Protocol Audit Report

Version 1.0

OSOK

July 31, 2025

Protocol Audit Report

OSOK

20250731

Prepared by: OSOK Lead Auditors: - OSOK

Table of Contents

- Medium
  * [M-1] Potentially incorrect voting period and delay in governor may affect governance
  * [M-2] Centralization Risk for trusted owners
  * [M-3] Low Quorum Threshold May Lead to Governance Centralization
- Low
  * [L-1] Unassigned return value when divesting AAVE funds
  * [L-2] Incorrect vault name and symbol
  * [L-3] Missing checks for `address(0)` when assigning values to address state variables
  * [L-4] Incorrect event emitted in updateGuardianAndDaoCut may mislead off-chain systems
  * [L-5] Incorrect event emission in updateGuardianStakePrice omits old value
- Gas
  * [G-1] Unused Custom Error
  * [G-2]: **`public`** functions not used internally could be marked `external`

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 OSOK 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

		Impact		
	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

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	3
Medium	3
Low	5
Info	0
Gas	2
Total	13

Findings

High

[H-1] Using `block.timestamp` as Uniswap deadline in `UniswapAdpater::_uniswapInvest` and `UniswapAdpater::_uniswapDivest` disables expiry protection

Description: In both `UniswapAdapter::_uniswapInvest` and `UniswapAdapter::_uniswapDivest`, the protocol calls the `swapExactTokensForTokens` function on the `UniswapV2Router01` contract, which includes a deadline parameter:

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

As seen below, the `UniswapAdapter::_uniswapInvest` function sets parameter to `block.timestamp`:

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

This deadline defines the time after which the transaction becomes invalid, and is commonly used to prevent swaps from executing under stale or manipulated market conditions.

Impact: - A malicious validator or MEV bot could observe the transaction, delay its inclusion, manipulate the token price, and then include the swap at an advantageous time. - The lack of an expiry mechanism allows swaps to execute with outdated or manipulated pricing, increasing the risk of slippage or value loss. - Combined with amountOutMin = 0, this creates a worst-case scenario where the protocol blindly accepts any return value, even if heavily front-run.

Proof of Concepts:

1. A user deposits into a vault using `VaultShares::deposit`, triggering `_uniswapInvest` or `_uniswapDivest`, which submits a Uniswap swap transaction with deadline =

block.timestamp.

2. An MEV bot or validator sees the pending transaction, pulls a flashloan, and manipulates Uniswap prices.
3. Since the transaction has no effective expiry, it still executes at a manipulated rate, resulting in reduced value for the protocol.

Recommended mitigation:

To support different DeFi protocols and custom safety constraints, allow custom data in the deposit function:

```
- 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) {
```

Additionally, consider pairing this with slippage protection (amountOutMin) to fully defend against MEV and front-running attacks.

[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`.

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

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

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

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

Proof of Concept:

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

Code

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

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

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

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

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

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

Recommended Mitigation: 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.

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:

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

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

[M-2] Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

5 Found Instances

- Found in src/dao/VaultGuardianToken.sol Line: 9

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

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

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

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

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

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

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

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

```
    function updateGuardianAndDaoCut(uint256 newCut) external
        onlyOwner {
```

[M-3] Low Quorum Threshold May Lead to Governance Centralization

The contract VaultGuardianGovernor sets the quorum fraction to only 4%, which is the minimum number of votes required for a proposal to pass. This low threshold poses a risk of governance centralization.

```
constructor(IVotes _voteToken)
    Governor("VaultGuardianGovernor")
    GovernorVotes(_voteToken)
    GovernorVotesQuorumFraction(4) // 4% quorum
```

Low

[L-1] 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-2] 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)
    }));
```

[L-3] Missing checks for `address(0)` when assigning values to address state variables

In VaultGuardiansBase::_becomeTokenGuardian, the contract sets s_guardians[msg.sender][token] to the provided tokenVault address without checking if it is the zero address. If tokenVault is address(0), this will lead to storing an invalid guardian vault, causing potential issues in later interactions.

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

[L-4] Incorrect event emitted in updateGuardianAndDaoCut may mislead off-chain systems

In the updateGuardianAndDaoCut function, the contract updates s_guardianAndDaoCut, but emits an unrelated event VaultGuardians__UpdatedStakePrice. This event likely does not reflect the actual operation performed and could confuse off-chain listeners or user interfaces relying on event logs.

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

[L-5] Incorrect event emission in updateGuardianStakePrice omits old value

In updateGuardianStakePrice, the VaultGuardians__UpdatedStakePrice event is emitted using the updated value for both oldStakePrice and newStakePrice, rather than preserving the actual old value.

This can mislead any off-chain system or user interface relying on this event for changes.

```
function updateGuardianStakePrice(uint256 newStakePrice) external onlyOwner {
+       uint256 oldStakePrice = s_guardianStakePrice;
        s_guardianStakePrice = newStakePrice;
-       emit VaultGuardians__UpdatedStakePrice(s_guardianStakePrice, newStakePrice);
+       emit VaultGuardians__UpdatedStakePrice(oldStakePrice, newStakePrice);
    }
```

Gas

[G-1] Unused Custom Error

it is recommended that the definition be removed when custom error is unused

4 Found Instances

- Found in src/protocol/VaultGuardians.sol Line: 43

  ```
      error VaultGuardians__TransferFailed();
  ```

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

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

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

  ```
      error VaultGuardiansBase__CantQuitGuardianWithNonWethVaults(address guardianAddress);
  ```

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

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

[G-2]: `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: 115

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

- Found in src/protocol/VaultShares.sol Line: 140

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

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

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

- Found in src/protocol/VaultShares.sol Line: 189

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

- Found in src/protocol/VaultShares.sol Line: 206

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