Lead Auditors:

- Gneissic audits

Table of Contents

Disclaimer

The Gneissic audit 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:

```
Main Branch
```

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
```

```
21 #-- vendor
22     #-- DataTypes.sol
23     #-- IPool.sol
24     #-- IUniswapV2Factory.sol
25     #-- 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	3
Medium	1
Low	1
Info	0
Gas	0
Total	5

High

[H-1] Lack of uniswapv2 slippage protection in the `UniswapAdapter::_uniswapInvest` function, enabling front runners to steal profits

Description: The `UniswapAdapter::_uniswapInvest` function enables users to swap half of it's ERC20 token, enabling the user to investing in both sides of the uniswap pool. This function calls the `swapExactTokensForTokens` which is called from the `IUniswapV2Router01` contract. The `swapExactTokensForTokens` holds two parameters to be aware of:

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

Impact: This will result in either one of the two happening

1. Anyone (e.g a frontrunning bot) sees the transaction in the mempool, frontruns it by taking a quickly taking out a flashloan, tanking the price before the transaction happens which could lead to the protocol executing the swap at unfavourable rates.
2. due to lack of deadline, the node that gets the transaction can hold the transaction in the mempool until it sees that it can profit from it. **Proof of Concept:** When a user calls the `VaultShares::deposit` function, to deposit some amount of ERC20 tokens. If the tokens are to be invested in the uniswap pool, the function later calls the `UniswapAdapter::_uniswapInvest` which then calls the `IUniswapV2Router01::swapExactTokensForTokens`. The `amountOutMin` parameter is set as 0, and the `deadline` parameter is set as `block.timestamp`. This will cause:

1. The node that gets the transaction to hold on to the transaction until it when it is profitable beofore it executes the transaction.
2. Anyone that sees the transaction can take out a flashloan, loans a large amount from the pool and tanks the price of the swap just before it happens.
3. A malicious node will execute the transaction and take most of the profits since the `amountOutmin` is set as 0. This could allow malicious MEV users and frontrunners to drain balances.

Recommended Mitigation: *For the deadline issue* consider adding a custom parameter to the `VaultShares::deposit` function, to allow a deadline and other customized data to be set.

```
- function deposit(uint256 assets, address receiver) publicoverride (
    ERC4626, IERC4626) nonReentrant returns (uint256)

+ function deposit(uint256 assets, address receiver, bytes customData)
    publicoverride (ERC4626, IERC4626) nonReentrant returns (uint256)
```

The custom data will allow for the deadline to be set. *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.

[H-2] Openzeppelin `ERC4626::totalAssets` checks for the vaults underlying assets even after the assets are been invested, leading to wrong returned values.

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 = (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,VaultShares 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 mint guardian tokens, and take over the DAO stealing dao fees and setting parameters.

Description: when a user becomes a guardian, he gets minted with vault guardian tokens (vg tokens). This process is by calling either `VaultGuardianBase::becomeGuardian` or `VaultGuardianBase::becomeTokenGuardian` which executes the `_becomeTokenGuardian` function, which then means them `i_VgTokens`.

```
function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
    private returns (address) {
        s_guardians[msg.sender][token] = IVaultShares(address(
            tokenVault));
        emit GuardianAdded(msg.sender, token);
       @> i_vgToken.mint(msg.sender, s_guardianStakePrice);
        token.safeTransferFrom(msg.sender, address(this),
            s_guardianStakePrice);
        bool succ = token.approve(address(tokenVault),
            s_guardianStakePrice);
        if (!succ) {
            revert VaultGuardiansBase__TransferFailed();
        }
        uint256 shares = tokenVault.deposit(s_guardianStakePrice, msg.
            sender);
        if (shares == 0) {
            revert VaultGuardiansBase__TransferFailed();
        }
        return address(tokenVault);
    }
```

Impact: guardians are also free to call the `quitGuardian` function, which in turn redeems their weth or usdc token to them, but does not burn their vault guardian tokens. This gives them an opening to farm the vault guardian tokens by continously minting them.

