Smart Contract Security Audit Report

BossBridge Security Assessment

Lead Auditor: Kevin Lee

September 7, 2025

Abstract

This document presents the findings of a comprehensive security audit conducted on Vault Guardians. The audit focused on identifying vulnerabilities, security flaws, and potential improvements in the smart contract implementation. Our methodology included automated analysis, manual code review, and extensive testing procedures.

Contents

1 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.

You can think of a `vaultGuardian` as a fund manager.

To prevent a vault guardian from running off with the funds, the vault guardians are only allowed to deposit and withdraw the ERC20s into specific protocols.

- Aave v3
- Uniswap v2
- None (just hold)

These 2 protocols (plus "none" makes 3) are known as the "investable universe".

The guardian can move funds in and out of these protocols as much as they like, but they cannot move funds to any other address.

The goal of a vault guardian, is to move funds in and out of these protocols to gain the most yield. Vault guardians charge a performance fee, the better the guardians do, the larger fee they will earn.

Anyone can become a Vault Guardian by depositing a certain amount of the ERC20 into the vault. This is called the `guardian stake`. If a guardian wishes to stop being a guardian, they give out all user deposits and withdraw their guardian stake.

Users can then move their funds between vault managers as they see fit.

The protocol is upgradeable so that if any of the platforms in the investable universe change, or we want to add more, we can do so.

2 Disclaimer

The Kevin Lee 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 person 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.

3 Audit Details

3.1 Scope

`src/abstract/AStaticTokenData.sol`

src/abstract/AStaticUSDCData.sol
src/abstract/AStaticWethData.sol
src/dao/VaultGuardianGovernor.sol
src/dao/VaultGuardianToken.sol
src/interfaces/IVaultData.sol
src/interfaces/IVaultGuardians.sol
src/interfaces/IVaultShares.sol
src/interfaces/InvestableUniverseAdapter.sol
src/protocol/VaultGuardians.sol
src/protocol/VaultGuardiansBase.sol
src/protocol/VaultShares.sol
src/protocol/investableUniverseAdapters/AaveAdapter.sol
src/protocol/investableUniverseAdapters/UniswapAdapter.sol
src/vendor/DataTypes.sol
src/vendor/IPool.sol
src/vendor/IUniswapV2Factory.sol
src/vendor/IUniswapV2Router01.sol

3.2 Roles

3.2.1 User (Depositor)

- Deposits supported ERC20 tokens into a guardian's vault.
- Can withdraw their share of the vault at any time via `redeem` or `withdraw`.
- Free to move funds between different vault guardians.
- Benefits from the guardian's strategy without needing to manage positions directly.

3.2.2 Vault Guardian (Fund Manager)

- Provides a `guardian stake` in order to become eligible to manage a vault.
- Responsible for moving deposited ERC20s within the **investable universe**:
  - Aave v3
  - Uniswap v2
  - None (hold in vault)
- Cannot move funds outside the approved universe.
- Can update strategies and reallocate funds at any time.
- Earns **performance fees** based on the yield generated for depositors.
- May resign by fully returning user deposits and withdrawing their stake.

3.2.3 DAO

- Receives a portion of all guardians' performance fees.
- Governs protocol parameters such as:
  - Pricing models and performance fee structure
  - Expansion or modification of the investable universe
- Incentivizes guardians by distributing DAO tokens for active participation.
- Maintains upgradeability of the protocol to adapt to ecosystem changes.

4 High

4.1 [H-1] The `vg_token` has not been burned when the guardian quits. Malicious user can create a tons of VaultShares to obtain a lots of `vg_token` to control the DAO.

Description: The `vg_token` is not being burned when a guardian quits, allowing malicious users to create numerous VaultShares and obtain a large amount of `vg_token`, potentially giving them control over the DAO.

Impact: This could lead to a situation where a malicious user gains disproportionate influence over the DAO by exploiting the unburned `vg_token`.

