Vault Guardians Protocol Audit Report

Version 1.0

0xl33

December 21, 2024

Vault Guardians Protocol Audit Report

0xl33

December 21, 2024

Prepared by: 0xl33

Table of Contents

- Table of Contents
- Protocol Summary
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    - [H-1] Missing `vgToken` burn in `_quitGuardian` function, allowing any user to take over the DAO, which allows stealing DAO fees and maliciously setting parameters
    - [H-2] In `_becomeTokenGuardian` function, if inputted `token` is USDC, then `token.safeTransferFrom` will try to transfer `10,000,000` USDC from the caller, which will revert and not let anyone become a USDC guardian
    - [H-3] Shares calculation in `VaultShares::deposit` is wrong, severely disrupting the protocol functionality
    - [H-4] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables MEV bots to steal user assets

- Medium
  - [M-1] Potentially incorrect voting period and delay in governor contract, which may affect governance
- Low
  - [L-1] Incorrect vault name and symbol
  - [L-2] Unassigned return value when divesting AAVE funds

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.

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

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	4
Medium	1
Low	2
Total	7

Findings

High

[H-1] Missing vgToken burn in `_quitGuardian` function, allowing any user to take over the DAO, which allows stealing DAO fees and maliciously setting parameters

Description:

In `VaultGuardiansBase::_becomeTokenGuardian` function, `msg.sender` gets minted `vgToken` equal to `s_guardianStakePrice` (by default it is `10e18`). However, in `VaultGuardiansBase::_quitGuardian` function, which is called when an existing guardian wants to quit, `vgToken`s do not get burned, leaving them in control of the caller, which enables a malicious user to repeatedly call `becomeGuardian` and `quitGuardian` to gain a big amount of voting power to control the protocol.

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:

```
1   "sweepErc20s(address)": "942d0ff9",
2   "transferOwnership(address)": "f2fde38b",
3   "updateGuardianAndDaoCut(uint256)": "9e8f72a4",
4   "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(
            "starting malicious guardian vgToken Balance:\t",
            startingMaliciousVGTokenBalance
        );
        console.log(
            "starting regular guardian 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(
            "ending malicious guardian vgToken Balance:\t",
            endingMaliciousVGTokenBalance
        );
        console.log(
            "ending regular guardian vgToken Balance:\t",
            endingRegularVGTokenBalance
        );
    }
```

Run the test with this command: `forge test --mt testDaoTakeover`

As you can see, the malicious guardian acquires `100e18` worth of vgTokens, by repeatedly becoming a guardian and quitting.

Recommended Mitigation: Add a burn operation in `VaultGuardiansBase::_quitGuardian` function, so if a guardian wants to quit, the contract will burn their vgTokens.

[H-2] In `_becomeTokenGuardian` function, if inputted `token` is USDC, then `token.safeTransferFrom` will try to transfer `10,000,000` USDC from the caller, which will revert and not let anyone become a USDC guardian

Description: When an existing WETH guardian wants to become a USDC guardian, they must call `VaultGuardiansBase::becomeTokenGuardian` and have `s_guardianStakePrice` worth of the asset in advance. As we can see in `VaultGuardiansBase` contract, `s_guardianStakePrice` is set to `10 ether` by default:

```
        // DAO updatable values
@>      uint256 internal s_guardianStakePrice = 10 ether;
        uint256 internal s_guardianAndDaoCut = 1000;
```

If we convert `10 ether` to a representation in decimals, it would be `10e18`, or in wei: `10000000000000000000`. The issue here is that the current implementation of USDC has 6 decimals, not 18, so if we convert `10 ether` to the amount of USDC, we would get `10,000,000e6` (10 million) USDC. In the context of `_becomeTokenGuardian`, the function is essentially asking the caller to provide 10 million USDC as the `s_guardianStakePrice`, which is ridiculous.

