Security Audit Report for AloeBlend Smart Contracts

Date: Jan 24, 2022

Version: 1.1

Contact: contact@blocksecteam.com

Digitally signed by BlockSec Audit Team
Date: 2022.01.25 10:27:54 +08'00'

Contents

1 Introduction **1**

1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 1
1.3 Procedure of Auditing . . . 2
1.3.1 Software Security . . . 2
1.3.2 DeFi Security . . . 2
1.3.3 NFT Security . . . 2
1.3.4 Additional Recommendation . . . 3
1.4 Security Model . . . 3

2 Findings **4**

2.1 Software Security . . . 4
2.1.1 Accumulated maintenance fees can be claimed by invoking `rebalance` twice in a single transaction. . . . 4
2.2 DeFi Security . . . 6
2.2.1 The estimate of cToken may be inaccurate . . . 6
2.3 Additional Recommendation . . . 6
2.3.1 Remove duplicated external call to save gas. . . . 6
2.3.2 Claim governance tokens from other Dapps. . . . 7
2.3.3 Update the interface `IOlympusStaking` to match the latest version. . . . 7
2.3.4 Support deflationary/inflationary tokens. . . . 8
2.3.5 Add the contract `factory` into the audit range. . . . 8
2.3.6 Sort `silo0` and `silo1` when creating vaults. . . . 9
2.4 Additional Comment . . . 9
2.4.1 The potential opportunity to manipulate Uniswap V3 pools . . . 9

Report Manifest

Item	Description
Client	Aloe Labs
Target	AloeBlend Smart Contracts

Version History

Version	Date	Description
1.0	Jan 9, 2022	First Release
1.1	Jan 24, 2022	Add fix status

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The files that are audited in this report include the following ones.

Repo Name	Github URL
AloeBlend	https://github.com/aloelabs/aloe-blend

The auditing process is iterative. Specifically, we will further audit the commits that fix the founding issues. If there are new issues, we will continue this process. Thus, there are multiple commit SHA values referred in this report. The commit SHA values before and after the audit are shown in the following.

Before and during the audit

Project		Commit SHA
AloeBlend	C1	fd1635d8928c74ed24550d3f0d9a63f284a7f872
	C2	a7395fd9e0911c04afceb14858c245f294202dae
	C3	671ab9981d4ab87eba27aa41a737e5716a580453
	C4	1080eae1f0c032c705b4050c5faf726a778d489c
	C5	8367728338da789d4c6e6a09129d317d2f6ff2f7
	C6	0cb955725e50b97beaece9968bf753c6474c2a05

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data Flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find two potential issues in the smart contract. We also have six recommendations, as follows:

- High Risk: 0
- Medium Risk: 0
- Low Risk: 2
- Recommendations: 6
- Additional comment: 1

ID	Severity	Description	Category	Status
1	Low	*Accumulated maintenance fees can be claimed by invoking* `rebalance` *twice in a single transaction.*	Software Security	Fixed
2	Low	*The estimate of cToken may be inaccurate*	DeFi Security	Fixed
3	-	*Remove duplicated external call to save gas*	Recommendation	Fixed
4	-	*Claim governance tokens from other Dapps*	Recommendation	Fixed
5	-	*Update the interface* `IOlympusStaking` *to match the latest version*	Recommendation	Fixed
6	-	*Support deflationary/inflationary tokens*	Recommendation	Confirmed
7	-	*Add the contract* `factory` *into the audit range*	Recommendation	Fixed
8	-	*Sort* `silo0` *and* `silo1` *when creating vaults*	Recommendation	Undetermined
9	-	*The potential opportunity to manipulate Uniswap V3 pools*	Additional comment	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Accumulated maintenance fees can be claimed by invoking `rebalance` twice in a single transaction.

Status Fixed.

Description

The AloeBlend protocol incentivizes users to invoke the function `rebalance` by distributing maintenance fees to them.

The `rebalance` will trigger the private function `recenter`, which updates the variable `recenterTimestamp`, as shown in the below code snippet.

```
359    function recenter(
360        RebalanceCache memory cache,
361        uint256 inventory0,
362        uint256 inventory1
363    ) private {
364        ......
365        recenterTimestamp = block.timestamp;
366        emit Recenter(_primary.lower, _primary.upper, cache.magic);
```

```
367     }
```

