Security Audit Report for Aloell core

Date: Aug 22, 2023

Version: 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	AloeII
Target	AloeII core

Version History

Version	Date	Description
1.0	Aug 22, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec 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 successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. 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 target of this audit is the code repo of the core contracts in AloeII project. The AloeII project is a lending and leverage borrowing protocol for participating in UniswapV3 farming. Please note that the contracts covered by this audit do not involve the periphery contracts.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (Version 1), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
AloeII-Core	Version 1	384a582c0d32a4254f03ec1f2c6c9952c7235a62
	Version 2	e2d4ba2339372426de9ce6f58e6a656e66f22e8b

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 does 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
* Permission management
* 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]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

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

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item 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 **five** potential issues. Besides, we have **one** recommendation and **four** notes.

- High Risk: 2
- Medium Risk: 1
- Low Risk: 2
- Recommendation: 1
- Note: 4

ID	Severity	Description	Category	Status
1	High	Unverified lenders in the `claimRewards` function	Software Security	Fixed
2	High	Precision loss in the `liquidate` function	Software Security	Fixed
3	Medium	Potential read-only reentrancy	DeFi Security	Fixed
4	Low	Potential DoS attack on the oracle	DeFi Security	Fixed
5	Low	Manipulatable implied volatility	DeFi Security	Confirmed
6	-	Avoid precision loss	Recommendation	Fixed
7	-	Potential cost-free minting due to uncredited balances	Note	-
8	-	The protocol will not support deflation/inflation tokens	Note	-
9	-	The protocol will not support double-entry tokens	Note	-
10	-	All pools are `UniswapV3` pools	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Unverified lenders in the `claimRewards` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Factory` contract, there is a `claimRewards` function that claims user rewards in multiple `Lender` contracts. However, the lenders are not verified, which leads to an untrusted external call issue. Specifically, a malicious user can pass a contract that is in his control to this function as one of the lenders and makes the return value of the `claimRewards` function of the controlled to be large enough for draining all reward tokens in this contract.

```
function claimRewards(Lender[] calldata lenders, address beneficiary) external returns (uint256 earned) {
    // Couriers cannot claim rewards because the accounting isn't quite correct for them -- we save gas
    // by omitting a `Rewards.updateUserState` call for the courier in `Lender._burn`
    require(!isCourier[msg.sender]);

```

```
    unchecked {
        uint256 count = lenders.length;
        for (uint256 i = 0; i < count; i++) {
            earned += lenders[i].claimRewards(msg.sender);
        }
    }

    REWARDS_TOKEN.safeTransfer(beneficiary, earned);
}
```

Listing 2.1: Factory.sol

Impact All reward tokens could be drained in this contract.

Suggestion Add checks on the `lenders` in the `claimRewards` function.

2.1.2 Precision loss in the `liquidate` function

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In the `Borrower` contract, there's a `liquidate` function that allows the liquidation of this contract's positions. For simplicity, let's assume a contract eligible for liquidation borrows token0 and only holds token1. A liquidator is incentivized with tokens to facilitate the swap of token1 into token0 to repay the debts. However, a precision loss issue arises during this process. Specifically, the debts are represented in token0 as `liabilities0`, and the incentives are represented in token1 as `incentive1`, as demonstrated in the following code segment.

```
unchecked {
    // Figure out what portion of liabilities can be repaid using existing assets
    uint256 repayable0 = Math.min(liabilities0, TOKEN0.balanceOf(address(this)));
    uint256 repayable1 = Math.min(liabilities1, TOKEN1.balanceOf(address(this)));

    // See what remains (similar to "shortfall" in BalanceSheet)
    liabilities0 -= repayable0;
    liabilities1 -= repayable1;

    if (liabilities0 + liabilities1 == 0 || (liabilities0 > 0 && liabilities1 > 0)) {
        // If both are zero or neither is zero, there's nothing more to do.
        // Callbacks/swaps won't help.
        incentive1 = 0;
    } else if (liabilities0 > 0) {
        uint256 unleashTime = slot0_ >> 160;
        require(0 < unleashTime && unleashTime < block.timestamp, "Aloe: grace");

        liabilities0 /= strain;
        incentive1 /= strain;

        // NOTE: This value is not constrained to `TOKEN1.balanceOf(address(this))`, so
            liquidators
```

```
        // are responsible for setting `strain` such that the transfer doesn't revert. This shouldn't
        // be an issue unless the borrower has already started accruing bad debt.
        uint256 available1 = mulDiv96(liabilities0, priceX96) + incentive1;

        TOKEN1.safeTransfer(address(callee), available1);
        callee.swap1For0(data, available1, liabilities0);

        repayable0 += liabilities0;
```

Listing 2.2: Borrower.sol

As the decimals of `token0` and `token1` can differ (e.g., 1e18 and 1e6), the `/=` operations on `liabilities0` and `incentive1` may truncate for one but not the other. The precision loss resulting from this truncation might lead to a situation where `liabilities0` becomes zero while the `incentives1` remains non-zero. In such a case, the liquidator could acquire the incentive tokens at no cost.

