Aegis Vault Security Review

Pashov Audit Group

Conducted by: btk, Said, ZanyBonzy

September 18th - September 20th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **labs-solo/aegis-vault** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Aegis Vault

Aegis Vault automates the conversion of one type of token into another using advanced strategies. It integrates with Uniswap V3 pools to optimize liquidity and allows users to withdraw their funds in different ways depending on their preferences.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - b88e9915b8087aea18b3c7e51b5f5cecab014a8f

fixes review commit hash - 0583052755dd9533f0b2dc1e2a6c16ca4ad3b8a1

Scope

The following smart contracts were in scope of the audit:

- `AegisVault`
- `AegisVaultERC20`
- `AegisVaultFactory`
- `AegisVaultCore`
- `Constants`
- `ERC20Initializable`
- `SymbolLib`
- `UV3Math`
- `PctMath`

7. Executive Summary

Over the course of the security review, btk, Said, ZanyBonzy engaged with Aegis to review Aegis Vault. In this period of time a total of **5** issues were uncovered.

Protocol Summary

Protocol Name	Aegis Vault
Repository	https://github.com/labs-solo/aegis-vault
Date	September 18th - September 20th
Protocol Type	Liquidity management

Findings Count

Severity	**Amount**
Critical	1
High	1
Medium	1
Low	2
Total Findings	**5**

Summary of Findings

ID	Title	Severity	Status
[C-01]	DoS attack on Aegis Vault when staking is enabled	Critical	Resolved
[H-01]	Potential issues that can arise from BGT being reward token	High	Resolved
[M-01]	getStakedAmount can be manipulated	Medium	Resolved
[L-01]	Remove resetICHIVaultApprovals	Low	Resolved
[L-02]	Interaction with uniswap quoter during deposits	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] DoS attack on Aegis Vault when staking is enabled

Severity

Impact: High

Likelihood: High

Description

When Aegis Vault's staking capability is enabled, `preAction` and `postAction` functions are executed within important operations. These actions interact with the Berachain reward vault. `preAction` will unstake, claim rewards, and transfer the rewards to the farming contract, while `postAction` will stake the ICHI shares back into the Berachain reward vault.

However, there is a vulnerability and a dangerous assumption in calling `rewardsVault.exit` within `_unstakeAndClaimRewards`. It assumes that the entire `stakedBalance` is self-staked. If an attacker delegates their stake to the aegis vault, it will cause a DoS when aegis vault attempts to `exit`. This will revert because `_checkSelfStakedBalance` first checks if the caller (vault)'s `_accountInfo[msg.sender].balance` minus `delegateTotalStaked` is sufficient and does not exceed the `amount`, which is equal to `_accountInfo[msg.sender].balance` when called via `exit`. This causes the call to always revert when the caller has a balance delegated by other users.

```
function exit() external nonReentrant {
>>>       uint256 amount = _accountInfo[msg.sender].balance;
>>>       _checkSelfStakedBalance(msg.sender, amount);
          _withdraw(msg.sender, amount);
          _getReward(msg.sender, msg.sender);
    }
```

```
function _checkSelfStakedBalance
      (address account, uint256 amount) internal view {
        unchecked {
            uint256 balance = _accountInfo[account].balance;

                         uint256 delegateTotalStaked = _delegateStake[account].delegat
>>>         uint256 selfStaked = balance - delegateTotalStaked;
>>>         if (selfStaked < amount) InsufficientSelfStake.selector.revertWith
  ();
        }
    }
```

An attacker can easily call `delegateStake` and specify the target Aegis vault as the `account`, increasing the Aegis vault's `delegateTotalStaked`.

```
function delegateStake
      (address account, uint256 amount) external nonReentrant whenNotPaused {
        _stake(account, amount);
        if (account != msg.sender) {
            unchecked {
>>>             DelegateStake storage info = _delegateStake[account];
                uint256 delegateStakedBefore = info.delegateTotalStaked;
                uint256 delegateStakedAfter = delegateStakedBefore + amount;
                // `<=` and `<` are equivalent here but the former is cheaper
                if
                  (delegateStakedAfter <= delegateStakedBefore) DelegateStakedOverflow
>>>             info.delegateTotalStaked = delegateStakedAfter;
                // if the total staked by all delegates doesn't overflow, the
                // following won't
                info.stakedByDelegate[msg.sender] += amount;
            }
        }
        emit DelegateStaked(account, msg.sender, amount);
    }
```

When attempting to disable staking, it will always fail because `setStakingStatus` also calls `_unstakeAndClaimRewards`, causing the Aegis Vault to become frozen.