The function `getRebalanceUrgency` returns a multiplier cached in `cache.urgency`, which grows with the time since last `recenter`, as shown in the below code snippet.

```
157     /// @inheritdoc IAloeBlendDerivedState
158     function getRebalanceUrgency() public view returns (uint32 urgency) {
159         urgency = uint32(FullMath.mulDiv(10_000, block.timestamp - recenterTimestamp, 24 hours));
160     }
```

```
254     /// @inheritdoc IAloeBlendActions
255     function rebalance(uint8 rewardToken) external nonReentrant {
256         uint32 gasStart = uint32(gasleft());
257         RebalanceCache memory cache;
258
259         // Get current tick & price
260         (cache.sqrtPriceX96, cache.tick, , , , , ) = UNI_POOL.slot0();
261         cache.priceX96 = uint224(FullMath.mulDiv(cache.sqrtPriceX96, cache.sqrtPriceX96, Q96));
262         // Get rebalance urgency (based on time elapsed since previous rebalance)
263         cache.urgency = getRebalanceUrgency();
264         ......
265     }
```

According to the above code, if a user invokes the function `rebalance` twice in a single transaction, the `cache.urgency` will be set to zero.

As shown in the below code snippet, the variable `rewardPerGas` will be to zero when the `cache.urgency` is zero, and all the maintenance fees will be transferred to the user who invokes the function `rebalance` (line 329 line 332).

```
325     // computations
326     uint224 rewardPerGas = uint224(FullMath.mulDiv(rewardPerGas0Accumulator, cache.urgency, 10_000
            ));
327     uint256 rebalanceIncentive = gasUsed * rewardPerGas;
328     // constraints
329     if (rewardPerGas == 0 || rebalanceIncentive > maintenanceBudget0)
330         rebalanceIncentive = maintenanceBudget0;
331     // payout
332     TOKEN0.safeTransfer(msg.sender, rebalanceIncentive);
```

Impact When the maintenance fee accumulates, an attacker can get all the maintenance fee by invoking the function `rebalance` twice in a single transaction.

Suggestion Do not allow the execution of the function `recenter` twice in a single block.

Feedback from the project The goal here is to determine whether the stored reward per gas value is 0. This will be the case during the first rebalance, and could be the case later on if someone calls rebalance 10 times in a single block. You're correct to point out that what we're actually checking is whether the stored value is 0 or the urgency is 0 (since they're multiplied together). We're addressing this by moving the urgency multiplication elsewhere.

This issue was fixed by commit `C3`.

2.2 DeFi Security

2.2.1 The estimate of cToken may be inaccurate

Status Fixed.

Description

In the `CompoundCEtherSilo` contracts (also exists in other contracts), the function `withdraw` is responsible for withdrawing assets from Compound, as shown in the below code snippet.

```
48    function withdraw(uint256 amount) external override {
49        if (amount == 0) return;
50        uint256 cAmount = 1 + FullMath.mulDiv(amount, 1e18, cEther.exchangeRateStored());
51
52        require(cEther.redeem(cAmount) == 0, "Compound: redeem ETH failed");
53        WETH.deposit{value: amount}();
54    }
```

Listing 2.1: CompoundCEtherSilo.sol

We notice that the function `withdraw` uses the `cEther.exchangeRateStored()` to estimate the amount of cEther to be redeemed. However, Compound always updates the `exchangeRate` at the beginning when executing the function `redeem`. Therefore, the estimated amount may be inaccurate.

Impact The invocation of the function `withdraw` may be reverted when the reserve is insufficient.

Suggestion Poke compound (update the `exchangeRate` in Compound) before interacting with Compound.

Feedback from the project We're accepting the suggestion to poke Compound before interacting with it further.

This issue was fixed by commit `C2`.

2.3 Additional Recommendation

2.3.1 Remove duplicated external call to save gas.

Status Acknowledged.

Description There is a duplicated function call in the contract `AloeBlend`. The function `deposit` observes the price from Uniswap twice. The unique call path for these function is as follows:

$deposit \rightarrow \mathbf{uni\_pool.slot0} \rightarrow \_computeLPShares \rightarrow getInventory \rightarrow \mathbf{uni\_pool.slot0} \rightarrow \_getDetailInventory$

```
129    function getInventory() public view returns (uint256 inventory0, uint256 inventory1) {
130        (uint160 sqrtPriceX96, , , , , , ) = UNI_POOL.slot0();
131        (inventory0, inventory1, , ) = _getDetailedInventory(sqrtPriceX96, true);
132    }
133%
```

Listing 2.2: AloeBlend.sol

Impact NA

Suggestion The price from Uniswap can be transferred as a parameter from the upper function `deposit`

Feedback from the project We've refactored the code to incorporate this suggestion and further improve gas efficiency.

This issue was fixed by commit `C4`.

2.3.2 Claim governance tokens from other Dapps.

Status Fixed.

Description

The protocol invests assets to other Dapps, such as Compound, Olympus, and Fuse. However, it does not claim the governance tokens from them.

Impact Part of the benefits is wasted.

Suggestion Add codes to claim these governance tokens.

