC)` pair but it won't work properly because `i_uniswapLiquidityToken = address(0)`.

Recommendation: We should set `i_uniswapLiquidityToken` like the below in the constructor.

```
i_uniswapLiquidityToken = address(constructorData.asset) == constructorData.weth ?
                          IERC20(i_uniswapFactory.getPair(constructorData.usdc, constructorData.weth)) :
                          ↪   // (usdc, weth) for weth vault
                          IERC20(i_uniswapFactory.getPair(address(constructorData.asset),
                          ↪   constructorData.weth)); // (asset, weth) for token vault
```

7.2.4 In `VaultShares.deposit()`, the fee mechanism might be unfair to prior users.

Severity: Medium

Description: `deposit()` mints additional shares for guardians without decreasing the depositor's shares. It means all depositors collectively cover the fee by inflating the shares, which might be unfair to prior users.

Impact: The fee mechanism wouldn't work as intended.

Proof of Concept: `deposit()` mints additional shares for guardians as a fee. As it doesn't decrease the depositor's shares, it will inflate the shares like below.

```
    function deposit(uint256 assets, address receiver)
        public
        override(ERC4626, IERC4626)
        isActive
        nonReentrant
        returns (uint256)
    {
        if (assets > maxDeposit(receiver)) {
            revert VaultShares__DepositMoreThanMax(assets, maxDeposit(receiver));
        }

        uint256 shares = previewDeposit(assets);
        _deposit(_msgSender(), receiver, assets, shares);

        _mint(i_guardian, shares / i_guardianAndDaoCut); //@audit inflate shares
        _mint(i_vaultGuardians, shares / i_guardianAndDaoCut);

        _investFunds(assets);
        return shares;
    }
```

- Originally the vault contains 1000 assets/1000 shares for Alice. `i_guardianAndDaoCut = 1000`. Alice can claim 1000 assets with her shares at the moment.
- Bob calls `deposit()` with 1000 assets, `previewDeposit(1000)` will return 1000 shares correctly.
- Currently, Bob receives 1000 shares, and `i_guardian` and `i_vaultGuardians` will get 1 share.
- So the vault will have 2000 assets/2002 shares and Alice can claim `totalAssets * aliceShares / totalSupply = 2000 * 1000 / 2002 = 999` assets now.

- From a logical standpoint, it makes sense for Bob to cover the fee and acquire 998 shares, ensuring no impact on other users.

Recommendation: We should consider changing the fee mechanism.

7.2.5 `VaultShares` doesn't work with 0 `aaveAllocation`

Severity: Medium

Description: In `VaultShares`, there are 3 allocation types of hold/Uniswap/Aave. If `aaveAllocation = 0`, it will try to supply 0 amount to the Aave pool and it will always revert.

Impact: Users wouldn't be able to create a vault that interacts with Uniswap only.

Proof of Concept: When users deposit funds to a vault using `deposit()`, `_investFunds()` is called to supply to Uniswap/Aave according to the allocation ratios.

```
    function _investFunds(uint256 assets) private {
        uint256 uniswapAllocation = (assets * s_allocationData.uniswapAllocation) /
        ↪  ALLOCATION_PRECISION;
        uint256 aaveAllocation = (assets * s_allocationData.aaveAllocation) / ALLOCATION_PRECISION;

        emit FundsInvested();

        _uniswapInvest(IERC20(asset()), uniswapAllocation);
        _aaveInvest(IERC20(asset()), aaveAllocation); //@audit revert if aaveAllocation = 0
    }
```

If users set `s_allocationData.aaveAllocation = 0` to interact with Uniswap only, it will try to supply 0 to Aave pool and it will revert due to this validation.

So users can't create such vaults as this function keeps reverting.

Recommendation: `_investFunds()` should check if the `aaveAllocation` amount is positive.

```
    function _investFunds(uint256 assets) private {
        uint256 uniswapAllocation = (assets * s_allocationData.uniswapAllocation) /
        ↪  ALLOCATION_PRECISION;
        uint256 aaveAllocation = (assets * s_allocationData.aaveAllocation) / ALLOCATION_PRECISION;

        emit FundsInvested();

        _uniswapInvest(IERC20(asset()), uniswapAllocation);

        if (aaveAllocation != 0) {
            _aaveInvest(IERC20(asset()), aaveAllocation);
        }
    }
```

7.2.6 `VaultShares` doesn't work with 0 `uniswapAllocation`

Severity: Medium

Description: In `VaultShares`, there are 3 allocation types of hold/Uniswap/Aave. If `uniswapAllocation = 0`, it will revert during the swap from `token` to `counterPartyToken`.

Impact: Users wouldn't be able to create a vault that interacts with Aave only.