Proof of Concept:

```
1      function testVgTokenCannotBurnWhenGuardianQuits() public hasGuardian {
2          console2.log("VGTokenBalanceBefore: ",
               ↪ vaultGuardianToken.balanceOf(guardian));
3          vm.startPrank(guardian);
4          wethVaultShares.approve(address(vaultGuardians), mintAmount);
5          vaultGuardians.quitGuardian();
6          vm.stopPrank();
7          console2.log("VGTokenBalanceAfter: ",
               ↪ vaultGuardianToken.balanceOf(guardian));
8          assertGe(vaultGuardianToken.balanceOf(guardian), 0);
9      }
```

Add these code to `VaultGuardiansBaseTest.t.sol` below `testVgTokenCannotBurnWhenGuardianQuits`, then run `forge test -m "testVgTokenCannotBurnWhenGuardianQuits"` to reproduce the issue.

Recommended Mitigation: Implement a mechanism to burn the `vg_token` when a guardian quits, ensuring that the token supply accurately reflects the number of active guardians.

4.2 [H-2] The protocol don't have a way to handle the AAVE reward.It will decrease the yield of the vault.

Description: The protocol currently lacks a mechanism to manage AAVE rewards, which are crucial for maintaining the vault's yield. Without proper handling, these rewards may be lost or underutilized.

Impact: If AAVE rewards are not properly managed, the vault's yield could decrease, leading to lower returns for users and potentially reducing the overall attractiveness of the vault.

Recommended Mitigation: Add functionality to claim and reinvest AAVE rewards into the vault, ensuring that they contribute to the overall yield.

4.3 [H-3] Guardian can withdraw his stake money and keep vault active. It will cause the vault always active even if the guardian is malicious and do harm to the vault.

Description: The current design allows guardians to withdraw their stake while keeping the vault active. This could be exploited by malicious guardians to harm the vault without losing their stake.

Impact: If a guardian can withdraw their stake and still control the vault, they could potentially act against the best interests of the vault and its users.

Proof of Concept:

```
1      function testWithdrawAndKeepVaultActive() public hasGuardian
           ↪ userIsInvested {
2          AllocationData memory al = AllocationData(1000, 0, 0);
3          weth.mint(mintAmount, guardian);
4          vm.startPrank(guardian);
5          weth.approve(address(vaultGuardians), mintAmount);
6          address wethVault = vaultGuardians.becomeGuardian(al);
7          wethVaultShares = VaultShares(wethVault);
8          vm.stopPrank();
9
10         vm.startPrank(user);
11         weth.mint(100 ether, user);
12         weth.approve(address(wethVaultShares), 100 ether);
13         wethVaultShares.deposit(100 ether, user);
14         vm.stopPrank();
15
16         vm.startPrank(guardian);
17         wethVaultShares.approve(address(wethVaultShares),
               ↪ type(uint256).max);
18         wethVaultShares.withdraw(10 ether, guardian, guardian);
19         assertEq(wethVaultShares.getIsActive(), true);
20         vm.stopPrank();
21     }
```

Add these code to `VaultSharesTest.t.sol` below `testGuardianCannotQuitWhenVaultIsActive`, then run `forge test -m "testWithdrawAndKeepVaultActive"` to reproduce the issue.

Recommended Mitigation: Implement a rule that requires guardians to maintain their stake while the vault is active, preventing them from withdrawing their stake without deactivating the vault.

4.4 [H-4] `UniswapAdapter::_uniswapInvest` lack of slippage protection. It will cause a loss when the price impact is high.

Description: The `_uniswapInvest` function does not implement any slippage protection mechanisms, which means that if the price impact of a swap is high, the actual amount received could be significantly lower than expected.

Impact: Without slippage protection, guardians may experience unexpected losses during token swaps, leading to a decrease in trust and usability of the vault.

