Prepared by: Aditya Mishra

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
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

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, 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	2
Medium	1
Low	2
Info	0
Gas	0
Total	6

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] Guardians can infinitely mint `VaultGuardianToken`s 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`

```
    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:

```
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

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] Use of ERC20Permit contract can cause the leakage of data and and also allows the user to change the ERC20 contract tokens which leads to risk of the centralization of the user

Description: Here in so many places there is the use of the ERC20Permit contract which allows the user to perform modification of the ERC20 token.

Impact: After the leakage the user data will be open to everyone and a hacker will change it whenever he wants it to do it.

- Found in src/dao/VaultGuardianToken.sol Line: 9

  ```
  contract VaultGuardianToken is ERC20, ERC20Permit, ERC20Votes, Ownable {
  ```

- Found in src/dao/VaultGuardianToken.sol Line: 21

  ```
      function mint(address to, uint256 amount) external onlyOwner {
  ```

- Found in src/protocol/VaultGuardians.sol Line: 40

  ```
  contract VaultGuardians is Ownable, VaultGuardiansBase {
  ```

Recommended Mitigation: Try to avoid the use of ERC20Permit and use thee ERC20 token.

[L-4] Use of `Ownable` smart contract leads to the centralization at risk

Description: Since in the functions `VaultGuardians::updateGuardianStakePrice` and `VaultGuardians::updateGuardianAndDaoCut` there is the use of the `Ownable::onlyOwner` which creates the problem in the centralization because the owner has full access to these functions and the owner can update the stack price and also the guardians DAo's cut.

Impact: Centralization will be at risk.

- Found in src/protocol/VaultGuardians.sol Line: 71

```
    function updateGuardianStakePrice(uint256 newStakePrice) external
onlyOwner {
```

- Found in src/protocol/VaultGuardians.sol Line: 82

```
    function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
```