**This is a V2 security Review V1 already done by the Cyfrin Team**

# Vault Guardians Protocol Security Review V2

**Reviewer:** Muhammad Faran
**Date:** July 17, 2025
**Repo:** Cyfrin/8-Vault-Guardians

---

## Table of Contents



---

## Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and view of the code was solely on the security aspects of the Solidity implementation of the contracts.

---

## Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation).
The matrix below illustrates how these two axes map to a single severity label:

| Likelihood ↓ / Impact → | High | Medium | Low |
|---|---|---|---|
| **High** | **High** | **High/Medium** | **Medium** |
| **Medium** | **High/Medium** | **Medium** | **Medium-Low** |
| **Low** | **Medium** | **Medium-Low** | **Low** |

---

## Executive Summary

| Severity | Number Of Issues Found |
|---|---|
| **High** | **1** |
| **Medium** | **0** |
| **Low** | **0** |
| **Informational** | **3** |

---

# Audit Details

## Scope

```
./src/
#-- abstract
|   #-- AStaticTokenData.sol
|   #-- AStaticUSDCData.sol
|   #-- AStaticWethData.sol
#-- dao
|   #-- VaultGuardianGovernor.sol
|   #-- VaultGuardianToken.sol
#-- interfaces
|   #-- IVaultData.sol
|   #-- IVaultGuardians.sol
|   #-- IVaultShares.sol
|   #-- InvestableUniverseAdapter.sol
```

```
#-- protocol
|    #-- VaultGuardians.sol
|    #-- VaultGuardiansBase.sol
|    #-- VaultShares.sol
|    #-- investableUniverseAdapters
|        #-- AaveAdapter.sol
|        #-- UniswapAdapter.sol
#-- vendor
     #-- DataTypes.sol
     #-- IPool.sol
     #-- IUniswapV2Factory.sol
     #-- IUniswapV2Router01.sol
```

---

# Protocol Summary

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

---

## Roles

There are 4 main roles associated with the system.

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

---

## [H-2] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapDivest` enables frontrunners to steal assets during withdrawal

**Description:**
In the `UniswapAdapter::_uniswapDivest` function, the protocol exits UniswapV2 LP positions by removing liquidity and performing a swap of one of the tokens back to the desired asset. When performing the swap, the contract again calls the `UniswapV2Router01.swapExactTokensForTokens()` function, but without setting adequate slippage.

Similar to the invest path, this swap call passes:

```
uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens(
    amountToSwap,
    0,                    // <--- amountOutMin set to 0 (no slippage prot
    s_pathArray,
    address(this),
    block.timestamp
);
```

This enables **frontrunners** and **MEV bots** to manipulate the price before the swap occurs, particularly because `amountOutMin` is set to zero, allowing the swap to succeed no matter how unfavorable the rate is.

**Impact:**

- Users withdrawing from the vault could receive significantly fewer assets than expected, if a bot sees the withdrawal transaction and manipulates the pool price before the swap.
- A malicious miner could reorder or delay the transaction to execute it during a price dip, resulting in loss of value for the vault or user.
- This creates a **guaranteed execution path** for attackers with **no slippage protection**, making it highly exploitable and economically unsafe.

**Proof of Concept:**

1. A user initiates a `VaultShares::redeem` or `VaultShares::withdraw` call from a vault containing a UniswapV2 LP position.
2. The `Vault` calls `UniswapAdapter::_uniswapDivest()`, which includes a call to `swapExactTokensForTokens` with `amountOutMin = 0`.
3. While the transaction is pending in the mempool:

    - A MEV bot or malicious actor takes a flashloan and performs a large swap to worsen the output price.
    - The pending withdrawal swap executes at the manipulated rate, leaving the user with fewer assets.
4. The bot repays its flashloan and captures the difference in value.

This exact pattern has been exploited across multiple protocols in the past, especially during redemptions or liquidations where guaranteed execution was paired with no slippage protection.

**Recommended Mitigation:**

*Similar to the invest path, implement one or more of the following solutions:*

1. **Slippage protection**
   Use an oracle (e.g., Chainlink price feeds) or TWAP to estimate expected output amount, and calculate a reasonable `amountOutMin` with a configurable slippage (e.g., 0.5% or 1%). Revert if the expected output is too low.

```
uint256 expectedOut = getExpectedAmountOut(amountIn, path); // from o
uint256 amountOutMin = expectedOut * 99 / 100; // allow 1% slippage
```

2. **Custom withdrawal parameters**
   Just like in the `deposit()` recommendation, allow passing custom `bytes customData` to the `redeem()` or `withdraw()` functions so the caller can specify slippage and deadline values themselves.

```
- function redeem(uint256 shares, address receiver, address owner)
+ function redeem(uint256 shares, address receiver, address owner, by
```

3. **Use TWAP oracles**
   Fetch historical price data from Uniswap TWAPs or Chainlink feeds and use those to validate acceptable price windows before executing the swap.

---

## [I-1] Empty Interfaces Provide No Functional or Documentation Value

**Description:**
The codebase includes couple of interfaces that are currently empty and do not define any functions, events, or structs. These interfaces serve no practical purpose and may introduce confusion during audits or future development.

**Impact:**

- Adds unnecessary clutter to the codebase.
- May confuse new developers or auditors trying to understand protocol boundaries.
- Could falsely signal modularity where there is none.

**Proof of Concept:**
These two interfaces are empty.

```
interface IInvestableUniverseAdapter {}

interface IVaultGuardians {}
```

**Recommended Mitigation:**

- Remove unused or empty interfaces unless they are placeholders for future extensions.
- If they are planned for future use, comment this explicitly and add a `TODO:` or `@notice` tag.

---

## [I-2] Non-Standard Import Practices Reduce Readability and Upgrade Safety

**Description:**
The project inconsistently uses import paths and sometimes relies on package-relative or deeply nested file paths instead of clear interface or contract package names. For example, contracts import OpenZeppelin modules like this:

```
import {VaultGuardiansBase, IERC20, SafeERC20} from "./VaultGuardiansBase
```

Instead of the more standard and maintainable:

```
import {ERC4626, ERC20, IERC20} from "@openzeppelin/contracts/token/ERC20
```

**Impact:**

- Makes it harder to refactor or upgrade dependencies.
- Breaks compatibility when switching package managers or restructuring the project directory.
- Decreases portability of contracts and complicates integration testing or modular reuse.

**Recommended Mitigation:**

- Standardize imports across the project using package-based paths (e.g., `@openzeppelin/...`) via `remappings.txt` or `foundry.toml`.
- Avoid deep relative paths to external libraries.

## [I-3] Unused Modifier in `VaultShares` Adds Dead Code and Confusion

**Description:**

The `VaultShares` contract includes a modifier that is never applied to any function. This may be leftover from previous development or an incomplete implementation.

**Impact:**

- Adds unnecessary complexity to the contract.
- May confuse reviewers or lead to incorrect assumptions about access control logic.
- Increases the risk of introducing bugs if reused inappropriately later.

**Proof of Concept:**

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

**Recommended Mitigation:**

- Remove unused modifiers to keep the contract clean and readable.
- If it's intended for future use, move it to a base contract or add an explanatory comment such as `// Reserved for future upgrade safety`.