Vault Guardian Audit Report

Version 1.0

Jack

August 07, 2025

Vault Guardian Audit Report

Jack

August 07, 2025

Prepared by: Jack

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.

You can think of a `vaultGuardian` as a fund manager.

To prevent a vault guardian from running off with the funds, the vault guardians are only allowed to deposit and withdraw the ERC20s into specific protocols.

- Aave v3
- Uniswap v2
- None (just hold)

These 2 protocols (plus “none” makes 3) are known as the “investable universe”.

The guardian can move funds in and out of these protocols as much as they like, but they cannot move funds to any other address.

The goal of a vault guardian, is to move funds in and out of these protocols to gain the most yield. Vault guardians charge a performance fee, the better the guardians do, the larger fee they will earn.

Anyone can become a Vault Guardian by depositing a certain amount of the ERC20 into the vault. This is called the `guardian stake`. If a guardian wishes to stop being a guardian, they give out all user deposits and withdraw their guardian stake.

Users can then move their funds between vault managers as they see fit.

The protocol is upgradeable so that if any of the platforms in the investable universe change, or we want to add more, we can do so.

Audit Details

Scope

The review focused on the commit hash `fde71d006420c557821114a17bb2791ecb046c4d` of the public https://github.com/Jack-OuCJ/vault-guardians-audit GitHub repository.

The list of files in scope can be found in Appendix 1

Actors/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.

Known Issues

- We are aware that USDC is behind a proxy and is susceptible to being paused and upgraded. Please assume for this audit that is not the case.

Disclaimer

This audit report should not be taken as a guarantee that all vulnerabilities have been identified and addressed. While every effort has been made to thoroughly review the smart contracts for potential vulnerabilities, it is not possible to absolutely ensure the complete absence of vulnerabilities. The audit is a time- and resource-limited engagement. As such, it is not possible to guarantee that all vulnerabilities will be identified or that the smart contracts will be completely secure following the audit. The auditor can not be held liable for any damages or losses that may arise from using the contracts in scope.

Severity classification

Severity	Description
High	A **directly** exploitable security vulnerability that leads to stolen/lost/locked/compromised assets or catastrophic denial of service. A security vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service. It may not be directly exploitable or may require certain, external conditions in order to be exploited.
Medium	Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.
Low	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Non-critical comments, recommendations or potential optimizations, not relevant to security. Code maintainers should use their own judgment as to whether to address such issues.

Summary

Severity	Number of issues found
High	3
Medium	2
Low	6
Informational	3
Total	14

Findings

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 `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.
    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.

```
- 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) {
```

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.

```
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`

```solidity
    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:

```diff
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] Unused GUARDIAN_FEE breaks the designed payment rule

Description:
A constant fee GUARDIAN_FEE = 0.1 ether is declared and documented, but neither becomeGuardian nor _becomeTokenGuardian enforces or collects this ETH fee. Only the WETH stake logic is executed.

Impact:
- Guardians can be created without paying the intended 0.1 ETH fee, reducing protocol revenue and weakening Sybil/spam deterrence.
- Divergence between docs and behavior can mislead users/integrators and complicate audits.

Recommended Mitigation:
Enforce the ETH fee and forward it to a designated recipient (e.g., owner/treasury). Emit an event for observability. If the fee is not required, remove the constant and update NatSpec.

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.

```
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)
    }));
```

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:

```
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-3] Missing access control in `VaultGuardians::sweepErc20s` allows arbitrary token sweeping

Description:
The function sweepErc20s is publicly callable and transfers the entire balance of any specified ERC20 from the contract to the owner without any access control. Any external account can trigger sweeping at any time.

Impact:
- Anyone can force-transfer all ERC20 balances held by the contract to the owner address.
- Legitimate in-contract accounting, vesting, or payout flows can be disrupted, leading to unexpected state and potential loss of user funds if tokens are meant to remain escrowed.