Recommendations

Consider using `withdraw` by providing `balanceOf - getTotalDelegateStaked` followed by `getReward`, instead of `exit`.

8.2. High Findings

[H-01] Potential issues that can arise from BGT being reward token

Severity

Impact: High

Likelihood: Medium

Description

The expected reward token gotten from staking on berachain is potentially the Bera Governance Token, BGT.

```
/// @notice ERC20 token in which rewards are denominated and distributed.
    ///         Typically the BGT(i.e. Berachain Governance Token)
    function REWARD_TOKEN() external view returns (IERC20);
```

BGT, by design, is not transferrable, except by specified senders. From the docs.

> $BGT is non-transferrable and can only be acquired by providing liquidity in PoL-eligible assets (e.g. liquidity on Bex).

And as can be seen from the token contract, (albeit testnet), the tokens can only be transferred by approved senders. Important to note that users only need to be approved to send the tokens, anyone can receive the token.

```
function transfer(
        address to,
        uint256 amount
    )
        public
        override(IERC20, ERC20Upgradeable)
        onlyApprovedSender(msg.sender)
        checkUnboostedBalance(msg.sender, amount)
        returns (bool)
    {
        return super.transfer(to, amount);
    }
```

Assuming staking is enabled, during deposits, withdrawals and rebalancing, the `_preAction` function is called. The function unstakes from berachain vault and attempts to send to the fee recipient.

```
for (uint256 i = 0; i < rewardsVaults.length; i++) {
                IBerachainRewardsVault rewardsVault = rewardsVaults[i];
                if (address(rewardsVault) != NULL_ADDRESS) {
                    _unstakeAndClaimRewards(rewardsVault); //@note
                    _transferRewardsToFarmingContract
                    //(self, rewardsVault); //@note
                }
```

Here, two issues are likely to arise:

1. The functions that attempt to claim rewards will all fail as although, aegis vault will be able to receive the tokens from BerachainRewardsVaults, it will not be able to transfer it to the farming contract due to not being an approved sender. This is potentially all of the major protocol operations that depend on `_preAction` function, the `collectRewards` function and `setStakingStatus` function.

2. The claimed rewards may be stuck in the farming contract if it's not made a whitelisted sender.

Recommendations

Dealing with this is a bit tricky since it is external admin-dependent. I'd recommend introducing queries for whitelisted status before the tokens are transferred and the farming contract is set. If the contracts are not whitelisted, skip the transfers instead.

For example:

When setting the farming contract, we can insist that only whitelisted farming contracts can be set.

```
function setFarmingContract(address _farmingContract) external override {
        _onlyAdmin();
+       IERC20 rewardToken = rewardsVault.REWARD_TOKEN();
+       require (rewardToken.isWhitelistedSender
+ (_farmingContract), "Not whitelisted");
        if (berachainState.farmingContract != _farmingContract) {
            berachainState.farmingContract = _farmingContract;
            emit FarmingContract(msg.sender, _farmingContract);
        }
    }
```

When handling rewards transfers in `preAction` and `collectRewards`, we can skip sending rewards if the vault itself is not whitelisted.

```
function _transferRewardsToFarmingContract(
      BerachainStatememoryself,
      IBerachainRewardsVaultrewardsVault
    ) private {
           if (self.farmingContract != NULL_ADDRESS) {
           IERC20 rewardToken = rewardsVault.REWARD_TOKEN();
           uint256 rewardBalance = rewardToken.balanceOf(address(this));
-          if (rewardBalance > 0 ) {
+          if (rewardBalance > 0 && rewardToken.isWhitelistedSender(address
+ (this)) {
                rewardToken.safeTransfer(self.farmingContract, rewardBalance);
            }
        }
    }
```

Overall, the fix depends on the protocol's plans for the tokens, but the token's intrinsic property should be taken into consideration.