Feedback from the project Good point. We've added code that allows these tokens to be claimed as part of the rebalance incentive, thus reducing load on 'maintenanceBudget0' and 'maintenanceBudget1'. The requisite reward per gas values is tracked in the same way that they are for token0 and token1. In the case of Compound, 'claimComp' can be called on another address' behalf, so no updates to the silo are required. We will add special claiming logic to future silos if necessary.

This issue was fixed by commit `C3`.

2.3.3 Update the interface `IOlympusStaking` to match the latest version.

Status Fixed.

Description In the contract `OlympusStakingSilo.sol`, the functions invoked in the function `OlympusStaking` including `unstake()` or `claim()` do not match the latest version of the Interfaces committed in the Github [1].

```
 8  interface IOlympusStaking {
 9    function claim(address _recipient) external;
10
11    function stake(uint256 _amount, address _recipient) external returns (bool);
12
13    function unstake(uint256 _amount, bool _trigger) external;
14
15    function OHM() external view returns (address);
16
17    function sOHM() external view returns (address);
18  }
```

Listing 2.3: OlympusStakingSilo.sol

Impact NA.

Suggestion Update the `IOlympusStaking`.

[1] https://github.com/OlympusDAO/olympus-contracts/blob/main/contracts/interfaces/IStaking.sol, 3bb3605195579a76ee6c06092756638847009 7d5

Feedback from the project Thanks for bringing this to our attention. We will update to the latest interface soon.

This issue was fixed by commit `C5`.

2.3.4 Support deflationary/inflationary tokens.

Status Confirmed.

Description The deflationary and inflationary tokens are not supported in the current implementation. Specifically, the actual transferred amount to the pool may be different from the value specified in the `transfer` function. The following code shows an example.

```
186    (uint160 sqrtPriceX96, , , , , , ) = UNI_POOL.slot0();
187    uint224 priceX96 = uint224(FullMath.mulDiv(sqrtPriceX96, sqrtPriceX96, Q96));
188
189    (shares, amount0, amount1) = _computeLPShares(amount0Max, amount1Max, priceX96);
190    require(shares != 0, "Aloe: 0 shares");
191    require(amount0 >= amount0Min, "Aloe: amount0 too low");
192    require(amount1 >= amount1Min, "Aloe: amount1 too low");
193
194    // Pull in tokens from sender
195    TOKEN0.safeTransferFrom(msg.sender, address(this), amount0);
196    TOKEN1.safeTransferFrom(msg.sender, address(this), amount1);
```

Listing 2.4: AloeBlend.sol

Impact NA

Suggestion Check the balance after receiving tokens.

Feedback from the project The Uniswap v3 periphery contracts also use `safeTransferFrom`. We feel that it is fine to make the same assumption, but if we want fee-on-transfer compatibility in the future, we can deploy modified contracts.

2.3.5 Add the contract `factory` into the audit range.

Status Fixed.

Description In the constructor function of the `AloeBlend` contract, there exists an external function call from the function of msg.sender. However, only the interface is provided. The specified implementation is not included in the AloeBlend project, thus is not audited. The possible issues in the implementation are unclear.

```
119    volatilityOracle = IFactory(msg.sender).VOLATILITY_ORACLE();
```

Listing 2.5: AloeBlend.sol

```
6    interface IFactory {
7        /// @notice The address of the volatility oracle
8        function VOLATILITY_ORACLE() external view returns (IVolatilityOracle);
9    }
```

Listing 2.6: IFactory.sol

Impact NA

Suggestion Add the contract `factory` into the audit range.

Feedback from the project The factory contract will be included in future audits.

2.3.6 Sort `silo0` and `silo1` when creating vaults.

Status Undetermined.

Description

The order of `silo0` and `silo1` is not be forced to be the same as the corresponding UniswapV3 pool, which may make some functions act unexpectedly.

```
54    function createVault(
55        IUniswapV3Pool pool,
56        ISilo silo0,
57        ISilo silo1
58    ) external returns (IAloeBlend vault) {
59        bytes memory constructorArgs = abi.encode(pool, silo0, silo1);
60        bytes32 salt = keccak256(abi.encode(pool, silo0, silo1));
61        vault = IAloeBlend(super._create(constructorArgs, salt));
62
63        getVault[pool][silo0][silo1] = vault;
64        didCreateVault[vault] = true;
65
66        emit CreateVault(vault);
67    }
```

Listing 2.7: Factory.sol

Impact Invalid pools are created when the project developers use the wrong order of `silo0` and `silo1`

Suggestion Sort `silo0` and `silo1` when creating vaults.

This recommendation is made to the commit `C5`.

2.4 Additional Comment

2.4.1 The potential opportunity to manipulate Uniswap V3 pools

Most of the TVLs in AloeBlend Smart Contracts are used to add liquidity to Uniswap V3 pools.

