Vault-Guardian Audit Report

Prepared by: Bikalpa

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

Bikalpa Regmi 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
Likelihood	High	H	H/M	M
	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

Commit Hash : main branch

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 like a mutual fund by depositing on aave and uniswap for yeild earning, but makes some large mistakes on tracking balances and profits.

Issues found

Severity	Number of Issue Found
High	3
Medium	1
Low	3
gas	1
Info	1
Total	9

Findings

High

[H-1] Missing burning of vg tokens leading to infinite vg tokens minting.

Description: The `VaultGuardianBase::_quitGuardian` was supposed to let the guardian left the DAO and loose all the rights given to them. However, the `VaultGuardianBase::_quitGuardian` function lacks the burning implementation of vg token. The users vg token doesn't burns even if he left.

Impact: This leads to a person performing multiple flashloans and gaining infinite vg tokens and gaining power without loosing or burning of his token & exploit the decisions.

Proof of Concept: Place the following code into VaultGuardiansBaseTest.t.sol

▶ Details

```
    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: Burn the vg token when the guardian quits.

[H-2] Lack of slippage protection in `UniswapAdapter.sol::_uniswapInvest`.

Description: In `UniswapAdapter::_uniswapInvest` the protocol swaps half of an ERC20 token so that they can invest in both sides of a Uniswap pool. 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. However, the UniswapAdapter::_uniswapInvest function sets those parameters to `0` and `block.timestamp`.

Impact: 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.

Proof of Concept:

1. User calls VaultShares::deposit with a vault that has a Uniswap allocation. i. 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: i. Hold onto this transaction which makes the Uniswap swap ii. Take a flashloan out iii. Make a major swap on Uniswap, greatly changing the price of the assets iv. 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-3] `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.

Medium

[M-1] Missing `clock()` and `CLOCK_MODE()` overrides in Governor contract

Description:
The `VaultGuardianGovernor` contract inherits from OpenZeppelin's `Governor` module but does not override the `clock()` and `CLOCK_MODE()` functions introduced in newer versions of the `Governor` base contract. These functions are part of OpenZeppelin's mechanism to support off-chain governance and allow voting snapshots to be taken using alternative clocks such as block numbers, timestamps, or off-chain oracles.

Without these overrides, the Governor contract relies on default implementations that may not match the intended behavior, especially in cross-chain or L2 environments, or when using off-chain vote counting systems. It also makes the contract incompatible with certain governance tools and delegates that rely on these functions for safe replay protection or signature verification.

Impact:

- **Governance tool incompatibility:** External tools and interfaces that depend on `clock()` and `CLOCK_MODE()` may fail or behave incorrectly.
- **Reduced upgradeability:** Future extensions requiring off-chain clock logic will break unless these are defined.
- **Potential security assumptions broken** if voting period tracking or replay protection depends on the clock source.

Recommended Mitigation:
The contract is missing these functions so write it :

```
function clock() public view virtual override returns (uint48) {
    return uint48(block.timestamp); // or block.number
}

function CLOCK_MODE() public pure virtual override returns (string memory) {
    return "mode=timestamp"; // or "mode=blocknumber"
}
```

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] Unused State Variable

State variable appears to be unused. No analysis has been performed to see if any inline assembly references it. Consider removing this unused variable.

▶ 1 Found Instances

- Found in src/protocol/VaultGuardiansBase.sol Line: 65

```
    uint256 private constant GUARDIAN_FEE = 0.1 ether;
```

[L-3] [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`.

Gas

[G-1] Make the following functions externals instead of public if it isn't used anywhere in the contract.

```
-    function setNotActive() public onlyVaultGuardians isActive { }
-    function rebalanceFunds() public isActive divestThenInvest nonReentrant {}
+    function setNotActive() external onlyVaultGuardians isActive { }
+    function rebalanceFunds() external isActive divestThenInvest nonReentrant {}
```

Informationals

[I-1] Consider using or removing the unused error.

- 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);
```