8.3. Medium Findings

[M-01] `getStakedAmount` can be manipulated

Severity

Impact: Medium

Likelihood: Medium

Description

`getStakedAmount` returns `stakedAmount` of Aegis vault within Berachain reward's vault by calling `balanceOf`.

```
function getStakedAmount
      (IBerachainRewardsVault rewardsVault) private view returns (uint256 stakedAmount
        if (address(rewardsVault) != NULL_ADDRESS) {
            stakedAmount = rewardsVault.balanceOf(address(this));
        }
        /// @dev else stakedAmount defaults to 0
    }
```

This function is called by `_calculateAegisVaultAmountsInICHIVaultIncludingStaked`, which is used in several important getter functions, such as `getUserBalance`, `getDepositPosition`, `getTargetPosition`, and `checkUpkeep`.

An attacker can manipulate this value by calling `delegateStake` within the Berachain reward contract and providing a staked balance to Aegis Vault, causing the returned values from those getters to be inaccurate and not using the actual self-staked balance.

```
function delegateStake
      (address account, uint256 amount) external nonReentrant whenNotPaused {
        _stake(account, amount);
        if (account != msg.sender) {
            unchecked {
                DelegateStake storage info = _delegateStake[account];
                uint256 delegateStakedBefore = info.delegateTotalStaked;
                uint256 delegateStakedAfter = delegateStakedBefore + amount;
                // `<=` and `<` are equivalent here but the former is cheaper
                if
                  (delegateStakedAfter <= delegateStakedBefore) DelegateStakedOverflow
                info.delegateTotalStaked = delegateStakedAfter;
                // if the total staked by all delegates doesn't overflow, the
                // following won't
                info.stakedByDelegate[msg.sender] += amount;
            }
        }
        emit DelegateStaked(account, msg.sender, amount);
    }
```

Recommendations

Modify `getStakedAmount` as follows:

```
function getStakedAmount
      (IBerachainRewardsVault rewardsVault) private view returns (uint256 stakedAmount
        if (address(rewardsVault) != NULL_ADDRESS) {
-             stakedAmount = rewardsVault.balanceOf(address(this));
+             stakedAmount = rewardsVault.balanceOf(address
+ (this)) - rewardsVault.getTotalDelegateStaked(address(this));
        }
        /// @dev else stakedAmount defaults to 0
    }
```

8.4. Low Findings

[L-01] Remove `resetICHIVaultApprovals`

In AegisVaultCore.sol, `resetICHIVaultApprovals` is now open to users, but is still in the IAegisVaultOwnerActions interface. Recommend moving it to `IAegisVaultActions` interface instead.

[L-02] Interaction with uniswap quoter during deposits

AegisVaultCore.sol interacts with uniswap quoter contract. It is used during deposits to calculate how much a user would get if they swapped on the targetVault.

```
import
   { IQuoter } from "@uniswap/v3-periphery/contracts/interfaces/IQuoter.sol";
```

```
function __deposit(
//...
                // figure out how much token0 the user would get if they were to
                // the swap on the targetVault
                uint256 realDeltaOut0 = _quoter.quoteExactInputSingle(
                    token1, token0, targetVault.fee(

                    ), ctx.userDepositAmount1.sub(ctx.userContributionToTotal1
                );
//...
                // figure out how much token1 the user would get if they were to
                // the swap on the targetVault
                uint256 realDeltaOut1 = _quoter.quoteExactInputSingle(
                    token0, token1, targetVault.fee(

                    ), ctx.userDepositAmount0.sub(ctx.userContributionToTotal0
                );
  //...
```

However, Uniswap Quoter can be gas-intensive and is not designed to be used onchain, as it's not very stable which can cause unexpected reversions. It is actually recommended for use in the backend by uniswap. From the docs

> Allows getting the expected amount out or amount in for a given swap without executing the swap

> These functions are not gas efficient and should not be called on chain. Instead, optimistically execute the swap and check the amounts in the callback.

Considering how complex the `deposit` function is, introducing such an unstable external call that can be gas intensive can quickly lead to unexpected reverts, reaching the gas limit, and potentially dossing deposits for users.

To fix this might require a slight redesign.

For instance, using an oracle for token valuations and/or using the output of the real swaps instead of quoting the expected amount as recommended by uniswap.

Aegis team comments

We acknowledge the concerns regarding the gas-intensive nature of Uniswap Quoter interactions during deposits. To mitigate this, we implemented and tested a QuoterView contract, which can be utilized on chains where gas costs are a primary concern. However, in most cases, including Berachain, we will continue to use the standard Uniswap V3 Quoter deployed by the underlying AMM.

Moreover, we have introduced the doCheckImpliedSlippage boolean, which allows bypassing the Quoter entirely in scenarios where all Aegis depositors are known, further optimizing efficiency.