Proof of Concept:

```
1           uint256[] memory amounts =
                ↪ i_uniswapRouter.swapExactTokensForTokens({
2               amountIn: amountOfTokenToSwap,
3               amountOutMin: 0,    // Lack of slippage protection here
4               path: s_pathArray,
5               to: address(this),
6               deadline: block.timestamp
7           });
```

Recommended Mitigation: Implement slippage protection by allowing users to set a maximum slippage tolerance when initiating swaps. This can be done by adjusting the `amountOutMin` parameter in the Uniswap router calls based on the user's specified slippage tolerance.

4.5 [H-5] `UniswapAdapter::_uniswapInvest` does not integrate a price oracle, leaving it vulnerable to flashloan-based price manipulation.

Description: The `_uniswapInvest` function does not utilize a price oracle to verify the price of tokens before executing swaps. This makes it vulnerable to flashloan attacks, where a malicious guardian could manipulate the price of a token to their advantage.

Impact: If a guardian can manipulate the price of a token during a swap, they could potentially profit at the expense of the vault and its users.

Recommended Mitigation: Implement a price oracle to verify token prices before executing swaps. This can help ensure that swaps are executed at fair market prices and reduce the risk of flashloan attacks.

5 Medium

5.1 [M-1] The system does not have a way to receive GUARDIAN_FEE and relative funcion is not payable.

Description:

```
function becomeGuardian(AllocationData memory wethAllocationData)
    ↪ external returns (address);
function becomeTokenGuardian(AllocationData memory allocationData, IERC20
    ↪ token) external onlyGuardian(i_weth) returns (address);
```

Impact: The guardian fee is not being collected as intended, which could lead to insufficient funds for the guardian system and potential exploitation by malicious actors.

Recommended Mitigation: Implement a payable function for `becomeGuardian` and `becomeTokenGuardian` to collect the `GUARDIAN_FEE`.

5.2 [M-2] The VaultGuardianBase::_becomeTokenGuardian function incorrectly charge 10 ether in amounts instead of 10 ETH vaule.

Description: The function `_becomeTokenGuardian` charges a fixed amount of 10 ether in the token's smallest unit, which may not equate to 10 ETH in value. This could lead to undercharging or overcharging guardians depending on the token's price.

Impact: Guardians may be charged an incorrect fee, leading to potential financial discrepancies and unfair treatment of guardians and decreasing a level which malicious user become a guardian without enough stake.

Proof of Concept:

```
1     function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
          ↪ private returns (address) {
2         s_guardians[msg.sender][token] =
              ↪ IVaultShares(address(tokenVault));
3         emit GuardianAdded(msg.sender, token);
4
5         i_vgToken.mint(msg.sender, s_guardianStakePrice);
6
7         // this should be weth.safeTransferFrom instead of
              ↪ token.safeTransferFrom
8         token.safeTransferFrom(msg.sender, address(this),
              ↪ s_guardianStakePrice);
9         bool succ = token.approve(address(tokenVault),
              ↪ s_guardianStakePrice);
10        if (!succ) {
11            revert VaultGuardiansBase__TransferFailed();
12        }
13                uint256 shares = tokenVault.deposit(s_guardianStakePrice,
                      ↪ msg.sender);
```

```
        if (shares == 0) {
            revert VaultGuardiansBase__TransferFailed();
        }
        return address(tokenVault);
    }
```

Recommended Mitigation:

Using `weth.safeTransferFrom` replace `token.safeTransferFrom` to ensure the fee is charged in WETH, which is pegged to ETH value.

5.3 [M-3] Uniswap does not exist `WETH/WETH` pool. When the vaultshare underlying asset is WETH, the `divestThenInvest` modifier will fail.

Description: The `divestThenInvest` modifier relies on the existence of a `WETH/WETH` pool on Uniswap to facilitate swaps and liquidity provision. However, such a pool does not exist, which means that any attempts to invest WETH through this function will fail.

Impact: This could lead to failed transactions and a poor user experience, as users will not be able to invest their WETH holdings through the vault.

