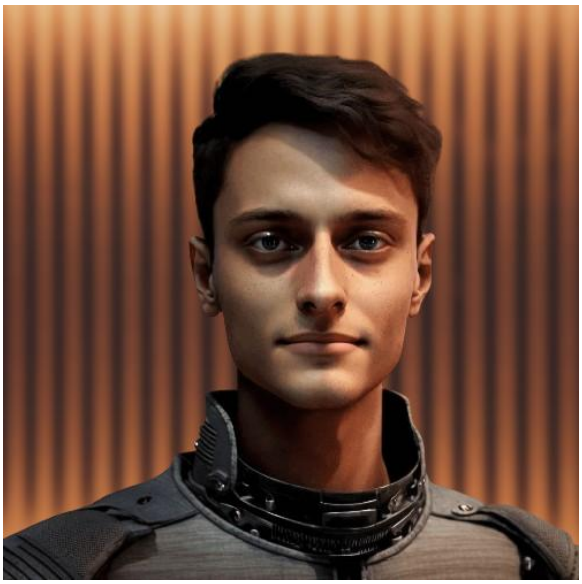

Vault Guardian Audit Report

Version 1.0

x.com/0xAdra

May 21, 2024

Vault Guardian Audit Report

0xAdra

2024-05-21

Prepared by: 0xAdra

Lead Auditors: 0xAdra

Disclaimer

0xAdra made 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 auditor 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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

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
```

```
20 |       #-- UniswapAdapter.sol
21 #-- vendor
22     #-- DataTypes.sol
23     #-- IPool.sol
24     #-- IUniswapV2Factory.sol
25     #-- IUniswapV2Router01.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- Tokens:
  - weth: https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
  - link: https://etherscan.io/token/0x514910771af9ca656af840dff83e8264ecf986ca
  - usdc: https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48

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.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	2

Findings

High

[H-1] Lack of `UniswapV2 slippage protection` in `UniswapAdapter::_uniswapInvest` enables frontrunners to steal profits.

Description: In the PoS mechanism, proposers know well in advance if they will propose one or consecutive blocks ahead of time. In such a scenario, a malicious validator can hold back the transaction and execute it at a more favourable block number.Consider allowing function caller to specify swap deadline input parameter.

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
2      uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens
           (
3          amountOfTokenToSwap,
4  @>      0,
5          s_pathArray,
6          address(this),
7  @>      block.timestamp
8      );
```

Impact: This results in either of the following to happen:

1. 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.

2. 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 `amountOutMin` issue:

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.

For the `deadline` issue:

Add a custom parameter to the `VaultShares::deposit` function so the Vault Guardians protocol can account for the customizations of DeFi projects that it integrates with. Like this:

```
- 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.

[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` check the balance of vault's 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: Include the following code in `VaultSharesTest.t.sol`

```
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.

[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`.

```
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      }
```

Guardians are also free to quit their role at any time by 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:

```
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.

Proof of Code

```
1
2       function testDAOTakeOver() public hasGuardian hasTokenGuardian {
3           address maliciousGuardian = makeAddr("maliciousGuardian");
4           uint256 startingVoterUsdcBalance = usdc.balanceOf(
                maliciousGuardian);
5           uint256 startingVoterWethBalance = weth.balanceOf(
                maliciousGuardian);
6           assertEq(startingVoterUsdcBalance, 0);
7           assertEq(startingVoterWethBalance, 0);
8
9           VaultGuardianGovernor governor = VaultGuardianGovernor(payable(
                vaultGuardians.owner()));
10          VaultGuardianToken vgToken = VaultGuardianToken(address(
                governor.token()));
11
12          //FlashLoan the tokens,or just buy a bunch for 1 block
13          weth.mint(mintAmount,maliciousGuardian);
14          // the same amount as the other guardians
15          uint256 startingMaliciousVGTokenBalance = vgToken.balanceOf(
                maliciousGuardian);
16
17          uint256 startingRegularVGTokenBalance = vgToken.balanceOf(
                guardian);
18
19          console.log("Malicious vgToken Balance:\t",
                startingMaliciousVGTokenBalance);
20          console.log("Regular vgToken Balance:\t",
                startingRegularVGTokenBalance);
21
22          // Malicious Guardian farms token
23          vm.startPrank(maliciousGuardian);
24          weth.approve(address(vaultGuardians),type(uint256).max);
25
26          for(uint256 i;i<10;i++) {
27              address maliciousWethSharesVault = vaultGuardians.
                    becomeGuardian(allocationData);
28
29              IERC20(maliciousWethSharesVault).approve(address(
                    vaultGuardians),
30              IERC20(maliciousWethSharesVault).
31              balanceOf(maliciousGuardian));
32
33              vaultGuardians.quitGuardian();
34          }
35          vm.stopPrank();
36
37          uint256 endingMaliciousVGTokenBalance = vgToken.balanceOf(
                maliciousGuardian);
38
39          uint256 endingRegularVGTokenBalance = vgToken.balanceOf(
```

```
            guardian);

        console.log("Malicious vgToken Balance:\t",
            endingMaliciousVGTokenBalance);
        console.log("Regular vgToken Balance:\t",
            endingRegularVGTokenBalance);
    }
```

```
[PASS] testDAOTakeOver() (gas: 32810004)
Logs:
  Malicious vgToken Balance:     0
  Regular vgToken Balance:       200000000000000000000
  Malicious vgToken Balance:     1000000000000000000000
  Regular vgToken Balance:       200000000000000000000
```

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

Description: The `VaultGuardianGovernor` contract, based on OpenZeppelin Contract's Governor, implements two function to define the voting delay (`votingDelay`) and period (`votingPeriod`). the contract intends to define a voting delay of 1 day., and the voting peroid 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 protocol 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
}
```

```
5
6  function votingPeriod() public pure override returns (uint256) {
7  -   return 7 days;
8  +   return 50400; // 1 week
9  }
```

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` .Consider modifying the function as follows , to avoif errors in off-chain reading these values to identify vaults.

```
1
2  else if (address(token) == address(i_tokenTwo)) {
3      tokenVault =
4      new VaultShares(IVaultShares.ConstructorData({
5          asset: token,
6  -       vaultName: TOKEN_ONE_VAULT_NAME,
7  +       vaultName: TOKEN_TWO_VAULT_NAME,
8  -       vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
9  +       vaultSymbol: TOKEN_TWO_VAULT_SYMBOL,
10         guardian: msg.sender,
11         allocationData: allocationData,
12         aavePool: i_aavePool,
13         uniswapRouter: i_uniswapV2Router,
14         guardianAndDaoCut: s_guardianAndDaoCut,
15         vaultGuardian: address(this),
16         weth: address(i_weth),
17         usdc: address(i_tokenOne)
18     }));
```

Recommended Mitigation: 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)
        });
}
```