Proof of Concept:
1) Attacker calls sweepErc20s(USDC).
2) The function reads the entire USDC balance and emits VaultGuardians__SweptTokens.
3) It then transfers the full amount to owner(), even though the caller is unprivileged.
Result: tokens are moved unexpectedly; any module relying on those balances breaks.

Recommended Mitigation:
- Restrict the function with appropriate access control (e.g., onlyOwner, onlyGuardian, or a role via AccessControl) and consider pausable/guardian veto:

[L-4] `VaultShares::i_guardianAndDaoCut` not validated; zero value causes division-by-zero and deposit DoS

Description:
The constructor sets i_guardianAndDaoCut directly from constructorData.guardianAndDaoCut without validation. Later, deposit mints rewards using shares / i_guardianAndDaoCut for both i_guardian and i_vaultGuardians. If i_guardianAndDaoCut == 0, the division reverts, breaking deposits.

Impact:
- Denial of service: All deposit calls revert when i_guardianAndDaoCut is zero, rendering the vault unusable for deposits. - Potential bricking at deployment: If a bad parameter is passed during construction, the vault is immediately non-functional. - Even if updated later, any interim operations depending on deposit will fail until corrected.

Recommended Mitigation:
- Validate i_guardianAndDaoCut on construction and on any update function. Enforce a minimum of 1 and consider an upper bound to avoid excessive dilution/rounding.

[L-5] Missing zero-address validation for i_aaveAToken and i_uniswapLiquidityToken leads to downstream failures

Description: In the constructor, the contract fetches dependent token addresses via external calls: - i_aaveAToken from IPool(constructorData.aavePool).getReserveData(asset).aTokenAddress - i_uniswapLiquidityToken from i_uniswapFactory.getPair(asset, i_weth)

However, the code does not validate that these returned addresses are non-zero. For assets without an Aave reserve or without an existing Uniswap pair, these calls can legitimately return address(0). Persisting zero addresses into immutable state breaks later interactions that assume valid ERC20 contracts.

Impact: - Subsequent token operations (approve, balanceOf, transfer, deposit/withdrawal flows, liquidity operations) against address(0) will revert or behave unexpectedly, causing Denial of Service for core functions. - Deployment might succeed but the vault becomes partially or completely unusable depending on which dependency is missing. - Silent misconfiguration risk: without explicit checks, operators may not realize the environment is unsupported until runtime failures occur.

Recommended Mitigation: - Validate non-zero addresses immediately after fetching them and revert with clear errors if unsupported. Consider allowing "optional" dependencies by gating features on availability, rather than hard failing, if that aligns with product design. - Emit events for observability when dependencies are set.

[L-6] nonReentrant is not the first modifier on `VaultShares::redeem`, `VaultShares::withdraw`, and `VaultShares::deposit`

Description:
The functions redeem, withdraw, and deposit apply nonReentrant after other modifiers (divestThen-Invest, isActive). Best practices recommend placing nonReentrant as the first modifier to ensure reentrancy guards are set before any external calls or complex logic that other modifiers might trigger. Root cause is modifier ordering oversight.

Affected snippets: - redeem(…): override(IERC4626, ERC4626) divestThenInvest nonReentrant - withdraw(…): override(IERC4626, ERC4626) divestThenInvest nonReentrant - deposit(…): override(ERC4626, IERC4626) isActive nonReentrant

Impact:
- If earlier modifiers (e.g., divestThenInvest) perform external calls, state changes, or invoke hooks before the reentrancy guard is active, an attacker could exploit reentrancy into the same or related functions, potentially bypassing checks or manipulating state. - Even if current modifiers are "view" or safe today, future changes could introduce external interactions, silently weakening the reentrancy protection due to ordering.

Recommended Mitigation:
- Place nonReentrant as the first non-override modifier in all externally callable functions. - Keep overrides first (per Solidity syntax), then nonReentrant, then business modifiers.