Impact The liquidator could get the incentive tokens for free.

Suggestion Unify the decimals of the `liabilities0` and the `incentive1`.

2.2 DeFi Security

2.2.1 Potential read-only reentrancy

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Lender` contract, there is a `flash` function to provide flashloans to users. To prevent reentrancy attacks, the function sets the `lastAccrualTime` to 0 as a reentrancy lock.

```
    function flash(uint256 amount, IFlashBorrower to, bytes calldata data) external {
        // Guard against reentrancy
        uint32 lastAccrualTime_ = lastAccrualTime;
        require(lastAccrualTime_ != 0, "Aloe: locked");
        lastAccrualTime = 0;

        ERC20 asset_ = asset();

        uint256 balance = asset_.balanceOf(address(this));
        asset_.safeTransfer(address(to), amount);
        to.onFlashLoan(msg.sender, amount, data);
        require(balance <= asset_.balanceOf(address(this)), "Aloe: insufficient pre-pay");

        lastAccrualTime = lastAccrualTime_;
    }
```

Listing 2.3: Lender.sol

However, this can lead to potential read-only reentrancy. For example, the `_previewInterest` function relies on the `lastAccrualTime`. Further, there are plenty of view functions (e.g., totalAssets) relying on the result of `_previewInterest`. Once there is any protocol relies on those functions, the protocol may suffer from read-only reentrancy attacks.

```
function _previewInterest(Cache memory cache) internal view returns (Cache memory, uint256, uint256) {
    unchecked {
        uint256 oldBorrows = (cache.borrowBase * cache.borrowIndex) / BORROWS_SCALER;
        uint256 oldInventory = cache.lastBalance + oldBorrows;

        if (cache.lastAccrualTime == block.timestamp || oldBorrows == 0) {
            return (cache, oldInventory, cache.totalSupply);
        }

        uint8 rf = reserveFactor;
        uint256 accrualFactor = rateModel.getAccrualFactor({
            utilization: (1e18 * oldBorrows) / oldInventory,
            dt: block.timestamp - cache.lastAccrualTime
        });

        cache.borrowIndex = (cache.borrowIndex * accrualFactor) / ONE;
        cache.lastAccrualTime = 0; // 0 in storage means locked to reentrancy; 0 in `cache` means `borrowIndex` was updated

        uint256 newInventory = cache.lastBalance + (cache.borrowBase * cache.borrowIndex) / BORROWS_SCALER;
        uint256 newTotalSupply = Math.mulDiv(
            cache.totalSupply,
            newInventory,
            newInventory - (newInventory - oldInventory) / rf
        );
        return (cache, newInventory, newTotalSupply);
    }
}
```

Listing 2.4: Ledger.sol

```

function totalAssets() external view returns (uint256) {
    (, uint256 inventory, ) = _previewInterest(_getCache());
    return inventory;
}
```

Listing 2.5: Ledger.sol

Impact The protocol may suffer from read-only reentrancy attacks.

Suggestion Check the reentrancy lock in view functions.

2.2.2 Potential DoS attack on the oracle

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `VolatilityOracle` contract uses the `UniswapV3` oracle, which relies on a series of archived historical price records in `UniswapV3` pools to calculate the Time-Weighted Average Price (TWAP). However, in blockchain networks known for fast block generation, such as Arbitrum, a potential vulnerability exists. Malicious actors could inject a large volume of prices within a short period, potentially triggering a Denial of Service (DoS) attack on the oracle.

Impact The protocol may suffer from DoS attack on the oracle.

Suggestion N/A

Feedback from the Project The check for oracle cardinality is changed to be stricter, besides, interested parties can expand the slot dynamically by off-chain monitoring.