Proof of Concept: When users deposit funds to a vault using `deposit()`, `_investFunds()` is called to supply to Uniswap/Aave according to the allocation ratios.

```
    function _investFunds(uint256 assets) private {
        uint256 uniswapAllocation = (assets * s_allocationData.uniswapAllocation) /
        ↪  ALLOCATION_PRECISION;
        uint256 aaveAllocation = (assets * s_allocationData.aaveAllocation) / ALLOCATION_PRECISION;

        emit FundsInvested();

        _uniswapInvest(IERC20(asset()), uniswapAllocation); //@audit revert if uniswapAllocation = 0
        _aaveInvest(IERC20(asset()), aaveAllocation);
    }
```

If users set `s_allocationData.uniswapAllocation = 0` to interact with Aave only, it will try to swap `amountOfTokenToSwap = amount / 2` amount of `token` to `counterPartyToken` using i_uniswapRouter.swapExactTokensForTokens() and it will revert due to this validation from UniswapV2Router01.swapExactTokensForTokens().

So users can't create such vaults as this function keeps reverting.

Recommendation: `_investFunds()` should check if `uniswapAllocation / 2` is positive(which equals `uniswapAllocation > 1`).

```
    function _investFunds(uint256 assets) private {
        uint256 uniswapAllocation = (assets * s_allocationData.uniswapAllocation) /
        ↪  ALLOCATION_PRECISION;
        uint256 aaveAllocation = (assets * s_allocationData.aaveAllocation) / ALLOCATION_PRECISION;

        emit FundsInvested();

        if (uniswapAllocation > 1) {
            _uniswapInvest(IERC20(asset()), uniswapAllocation);
        }

        if (aaveAllocation != 0) {
            _aaveInvest(IERC20(asset()), aaveAllocation);
        }
    }
```

7.2.7 `GovernorVotesQuorumFraction` is initialized with too low `quorumNumeratorValue`

Severity: Medium

Description: In `VaultGuardianGovernor.sol`, `GovernorVotesQuorumFraction` is initialized with 4.

```
    constructor(IVotes _voteToken)
        Governor("VaultGuardianGovernor")
        GovernorVotes(_voteToken)
        GovernorVotesQuorumFraction(4) //@audit 4% is too low
    {}
```

It means the standard quorum is 4% of total supply which might be too low.

Impact: `VaultGuardianGovernor` wouldn't work as expected due to the low quorum number.

Proof of Concept: From Openzeppelin's GovernorVotesQuorumFraction, `The fraction is specified as numerator / denominator. By default the denominator is 100, so quorum is specified as a percent: a numerator of 10 corresponds to quorum being 10% of total supply.`

According to documents, `the requirement for a quorum is protection against totally unrepresentative action in the name of the body by an unduly small number of persons.`

The existing quorum threshold of 4% appears insufficient, potentially permitting malicious actions by a small fraction of members.

Recommendation: Recommend increasing the default quorum fraction.

7.2.8 Unfair mint logic of `VaultGuardianToken`

Severity: Medium

Description: When a user requests to become a guardian by creating a vault using one of 3 tokens(WETH, LINK, USDC), it mints a `vgToken` in `VaultGuardiansBase._becomeTokenGuardian()`.

However, it currently generates the same quantity of vgToken for all three tokens, despite their varying values in USD.

Impact: The governance mechanism may not function as intended due to the incorrect distribution of `vgToken`.

Proof of Concept: `_becomeTokenGuardian()` mints `vgToken` when users create token(WETH/LINK/USDC) vaults.

```
    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); //@audit mint same amount for 3 tokens
        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);
    }
```

But this mint logic is unfair.

- Alice and Bob both became guardians by creating WETH vaults, each depositing 10 ETH (equivalent to 18K USD) and receiving 10 `vgToken` each.
- On the other hand, Charlie, creating WETH/LINK vaults, spent 10 ETH (18K USD) and 10 LINK (130 USD) to receive 20 `vgToken`.
- In the end, the total voting power of Alice and Bob becomes 20 after spending 20 ETH (36K USD). However, Charlie attains the same power with only 10 ETH and 10 LINK (18K + 130 USD).

Recommendation: We should adjust the amount of `vgToken` based on the respective value of each token.

7.2.9 `UniswapAdapter._uniswapInvest()` doesn't handle the remaining funds properly after adding liquidity.

Severity: Medium

Description: When we use `i_uniswapRouter.addLiquidity()`, one token might have some remaining amount according to the current reserve ratio. But `_uniswapInvest()` ignores the remainder.

Impact: Some `counterPartyToken` might be locked inside the vault forever.

Proof of Concept: `_uniswapInvest()` is used to add a liqudity to Uniswap.

It swaps half of the `token` to `counterPartyToken` and adds liquidity to the `(token, counterPartyToken)` pair.

```
function _uniswapInvest(IERC20 token, uint256 amount) internal {
        IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
        // We will do half in WETH and half in the token
        uint256 amountOfTokenToSwap = amount / 2;
        s_pathArray = [address(token), address(counterPartyToken)];

        // Swap and approve
        // ...

        // amounts[1] should be the WETH amount we got back
        (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
        });
        emit UniswapInvested(tokenAmount, counterPartyTokenAmount, liquidity);
    }
```

But in `i_uniswapRouter.addLiquidity()`, it calculates the optimal amounts according to the current reserve ratio.

Here is a simple example.