Informational

[I-1] Incorrect event emitted in `VaultGuardians::updateGuardianAndDaoCut`

Description: The function updateGuardianAndDaoCut updates s_guardianAndDaoCut but emits an unrelated event VaultGuardians__UpdatedStakePrice. This causes a mismatch between the state change and the emitted log that off-chain consumers rely on.

Recommended Mitigation: - Emit a dedicated event that matches the state change and includes both old and new values: - Declare: event GuardianAndDaoCutUpdated(uint256 oldCut, uint256 newCut); - Implement: function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner { uint256 old = s_guardianAndDaoCut; s_guardianAndDaoCut = newCut; emit GuardianAndDaoCutUpdated(old, newCut); }

[I-2] Missing zero-address validation for tokenVault in `VaultGuardiansBase::_becomeTokenGuardian`

Description:
_becomeTokenGuardian assigns tokenVault to state and immediately uses it without verifying that tokenVault is a non-zero contract address. If a zero address is passed due to construction failure or upstream logic error, subsequent approve/deposit calls will revert or result in undefined behavior assumptions.

Impact:
- Immediate revert when calling token.approve(address(tokenVault), …) with address(0), halting the guardian onboarding flow.
- Potential state inconsistency: s_guardians mapping is updated before failures occur, leaving a stale or misleading guardian entry pointing to an invalid vault.
- Reduced diagnosability since the revert reason will be generic (TransferFailed) rather than indicating an invalid tokenVault address.

Recommended Mitigation:
- Validate tokenVault is a non-zero address and, ideally, that it points to a contract implementing the expected interface before updating state or making external calls.
- Reorder operations to perform validations before state changes and side effects.

[I-3] Unused custom errors obscure intent and add maintenance overhead

Description:
In contract VaultGuardiansBase, several custom errors are declared but never used in the provided code paths: - VaultGuardiansBase__NotEnoughWeth(uint256 amount, uint256 amountNeeded) - VaultGuardiansBase__CantQuitGuardianWithNonWethVaults(address guardianAddress) - VaultGuardiansBase__FeeTooSmall(uint256 fee, uint256 requiredFee)

Other errors in the same block are used, which highlights inconsistency and suggests missing validations tied to fees, WETH sufficiency, or quitting constraints. The root cause is leftover or planned-but-unimplemented checks that leave dead code artifacts.

Impact:
- Ambiguity and intent drift: Readers assume corresponding checks exist, wasting time searching for them. - Higher audit/maintenance cost: Reviewers must confirm whether logic is missing or the errors are obsolete. - Potentially missing safeguards: The presence of these errors implies important rules that might be unintentionally unenforced.

Recommended Mitigation:

- If these rules are intended, implement the checks and use the errors: - Validate fee payments and revert with VaultGuardiansBase__FeeTooSmall(fee, requiredFee). - Enforce WETH sufficiency before operations and revert with VaultGuardiansBase__NotEnoughWeth(amount, amountNeeded). - Gate quitting logic for non-WETH vaults and revert with VaultGuardiansBase__CantQuitGuardianWithNonWethVaults(guardian). - If not intended, remove the unused error definitions to reduce noise.

Appendix

Appendix 1

This audit covered the following files:

File
src/abstract/AStaticTokenData.sol
src/abstract/AStaticUSDCData.sol
src/abstract/AStaticWethData.sol
src/dao/VaultGuardianGovernor.sol
src/dao/VaultGuardianToken.sol
src/interfaces/IVaultData.sol
src/interfaces/IVaultGuardians.sol
src/interfaces/IVaultShares.sol
src/interfaces/InvestableUniverseAdapter.sol
src/protocol/VaultGuardians.sol
src/protocol/VaultGuardiansBase.sol
src/protocol/VaultShares.sol
src/protocol/investableUniverseAdapters/AaveAdapter.sol
src/protocol/investableUniverseAdapters/UniswapAdapter.sol
src/vendor/DataTypes.sol
src/vendor/IPool.sol
src/vendor/IUniswapV2Factory.sol

File
src/vendor/IUniswapV2Router01.sol