2.2.3 Manipulatable implied volatility

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `BalanceSheet` library uses a prediction mechanism to evaluate the health of a borrower's position. A vital variable, `iv` (i.e., implied volatility), is responsible for calculating a prediction range of the token price, which subsequently influences the calculation of the health factor. However, `iv` is susceptible to manipulation, as illustrated in the following code snippet.

```
// Populate remaining ‘PoolData‘ fields
data.oracleLookback = UNISWAP_AVG_WINDOW;
data.tickLiquidity = pool.liquidity();

// Populate ‘FeeGrowthGlobals‘
Volatility.FeeGrowthGlobals[FEE_GROWTH_ARRAY_LENGTH] storage arr = feeGrowthGlobals[pool];
Volatility.FeeGrowthGlobals memory a = _getFeeGrowthGlobalsOld(arr, lastWrite.index);
Volatility.FeeGrowthGlobals memory b = _getFeeGrowthGlobalsNow(pool);

// Default to using the existing IV
uint256 iv = lastWrite.iv;
// Only update IV if the feeGrowthGlobals samples are approximately ‘FEE_GROWTH_AVG_WINDOW‘
    hours apart
if (
    _isInInterval({
        min: FEE_GROWTH_AVG_WINDOW - 3 * FEE_GROWTH_SAMPLE_PERIOD,
        x: b.timestamp - a.timestamp,
        max: FEE_GROWTH_AVG_WINDOW + 3 * FEE_GROWTH_SAMPLE_PERIOD
    })
) {
    // Estimate, then clamp so it lies within [previous - maxChange, previous + maxChange]
    iv = Volatility.estimate(cachedMetadata[pool], data, a, b, IV_SCALE);
```

Listing 2.6: VolatilityOracle.sol

```
function estimate(
    PoolMetadata memory metadata,
    Oracle.PoolData memory data,
    FeeGrowthGlobals memory a,
```

```
        FeeGrowthGlobals memory b,
        uint256 scale
    ) internal pure returns (uint256) {
        unchecked {
            if (data.secondsPerLiquidityX128 == 0 || b.timestamp - a.timestamp == 0) return 0;

            uint128 revenue0Gamma1 = computeRevenueGamma(
                a.feeGrowthGlobal0X128,
                b.feeGrowthGlobal0X128,
                data.secondsPerLiquidityX128,
                data.oracleLookback,
                metadata.gamma1
            );
            uint128 revenue1Gamma0 = computeRevenueGamma(
                a.feeGrowthGlobal1X128,
                b.feeGrowthGlobal1X128,
                data.secondsPerLiquidityX128,
                data.oracleLookback,
                metadata.gamma0
            );
            // This is an approximation. Ideally the fees earned during each swap would be multiplied by the price
            // *at that swap*. But for prices simulated with GBM and swap sizes either normally or uniformly distributed,
            // the error you get from using geometric mean price is <1% even with high drift and volatility.
            uint256 volumeGamma0Gamma1 = revenue1Gamma0 + amount0ToAmount1(revenue0Gamma1, data.sqrtMeanPriceX96);

            // Fits in uint128
            uint256 sqrtTickTVLX32 = FixedPointMathLib.sqrt(
                computeTickTVLX64(metadata.tickSpacing, data.currentTick, data.sqrtPriceX96, data.tickLiquidity)
            );
            if (sqrtTickTVLX32 == 0) return 0;

            // Fits in uint48
            uint256 timeAdjustmentX32 = FixedPointMathLib.sqrt((scale << 64) / (b.timestamp - a.timestamp));
            return (uint256(2e18) * timeAdjustmentX32 * FixedPointMathLib.sqrt(volumeGamma0Gamma1)) / sqrtTickTVLX32;
        }
    }
```

Listing 2.7: Volatility.sol

The calculation of the new `iv` depends on `data.tickLiquidity`, which is the return value of the `pool.liquidity` function. A malicious user could add or remove liquidity at the current tick to manipulate this value, and consequently, the new `iv`. Even though the `iv` is confined within a pre-set range, potential manipulation of the `pool.liquidity` function could still impact the health factor of a borrower's position.

Impact The health factor of a borrower's position could be manipulated.

Suggestion N/A

Feedback from the Project Both `pool.liquidity` and the pool's `feeGrowthGlobals` can be manipulated. This is why we have a rate limit on how quickly the reported `IV` can change `(uint256 maxChange = timeSinceLastWrite * IV_CHANGE_PER_SECOND)`. We've decided not to change the `IV_CHANGE_PER_SECOND` constant. When we add governance setters we may allow it to be updated, but not right now.