- For the `(tokenA, tokenB) = (token, counterPartyToken)` pair, let's assume `(reserveA, reserveB) = (200, 200)`. We consider there is no swap fee for simplicity.
- `_uniswapInvest()` is called with `amount = 100` and it swaps 50 `token` to `counterPartyToken`.
- According to the Uniswap formula, `K = 200 * 200 = (200 + 50) * (200 - 40) = 250 * 160`. So 40 `counterPartyToken` will be returned.
- While adding a liquidity, `addLiquidity()` is called with `(amountADesired, amountBDesired) = (50, 40)` and the current `(reserveA, reserveB) = (250, 160)`.
- So `amountBOptimal` will be amountADesired * reserveB / reserveA = 50 * 160 / 250 = 32.
- As a result, `amountBDesired - amountBOptimal = 40 - 32 = 8` `counterPartyToken` won't be used to add liquidity and will be locked forever as there is no relevant logic.

If `token` isn't fully used, the impact is low because it will be included in `Hold` balance but `counterPartyToken` wouldn't be retrieved.

Recommendation: We should add a logic to manage unused `counterPartyToken`(like swap back to `token`) in `_uniswapInvest()`. Otherwise, `_uniswapDivest()` should use an entire `counterPartyToken` balance during the swap.

7.3 Low

7.3.1 `VaultGuardians.updateGuardianAndDaoCut()` should check if `s_guardianAndDaoCut != 0`

If `s_guardianAndDaoCut` is set to 0 by fault, deposit() will revert due to 0 division error.

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

7.3.2 Wrong funds split after calling `VaultShares.updateHoldingAllocation()`

After updating the allocation settings using `updateHoldingAllocation()`, it doesn't rebalance the vault automatically. As `_investFunds()` splits the incoming funds with the new allocation ratio without rebalancing the original funds, it would be good to rebalance after changing the settings.

```
    function updateHoldingAllocation(AllocationData memory tokenAllocationData) public
    ↪ onlyVaultGuardians isActive {
        uint256 totalAllocation = tokenAllocationData.holdAllocation +
        ↪ tokenAllocationData.uniswapAllocation
            + tokenAllocationData.aaveAllocation;
        if (totalAllocation != ALLOCATION_PRECISION) {
            revert VaultShares__AllocationNot100Percent(totalAllocation);
        }
        s_allocationData = tokenAllocationData;
        emit UpdatedAllocation(tokenAllocationData);
    }
```

7.3.3 `UniswapAdapter._uniswapInvest()` might use a less token amount(1 wei difference)

```
    function _uniswapInvest(IERC20 token, uint256 amount) internal {
        IERC20 counterPartyToken = token == i_weth ? i_tokenOne : i_weth;
        // We will do half in WETH and half in the token
        uint256 amountOfTokenToSwap = amount / 2;
        s_pathArray = [address(token), address(counterPartyToken)];

        bool succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap);
        if (!succ) {
            revert UniswapAdapter__TransferFailed();
        }
        uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens({
            amountIn: amountOfTokenToSwap,
            amountOutMin: 0,
            path: s_pathArray,
            to: address(this),
            deadline: block.timestamp
        });

        succ = counterPartyToken.approve(address(i_uniswapRouter), amounts[1]);
        if (!succ) {
            revert UniswapAdapter__TransferFailed();
        }
        succ = token.approve(address(i_uniswapRouter), amountOfTokenToSwap + amounts[0]); //@audit less
        ↪ amount
        if (!succ) {
            revert UniswapAdapter__TransferFailed();
        }
        ...
    }
```

If `amount = amountOfTokenToSwap * 2 + 1`, the current token amount should be `amountOfTokenToSwap + 1 + amounts[0]` instead. As a mitigation, we can use `amount - amountOfTokenToSwap + amounts[0]`.

7.3.4 `UniswapAdapter._uniswapDivest()` returns a wrong `amountOfAssetReturned`

`amountOfAssetReturned` should be `tokenAmount + amounts[1]` instead of `amounts[1]`.

```
        emit UniswapDivested(tokenAmount, amounts[1]);
        amountOfAssetReturned = amounts[1]; //@audit plus tokenAmount
```

7.4 Informational

7.4.1 Events should be emitted before changing the settings

The `VaultGuardians__UpdatedStakePrice` event emits the same values as it's updated already.

```
    function updateGuardianStakePrice(uint256 newStakePrice) external onlyOwner {
        s_guardianStakePrice = newStakePrice;
        emit VaultGuardians__UpdatedStakePrice(s_guardianStakePrice, newStakePrice); //@audit should
        ↪  emit before update
    }
```

7.4.2 Wrong event

The event name is incorrect.

```
    function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
        s_guardianAndDaoCut = newCut;
        emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut, newCut); //@audit incorrect name
    }
```

7.4.3 Unused modifier

In `VaultShares`, the `onlyGuardian` modifier isn't used.

```
    modifier onlyGuardian() { //@audit unused
        if (msg.sender != i_guardian) {
            revert VaultShares__NotGuardian();
        }
        _;
    }
```