TSwap Audit Report

Version 1.0

Max

December 27, 2025

Protocol Audit Report

Max

12 27, 2025

Prepared by: Max Lead Auditors: - xxxxxxx

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The OriginShift 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

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

Audit Details

Scope

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	3
Info	1
Total	9

Findings

HIGH

[H-1] `VaultShares::divestThenInvest()` Liquidity Removal Followed by Re-Deposit Is Vulnerable to MEV Sandwich Attacks.

Description: `VaultShares::divestThenInvest()` The contract removes liquidity from an AMM pool and immediately performs swaps and liquidity re-deposits within the same transaction.Although the operations are atomic at the contract level, the transaction itself is publicly visible before execution and can be reordered within a block. MEV searchers can front-run and back-run this transaction to manipulate the pool price during execution.

Impact:

- The contract executes swaps at manipulated prices
- Liquidity is re-added at unfavorable ratios
- Value is extracted by MEV bots
- The contract (or vault users) suffer deterministic losses

Recommended Mitigation:

- Do not remove 100% of liquidity before performing swaps
- use off-chain oracle instead of on-chain AMM

[H-2]`UniswapAdapter` Missing Slippage Protection in Liquidity Removal and Swap Enables MEV Extraction.

Description: the `_uniswapDivest` function removes liquidity from Uniswap V2 Pool and subsequently swap the recived token back to target address. `Both removeLiquidity` and `swapExactTokensForTokens` are executed with slippage parameters set to zero (amountAMin = 0, amountBMin = 0, amountOutMin = 0).

```
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       );
```

Impacts:

- Liquidity may be removed at manipulated prices
- Token swaps can execute with extreme slippage
- Vault or protocol funds may suffer consistent losses
- 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.

Prove 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:

- Enforce non-zero `amountAMin` and `amountBMin` when calling `removeLiquidity` *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-3]`UniswapAdapter::_uniswapDivest()`Missing Approving partyToken to the Vault before Swapping partyToken to Vault's Asset. This means that Vualt will never recived actual Asset.

Description: The `_uniswapDivest` function performs a token swap using swapExactTokensForTokens, which requires the Uniswap router to have sufficient allowance for the input token. However, the function relies on a direct approve call immediately before the swap, without resetting the allowance or using a safe approval pattern.

impacts: `_uniswapDivest` always reverts, because of `ERC20InsufficientAllowance`

Recommended Mitigation:

change the `UniswapAddapter.sol`

```
+ counterPartyToken.approve(
+    address(i_uniswapRouter),
+    counterPartyTokenAmount
+ );
```

[H-4] `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);
```

```
18      vm.stopPrank();
19
20      // prints 41.25 ETH
21      console.log(wethVaultShares.totalAssets());
22  }
```

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.

This would take a considerable re-write of the protocol.

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]`AavePoolMockAaveMock` Never Mint aTokens, this is an error of logic

Description: In Aave, depositing collateral (e.g., USDC) results in the minting of a corresponding aToken (aUSDC) to represent the deposited assets. However, the AaveMock implementation does not mint aTokens upon deposit.

[L-2]`UniswapRouterMock::addLiquidity()` Fails to initialize `liquidity`, causing LP-token to always be zero.

Description: In Uniswap V2,the liquidity generated from a user deposit is calculated during `addLiquidity` process, and LP-tokens are minted to the user accordingly. However, this mock contract fails to initialize the liquidity, which means no matter what user deposit, the LP-token is zero forever.

[L-3]`UniswapRouterMock::swapTokensForExactTokens()` use `amountInMax` as the params to transferred, but `amountInMax` is used for slipProtection.

Description: there are a wrong logic in `UniswapRouterMock::swapTokensForExactTokens()`, use `amountInMax` in the actual transfer

GAS

INFO

[I-1]`IInvestableUniverseAdapter` and `IVaultGuardians`interfaces is empty and no one use the interface

Discription there are empty. you would better add some function to them, reprecenting the role of InvestableUniverseAdapter and VaultGuardians