As shown in the the following code at L179 of the contract `AloeBlend`, the function `deposit` uses *the real-time price of Uniswap V3 pools*. The codes come from the commit `C6`.

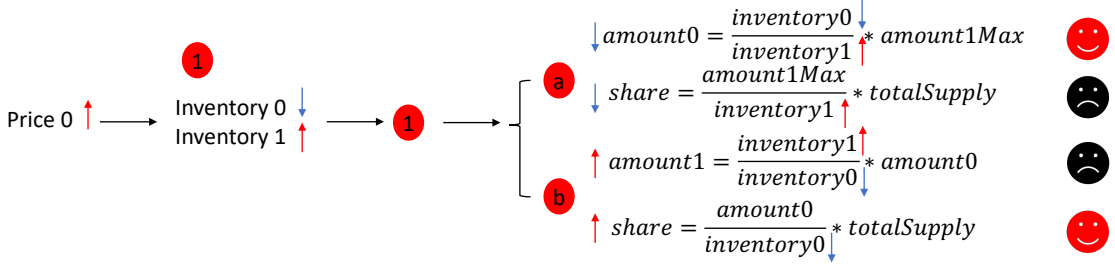
```
179   function deposit(
180       uint256 amount0Max,
181       uint256 amount1Max,
182       uint256 amount0Min,
183       uint256 amount1Min
184   )
185       external
186       returns (
187           uint256 shares,
188           uint256 amount0,
```

```
189             uint256 amount1
190         )
191     {
192         require(amount0Max != 0 || amount1Max != 0, "Aloe: 0 deposit");
193         // Reentrancy guard is embedded in `_loadPackedSlot` to save gas
194         (Uniswap.Position memory primary, Uniswap.Position memory limit, , ) = _loadPackedSlot();
195         packedSlot.locked = true;
196
197         // Poke all assets
198         primary.poke();
199         limit.poke();
200         silo0.delegate_poke();
201         silo1.delegate_poke();
202
203         (uint160 sqrtPriceX96, , , , , , ) = UNI_POOL.slot0();
204         (uint256 inventory0, uint256 inventory1, ) = _getInventory(primary, limit, sqrtPriceX96,
                true);
205         (shares, amount0, amount1) = _computeLPShares(
206             totalSupply,
207             inventory0,
208             inventory1,
209             amount0Max,
210             amount1Max,
211             sqrtPriceX96
212         );
```

Listing 2.8: AloeBlend.sol

```
} else {
    // The branches of this ternary are logically identical, but must be separate to avoid overflow
    bool cond = _inventory0 < _inventory1
        ? FullMath.mulDiv(_amount1Max, _inventory0, _inventory1) < _amount0Max   1
        : _amount1Max < FullMath.mulDiv(_amount0Max, _inventory1, _inventory0);   2

    if (cond) {
        amount1 = _amount1Max;
        amount0 = FullMath.mulDiv(amount1, _inventory0, _inventory1);
        shares = FullMath.mulDiv(amount1, _totalSupply, _inventory1);          a
    } else {
        amount0 = _amount0Max;
        amount1 = FullMath.mulDiv(amount0, _inventory1, _inventory0);
        shares = FullMath.mulDiv(amount0, _totalSupply, _inventory0);          b
    }
}   Hayden Shively [11 days ago] • Rearrange functions in AloeBlend (#13)
```



Flashloan to make price 0 up, and the situation of manipulating price 1 is the same

Figure 2.1: Price manipulation analysis

In particular, the function `deposit` uses the real-time price to calculate the number of token0 and token1 the protocol owns in the Uniswap V3 pool: `inventory0` and `inventory1` (assuming the protocol only

deposits in the Uniswap V3 pool). After that, the function `deposit` uses the `inventory0` and `inventory1` to determine the number of shares that should be minted to users. Since the `inventory0` and `inventory1` can be manipulated via Flash Loan, during the audit, we tried to understand whether the price manipulation attack can be launched by an the attacker to make profit, i.e., by depositing less tokens but getting more shares than he/she should get.

Specifically, the follow-up impact is shown in the figure 2.1 if the attacker lifts the token0's price (using the Flash Loan) of the Uniswap V3 pool. Although the branch `1->b` can mint more shares, it requires more token1 to be deposited. But the increment of the returned share is less than the amount of token1 (amount1) that will be deposited into the protocol (see the formula 1.b in Figure 2.1).

On the contrary, the branch `1->a` seems to be profitable. That's because though the number of share decreases, the amount0 (number of token0 needs to be deposited) also decreases. We built a local environment using brownie to verify this attack surface. We performed several experiments using different settings but cannot make any profit.

With this being said, we highly suggest that the project monitors the protocol's real-time status after the launch, since the protocol's real-time status space is much bigger than the one that can be reached through static code auditing. If any abnormal status is detected, the protocol can be paused for further emergency response.