Additionally, there is an issue in the same function, if the inputted `token` is LINK, too. In this case, `s_guardianStakePrice` would be equal to `10` LINK, which, compared to the value of WETH, is much cheaper. At the time of writing this, the price of WETH is `~3350` USD and the price of LINK is `~22.5` USD. This price difference is obvious, which makes becoming a LINK guardian much cheaper.

Impact:

If a user wants to become a USDC guardian, this will most likely never happen, due to the stake requirement being 10 million USDC. This leaves the protocol with only 2 available asset options: WETH and LINK.

Additionally, the price difference of WETH and LINK makes the stake requirement much cheaper for becoming a LINK guardian, which is not fair and, quite frankly, not logical.

Proof of Concept:

We can use basic math to prove this issue:

WETH and LINK tokens have 18 decimals, so: `s_guardianStakePrice` = 10 ether = 10e18 = 10000000000000000000

USDC has 6 decimals, so: 10e6 = 10000000

`s_guardianStakePrice` is represented as 10e18

We can convert from 18 decimals to 6 decimals like so: 10e18 / 10e(18-6) = 10e18 / 10e12 = 10e6 = 10,000,000 (10 million)

10e18 = 10,000,000e6

So `10` of a token with `18` decimals is the same as `10000000` (10 million) of a token with 6 decimals.

Recommended Mitigation:

1. Firstly, the protocol could implement a different version of `s_guardianStakePrice`, specifically for tokens with 6 decimals, such as USDC and USDT, and use that as the stake requirement when a user wants to become a USDC guardian.
2. Secondly, the protocol could implement price oracles, who will provide the ability to compare WETH and LINK prices, and based on that, decide how much WETH or LINK to charge a user as a stake price.
3. Another option would be to accept the stake price as a stablecoin, this way the stake price for all implemented assets would always remain the same, simplifying the process and saving gas.

[H-3] Shares calculation in `VaultShares::deposit` is wrong, severely disrupting the protocol functionality

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, at the time of the check, the assets are invested in Aave and/or 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

Edit the `testWithdraw` function in `VaultSharesTest.t.sol` file like so:

```
    function testWithdraw() public hasGuardian userIsInvested {
        uint256 startingBalance = weth.balanceOf(user);
        uint256 startingSharesBalance = wethVaultShares.balanceOf(user)
            ;
        uint256 amoutToWithdraw = mintAmount;

        vm.prank(user);
        wethVaultShares.withdraw(amoutToWithdraw, user, user);

        assertEq(weth.balanceOf(user), startingBalance +
            amoutToWithdraw);
        assert(wethVaultShares.balanceOf(user) < startingSharesBalance)
            ;
    }
```

Run the test with this command: `forge test --mt testWithdraw`

You will get an output like this:

```
Ran 1 test for test/unit/concrete/VaultSharesTest.t.sol:VaultSharesTest

[FAIL: ERC4626ExceededMaxWithdraw(0x6CA6d1e2D5347Bfab1d91e883F1915560e09129D
, 100000000000000000000 [1e20], 39682539682539682540 [3.968e19])]
testWithdraw()(gas: 3220248)
```

```
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 7.99
ms (2.54ms CPU time)
```

As you can see, we get an `ERC4626ExceededMaxWithdraw` error, with the actual `amoutToWithdraw` being `100000000000000000000` (100e18) and `MaxWithdraw` based on shares being `39682539682539682540` (~39.68e18). This proves that, with the current implementation, users cannot withdraw their whole invested amount of assets, because the shares are calculated incorrectly.

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 automated 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-4] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables MEV bots to steal user assets

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:

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

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`:

```
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: - 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.

Medium

[M-1] Potentially incorrect voting period and delay in governor contract, which may affect governance

Description:

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.

Recommended Mitigation:

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
}
```

Low

[L-1] Incorrect vault name and symbol

Description:

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`.

Recommended Mitigation:

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

Description:

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.

Recommended Mitigation:

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)
        });
}
```