2.3 Additional Recommendation

2.3.1 Avoid precision loss

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `computeLiquidationIncentive` function of the `BalanceSheet` contract, the division is performed before the multiplication (Line 110), which may lead to precision loss.

```
105    if (liabilities0 > assets0) {
106        // shortfall is the amount that cannot be directly repaid using Borrower assets at this
               price
107        uint256 shortfall = liabilities0 - assets0;
108        // to cover it, a liquidator may have to use their own assets, taking on inventory risk.
109        // to compensate them for this risk, they're allowed to seize some of the surplus asset.
110        incentive1 += mulDiv96(shortfall / LIQUIDATION_INCENTIVE, meanPriceX96);
111    }
```

Listing 2.8: BalanceSheet.sol

Impact Precision loss may happen.

Suggestion Perform multiplication before division.

2.4 Note

2.4.1 Potential cost-free minting due to uncredited balances

Description The `deposit` function in the `Lender` contract has a potential vulnerability that could allow a malicious user to mint shares without providing tokens.

During the deposit process, users may first pre-pay tokens and then call the `deposit` function to mint shares based on the input amount. However, this function does not enforce a match between the actual transfer amount and the input. If the actual transfer exceeds the input amount, the excess remains in the contract. A malicious user could subsequently call deposit to mint shares using these residual tokens without providing any funds.

Furthermore, it's important to note that malicious users can continuously monitor the pool's status. This is possible because the transfer and deposit actions are not encompassed within a single atomic transaction. This vulnerability paves the way for for malicious users to engage in front-running. Particularly, they can preemptively perform a deposit action as soon as they detect a token transfer to the contract.

```
ERC20 asset_ = asset();
bool didPrepay = cache.lastBalance <= asset_.balanceOf(address(this));
if (!didPrepay) {
```

```
        asset_.safeTransferFrom(msg.sender, address(this), amount);
    }

    emit Deposit(msg.sender, beneficiary, amount, shares);
```

Listing 2.9: Lender.sol

Feedback from the Project This is intended behavior for our push-architecture. Users are expected to use our periphery contracts to ensure deposits are done atomically, or they can use the pull-based fallback option `(if (!didPrepay) asset.safeTransferFrom(...))`.

2.4.2 The protocol will not support deflation/inflation tokens

Description Aloell core adopts an internal-accounting design, which requires a precise match between the deposit amount and the actual transferred amount. Any discrepancy can disrupt the internal accounting. For example, in the `deposit` function of `Lender.sol`, if `asset_` is a deflation or inflation token, the actual token balance change will not align exactly with the amount used in this function. To prevent related side effects, the protocol should not support such tokens.

```
    cache.lastBalance += amount;

    // Save state to storage (thus far, only mappings have been updated, so we must address
        everything else)
    _save(cache, /* didChangeBorrowBase: */ false);

    // Ensure tokens are transferred
    ERC20 asset_ = asset();
    bool didPrepay = cache.lastBalance <= asset_.balanceOf(address(this));
    if (!didPrepay) {
        asset_.safeTransferFrom(msg.sender, address(this), amount);
    }

    emit Deposit(msg.sender, beneficiary, amount, shares);
```

Listing 2.10: Lender.sol

2.4.3 The protocol will not support double-entry tokens

Description In the `Borrower` contract, there's a function named `rescue` that allows the owner to withdraw tokens that have been transferred by mistake. This function stipulates that the tokens to be rescued should not be the two tokens that could potentially be borrowed assets.

```
    function rescue(ERC20 token) external {
        require(token != TOKEN0 && token != TOKEN1);
        token.safeTransfer(slot0.owner, token.balanceOf(address(this)));
    }
```

Listing 2.11: Borrower.sol

However, there are double-entry tokens that the protocol should not support, such as certain `Synthetix` tokens (e.g., sBTC). If the protocol were to support such tokens, the owner of the `Borrower` contract could easily withdraw all tokens from the secondary entry by invoking the `rescue` function, an action that should be disallowed.

Feedback from the Project AloeII core will not support double-entry tokens.

2.4.4 All pools are `UniswapV3` pools

Description AloeII core uses `UniswapV3` pools for its liquidity mining process. During the audit, we made the assumption that all pools are `UniswapV3` pools, not those from forked protocols. This is because any potential effects stemming from differences between the forked protocol and the original `UniswapV3` could lead to unexpected behaviors.