Proof of Concept:

```
    constructor(ConstructorData memory constructorData)
        ERC4626(constructorData.asset)
        ERC20(constructorData.vaultName, constructorData.vaultSymbol)
        AaveAdapter(constructorData.aavePool)
        UniswapAdapter(constructorData.uniswapRouter,
            ↪ constructorData.weth, constructorData.usdc)
    {
        i_guardian = constructorData.guardian;
        i_guardianAndDaoCut = constructorData.guardianAndDaoCut;
        i_vaultGuardians = constructorData.vaultGuardians;
        s_isActive = true;
        updateHoldingAllocation(constructorData.allocationData);

        // External calls

        // @audit-low WETH/WETH pool is not existed
        i_aaveAToken =
            IERC20(IPool(constructorData.aavePool).getReserveData(address(constructorData.
        i_uniswapLiquidityToken =
            ↪ IERC20(i_uniswapFactory.getPair(address(constructorData.asset),
            ↪ address(i_weth)));
    }
```

Recommended Mitigation: Implement a check to prevent the use of the `divestThenInvest` modifier when the underlying asset is WETH, or provide WETH/ETH

5.4 [M-4] Stake fund should not be used to invest, because the stake fund is not belong to the vault, it belong to the guardian and it is should be a manchenism to ensure the guardian will do the right thing.

Description: The stake fund is not owned by the vault, but rather by the guardian. This creates a potential conflict of interest, as the guardian may not act in the best interest of the vault when making investment decisions.

Impact: If the guardian uses the stake fund for investments, it could lead to misaligned incentives and potential losses for the vault. This undermines the purpose of the stake fund as a mechanism to ensure the guardian acts responsibly.

Proof of Concept:

```
1      function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
           ↪ private returns (address) {
2          s_guardians[msg.sender][token] =
               ↪ IVaultShares(address(tokenVault));
3          emit GuardianAdded(msg.sender, token);
4
5          i_vgToken.mint(msg.sender, s_guardianStakePrice);
6
7          token.safeTransferFrom(msg.sender, address(this),
               ↪ s_guardianStakePrice);
8          bool succ = token.approve(address(tokenVault),
               ↪ s_guardianStakePrice);
9          if (!succ) {
10             revert VaultGuardiansBase__TransferFailed();
11         }
12
13         // @audit-medium stake money should not be used to invest
14         uint256 shares = tokenVault.deposit(s_guardianStakePrice,
               ↪ msg.sender);
15         if (shares == 0) {
16             revert VaultGuardiansBase__TransferFailed();
17         }
18         return address(tokenVault);
19     }
```

Recommended Mitigation: Implement strict controls to prevent the use of the stake fund for investments, ensuring that it remains a tool for the guardian to fulfill their duties without risking the vault's assets.

6 Low

6.1 [L-1] `VaultGuardians::updateGuardianAndDaoCut` emit a wrong event. The event should be `VaultGuardians__UpdatedGuardianAndDaoCut` instead of `VaultGuardians__UpdatedGuardianAndDaoFee`.

Description:

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

Recommended Mitigation:

Add a new event `VaultGuardians__UpdatedGuardianAndDaoCut` and emit it in the function `updateGuardianAndDaoCut`.

6.2 [L-2] Attacker can deposit a ton of money to VaultShares to manipulate the price of shares. It will cause next user deposit but cannot get the right amount of shares.

Description:

Impact:

Proof of Concept:

```
    function testFirstDepositPriceManipulation() public {
        vm.startPrank(guardian);
        weth.mint(1_100_000_0000000000000000 ether, guardian);

        weth.approve(address(vaultGuardians), 100 ether);

        address wethVault = vaultGuardians.becomeGuardian(allocationData);
        weth.approve(wethVault, type(uint256).max);
        VaultShares(wethVault).deposit(10 ether, guardian);

        weth.transfer(wethVault, 1_000_000_0000000000000000 ether);
        vm.stopPrank();

        vm.startPrank(user);
        weth.mint(100 ether, user);
        weth.approve(wethVault, 100 ether);

        VaultShares(wethVault).deposit(100 ether, user);

        uint256 userRedeem =
            ↪ VaultShares(wethVault).redeem(VaultShares(wethVault).balanceOf(user),
            ↪ user, user);
        assertEq(userRedeem, 0);
        console2.log("userRedeem", userRedeem);
```

```
        vm.stopPrank();

        vm.startPrank(guardian);
        VaultShares(wethVault).approve(guardian, type(uint256).max);
        uint256 guardianRedeem =
            ↪ VaultShares(wethVault).redeem(VaultShares(wethVault).balanceOf(guardian),
            ↪ guardian, guardian);
        console2.log("guardianRedeem", guardianRedeem);
        vm.stopPrank();
    }
```