Proof of Concept:

1. User becomes vault guardian and is minted vault guardian tokens.
2. User quits and is given given back origninal weth allocation.
3. User becomes guardian with the same weth allocation.
4. user repeatedly mints vault guardian tokens indefinitely.

add the code below in the `VaultGuardianBaseTest.t.sol` and run `forge test --mt testTakeOver -vvv`

code

```
function testTakeOver() public hasGuardian hasTokenGuardian {
        address maliciousGuardian = makeAddr("maliciousGuardian");
        uint256 wethStartingBalanceOfMaliciousGuardian  = weth.
            balanceOf(maliciousGuardian);
        uint256 usdcStartingBalanceOfMaliciousGuardian  = usdc.
            balanceOf(maliciousGuardian);
        assertEq(wethStartingBalanceOfMaliciousGuardian, 0);
        assertEq(usdcStartingBalanceOfMaliciousGuardian, 0);
        VaultGuardianGovernor governor  = VaultGuardianGovernor(payable
            (vaultGuardians.owner()));
        VaultGuardianToken vgToken  = VaultGuardianToken(address(
            governor.token()));
```

```
9           weth.mint(mintAmount, maliciousGuardian);
10          uint256 startingMaliciousVgTokenBalance = vgToken.balanceOf(
                maliciousGuardian);
11          uint256 startingVgTokenOfRegularGuardian  = vgToken.balanceOf(
                guardian);
12          console.log("startingMaliciousVgTokenBalance:",
                startingMaliciousVgTokenBalance) ;
13          console.log("startingRegularVgTokenBalance:",
                startingVgTokenOfRegularGuardian);
14
15          //malicious guardian farms vault guardian tokens
16          vm.startPrank(maliciousGuardian);
17          weth.approve(address(vaultGuardians), type(uint256).max);
18          for (uint256 index = 0; index < 10; index++) {
19              address maliciousWethSharesVault  = vaultGuardians.
                    becomeGuardian(allocationData);
20              IERC20(maliciousWethSharesVault).approve(address(
                    vaultGuardians), IERC20(maliciousWethSharesVault).
                    balanceOf(maliciousGuardian));
21
22          vaultGuardians.quitGuardian();
23          }
24      vm.stopPrank();
25      uint256 endingVgTokenBalanceOfMaliciousUser  = vgToken.balanceOf(
            maliciousGuardian);
26      uint256 endingVgTokenBalanceOfRegularUser  = vgToken.balanceOf(
            guardian);
27      console.log("endingVgTokenBalanceOfMaliciousUser",
            endingVgTokenBalanceOfMaliciousUser);
28      console.log("endingVgTokenBalanceOfRegularUser",
            endingVgTokenBalanceOfRegularUser);
29
30      }
```

Recommended Mitigation:

1. When a user calls the `quitGuardian`, the vault tokens should be burnt
2. Simply don't allocate vgTokens to guardians. Instead, mint the total supply on contract deployment.
3. Mint vgTokens on a vesting schedule after a user becomes a guardian.

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] Use of wrong event in the `VaultGuardians::updateGuardianAndDaoCut()` function

Description: In the `VaultGuardians::updateGuardianAndDaoCut()` function, `VaultGuardians__Upda` event was emitted instead of the `VaultGuardians__UpdatedFee`. This emit a wrong event which can disrupt the protocol.

Impact: If a user listens to the event when the `VaultGuardians::updateGuardianAndDaoCut()` is called, he gets a updatedStakePrice event instead of updatedFee.

Proof of Concept:

```
 function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
      s_guardianAndDaoCut = newCut;

      emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut,
         newCut);
  }
```

Recommended Mitigation: exchange the `VaultGuardians__UpdatedStakePrice` event with the `VaultGuardians__UpdatedFee` event.

```
function updateGuardianAndDaoCut(uint256 newCut) external onlyOwner {
        s_guardianAndDaoCut = newCut;
-        emit VaultGuardians__UpdatedStakePrice(s_guardianAndDaoCut,
    newCut);
```

```
4 +         emit VaultGuardians__UpdatedFee(s_guardianDaoCut, newCut);
5       }
```