Protocol Audit Report

Version 1.0

Mohamed Ibrahim

September 5, 2024

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
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 enables users to deposit specific tokens into a vault overseen by the vaultGuardian. The vaultGuardian's objective is to efficiently manage the vault, aiming to maximize its value for the users who have contributed their tokens

Roles

There are 4 main roles associated with the system.

Role	Description
Vault Guardian DAO	Central governance body managing protocol parameters
DAO Participants	Token holders with voting rights and profit-sharing
Vault Guardians	Professional managers overseeing investments
Investors	Users depositing assets into protocol vaults

- **Vault Guardian DAO**: This organization receives a share of the profits, governed by the `VaultGuardianToken`. The DAO has control over certain protocol parameters, including:
  - `s_guardianStakePrice`
  - `s_guardianAndDaoCut`
  - A percentage of the `ERC20` tokens generated by the protocol
- **DAO Participants**: Holders of the `VaultGuardianToken` who are eligible to vote on protocol decisions and share in the protocol's profits.
- **Vault Guardians**: These strategists or hedge fund managers are responsible for managing the assets within the investable universe. They receive a portion of the revenue generated by the protocol.
- **Investors**: Users who deposit assets into the protocol, earning yield from the investments made by the Vault Guardians.

Issues Found

Severity	Number of issues found
High	3
Medium	1
Low	2
Info	0
Gas	0
Total	6

Severity Findings Summary

Severi	ID	Title	Impact
High	**H-1**	Absence of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` allows frontrunners to exploit and steal profits	Frontrunning exploits can result in significant financial losses for the protocol.
High	**H-2**	`ERC4626::totalAssets` checks the balance of the vault's underlying asset even when invested, resulting in incorrect values	Protocol functionality is disrupted due to incorrect calculations of assets in the vault.
High	**H-3**	Guardians can infinitely mint `VaultGuardianTokens` and take over DAO, stealing fees and maliciously setting parameters	Governance takeover by malicious actors can destabilise the DAO and result in theft of protocol funds.
Mediu	**M-1**	Potentially incorrect voting period and delay in governor may affect governance	Governance mechanisms could operate incorrectly, leading to unintended protocol behaviour.
Low	**L-1**	Incorrect vault name and symbol	Off-chain clients may misidentify vaults, leading to user confusion.
Low	**L-2**	Unassigned return value when divesting AAVE funds	May cause issues in future changes due to incorrect assumptions about return values.

High

[H-1] The absence of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` allows frontrunners to exploit and 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 `UniswapV2Router01` 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:

- Before executing a swap, perform a price check on a data feed, such as a `Chainlink` price feed, and revert the transaction if the exchange rate is too unfavorable.
- Only deposit one side of a Uniswap liquidity pool, without performing the swap itself. If a pool doesn't exist for a specific ERC20 pair or lacks sufficient liquidity, prevent any investment in that pool.

[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` 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 incentivized 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-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`.

```
    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)`
- `transferOwnership(address)`
- `updateGuardianAndDaoCut(uint256)`
- `updateGuardianStakePrice(uint256)`

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()));

        weth.mint(mintAmount, maliciousGuardian);
        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);
        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: ", endingMaliciousVGTokenBalance);
        console.log("Regular vgToken Balance: ", endingRegularVGTokenBalance);
```

```
33      }
```

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 =
```

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

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:

```
1   function _aaveDivest(IERC20 token, uint256 amount) internal returns (uint256
        amountOfAssetReturned) {
2   -       i_aavePool.withdraw({
3   +       amountOfAssetReturned = i_aavePool.withdraw({
4               asset: address(token),
5               amount: amount,
6               to: address(this)
7           });
8   }
```

Risk Assessment Summary

ID	Severity	Title	Status
H-1	High	Slippage Protection	Pending
H-2	High	Asset Calculation	Pending
H-3	High	Token Minting	Pending
M-1	Medium	Voting Period	Pending

ID	Severity	Title	Status
L-1	Low	Vault Naming	Pending
L-2	Low	Return Value	Pending

Recommendations

Priority	Recommendation
Critical	Implement slippage protection
Critical	Redesign asset tracking
Critical	Reform token minting
High	Update governance timing
Medium	Fix naming conventions
Medium	Implement return handling