Recommended Mitigation: Add a max limit of deposit amount in a single transaction.

6.3 [L-3] Lack of a mechanism to handle the Uniswap breakdown, it will cause a error in `_uniswapInvest`.

Description: The _uniswapInvest function depends on successful execution of Uniswap swaps/liquidity operations. However, there is no error-handling, fallback, or retry mechanism in place if Uniswap reverts, experiences downtime, or returns unexpected results.

Impact: If the Uniswap protocol experiences a breakdown or failure, the `_uniswapInvest` function may not be able to complete its operations, leading to potential loss of funds or locked assets.

Recommended Mitigation: Using `try-catch` to handle the error.

6.4 [L-4] Reenterancy modifier is not the first modifier of `VaultShares::redeem` function. It may cause reenterancy attack.

Description: In the VaultShares::redeem function, the reentrancy guard modifier (e.g., nonReentrant) is not placed as the first modifier in the function declaration.

Impact: Attackers may exploit the unprotected logic inside modifiers that execute before nonReentrant. This could lead to reentrancy attacks, allowing attackers to manipulate the contract's state or drain funds.

Proof of Concept:

```
function redeem(uint256 shares, address receiver, address owner)
        public
        override(IERC4626, ERC4626)
        divestThenInvest
        nonReentrant
        returns (uint256)
```

Recommended Mitigation: Move `nonReentrant` to be the first modifier of `redeem` function.

6.5 [L-5] In UniwapAdapter, amountOfTokenToSwap + amounts[0] is more asset in vault than expects, may break the allocation.

Description: The calculation of `amountOfTokenToSwap + amounts[0]` in the `_uniswapInvest` function may result in more assets being added to the vault than expected. This could lead to an imbalance in the asset allocation within the vault.

Impact: If the vault's asset allocation is disrupted, it could affect the overall performance and risk profile of the vault, potentially leading to losses for investors.

Proof of Concept:

```
    function testUniswapAllocation() public {
        weth.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        weth.approve(address(vaultGuardians), type(uint256).max);
        address wethVault = vaultGuardians.becomeGuardian(allocationData);
        wethVaultShares = VaultShares(wethVault);
        vm.stopPrank();

        console2.log("UnInvest: ", weth.balanceOf(wethVault));
        console2.log("InvestUniSwapWETH: ",
            ↪ weth.balanceOf(uniswapRouter));
        console2.log("InvestUniSwapUSDC(ValueETH): ",
            ↪ usdc.balanceOf(uniswapRouter) / 4000);
        console2.log("InvestAave: ", weth.balanceOf(aavePool));

    }
```

Add these code to `VaultGuardiansTest.t.sol` below `testGuardianCannotQuitWhenVaultIsActive`, then run `forge test -m "testUniswapAllocation"` to reproduce the issue.

Recommended Mitigation: Implement checks to ensure that the total assets in the vault do not exceed the expected allocation after performing swaps and liquidity additions.

7 Informational

7.1 [I-1] There are interfaces defined but not used.

Description: In `IVaultGuardians.sol` and `IInvestableUniverseAdapter.sol`, interfaces are defined but not utilized within the codebase. This can lead to confusion and unnecessary complexity in the code.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;
interface IVaultGuardians {}
```

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```
4 interface IInvestableUniverseAdapter {
5 // function invest(IERC20 token, uint256 amount) external;
6 // function divest(IERC20 token, uint256 amount) external;
7 }
```