Security Audit Report for ApeX Protocol Smart Contract

Date: Jan 23, 2022

Version: 1.2

Contact: contact@blocksecteam.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2022.01.24
21:50:16 +08'00'

Contents

Report Manifest

Item	Description
Client	ApeX
Target	ApeX Protocol Smart Contract

Version History

Version	Date	Description
1.0	Jan 9, 2022	First Release
1.1	Jan 14, 2022	Second Release
1.2	Jan 23, 2022	Third Release

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 auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [1] during the audit are shown in the following.

Repo Name	Stage	Commit SHA
ApeX Protocol	Initial	19996f81f67cd5c3fa22add6f71c7de309f253f8
ApeX Protocol	Update	85fe3eb33bb68fcb172dd1e28a667745a17dbfca
ApeX Protocol	Final	58aa173c625dc27b34ee91a47b91831e604258c3

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.

[1] https://github.com/ApeX-Protocol/apex-protocol

- **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 [2] and Common Weakness Enumeration [3]. 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.

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

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

Chapter 2 Findings

In total, we find `ten` potential issues in the smart contract, and we also have `four` recommendations, as follows:

- High Risk: 1
- Medium Risk: 6
- Low Risk: 3
- Recommendations: 4

ID	Severity	Description	Category	Status
1	Medium	Incorrect Ownership for BondPool	Software Security	Fixed
2	Low	Potential Reentrancy Issue in StakingPool	Software Security	Fixed
3	High	Price Manipulation in Margin Ratio Calculation	DeFi Security	Fixed
4	Medium	Ineffective Price Slippage Check in Amm	DeFi Security	Fixed
5	Medium	Possible DoS in Liquidation (Version 1)	DeFi Security	Fixed
6	Low	Possible DoS in Liquidation (Version 2)	DeFi Security	Acknowledged
7	Low	Incorrect Calculation for Payouts	DeFi Security	Fixed
8	Medium	Potential Unsafe Price Oracle	DeFi Security	Fixed
9	Medium	Incorrect Accounting in StakingPool	DeFi Security	Fixed
10	Medium	Incorrect Weight Accounting in StakingPool	DeFi Security	Fixed
11	-	Potential Unchecked Integer Overflow	Additional Recommendations	Fixed
12	-	Partial Liquidation	Additional Recommendations	Acknowledged
13	-	Inconsistent Calculation Basis for `_mintFee`	Additional Recommendations	Fixed
14	-	No Access Control in the Invitation Contract	Additional Recommendations	Acknowledged

The details are provided in the following sections.

2.1 Software Security

2.1.1 Incorrect Ownership for BondPool

Status Fixed

Description The owner of the BondPool is set to `msg.sender` in the `constructor`. However, the BondPool can only be deployed by the BondPoolFactory (otherwise it is not registered), which has no logic to call `BondPool.setPendingOwner()`. As a result, all `onlyOwner` functions in the BondPool cannot be invoked.

```
23    constructor(
24        address apeXToken_,
25        address treasury_,
26        address priceOracle_,
27        address amm_,
28        uint256 maxPayout_,
29        uint256 discount_,
30        uint256 vestingTerm_
31    ) {
32        owner = msg.sender;
```

Listing 2.1: BondPool.sol

```
46    function createPool(address amm) external override onlyOwner returns (address) {
47        address pool = address(new BondPool(apeXToken, treasury, priceOracle, amm, maxPayout,
              discount, vestingTerm));
48        allPools.push(pool);
49        emit BondPoolCreated(amm, pool);
50        return pool;
51    }
```

Listing 2.2: BondPoolFactory.sol

Impact All `onlyOwner` functions (mainly for governance purpose) cannot be invoked.

Suggestion Revise the ownership assignment for BondPool.

2.1.2 Potential Reentrancy Issue in StakingPool

Status Fixed

Description The function `deposit()` of StakingPool is vulnerable to reentrancy attacks if certain types of tokens are used as the `poolToken`. Specifically: 1) a user can call `stake()`; and 2) if the `poolToken` supports callbacks to the sender, in line 53 of the following code, the `transferFrom()` function will call the `msg.sender`; and 3) in the callback, the user can invoke the `_processRewards()` function, which will double-count the user's rewards.

```
39    function stake(uint256 _amount, uint256 _lockUntil) external override nonReentrant {
40        require(_amount > 0, "cp._stake: INVALID_AMOUNT");
41        uint256 now256 = block.timestamp;
42        // ...
43        uint256 yieldLockTime = factory.yieldLockTime();
44        require(
45            _lockUntil == 0 || (_lockUntil > now256 && _lockUntil <= now256 + yieldLockTime),
46            "cp._stake: INVALID_LOCK_INTERVAL"
47        );
48
49        address _staker = msg.sender;
50        User storage user = users[_staker];
51        _processRewards(_staker, user);
```

```
52
53        IERC20(poolToken).transferFrom(_staker, address(this), _amount);
```

Listing 2.3: StakingPool.sol

```
229    function _processRewards(address _staker, User storage user) internal {
230        syncWeightPrice();
231
232        // ...
233        if (user.tokenAmount == 0) return;
234        uint256 yieldAmount = (user.totalWeight * yieldRewardsPerWeight) /
235            REWARD_PER_WEIGHT_MULTIPLIER -
236            user.subYieldRewards;
237        if (yieldAmount == 0) return;
238
239        // ...
240        if (poolToken == apex) {
241            uint256 yieldWeight = yieldAmount * MAX_TIME_STAKE_WEIGHT_MULTIPLIER;
242            uint256 now256 = block.timestamp;
243            uint256 lockUntil = now256 + factory.yieldLockTime();
244            uint256 depositId = user.deposits.length;
245            Deposit memory newDeposit = Deposit({
246                amount: yieldAmount,
247                weight: yieldWeight,
248                lockFrom: now256,
249                lockUntil: lockUntil,
250                isYield: true
251            });
252            user.deposits.push(newDeposit);
253            user.tokenAmount += yieldAmount;
254            user.totalWeight += yieldWeight;
255            usersLockingWeight += yieldWeight;
256            emit YieldClaimed(_staker, depositId, yieldAmount, now256, lockUntil);
257        } else {
258            address apexStakingPool = factory.getPoolAddress(apex);
259            IStakingPool(apexStakingPool).stakeAsPool(_staker, yieldAmount);
260        }
261    }
```

Listing 2.4: StakingPool.sol

Impact Users can get extra rewards if the `poolToken` supports callbacks to the sender.

Suggestion Add `nonReentrant` to protect the `_processRewards()` function.

2.2 DeFi Security

2.2.1 Price Manipulation in Margin Ratio Calculation

Status Fixed

Description The margin ratio calculation in Margin is based on `Amm.estimateSwap()`, which depends on current reserves of Amm. However, the current reserves can be manipulated by borrowing the flash loan

and opening a long position. In other words, a flash loan attack can be launched against Margin to make it miscalculate margin ratio for any account, leading to bad debts (and other severe consequences).

```
610    function _calMarginRatio(int256 quoteSize, int256 baseSize) internal view returns (uint256
           marginRatio) {
611        if (quoteSize == 0 || (quoteSize > 0 && baseSize >= 0)) {
612            marginRatio = 10000;
613        } else if (quoteSize < 0 && baseSize <= 0) {
614            marginRatio = 0;
615        } else if (quoteSize > 0) {
616            // ...
617            uint256[2] memory result = IAmm(amm).estimateSwap(
618                address(quoteToken),
619                address(baseToken),
620                quoteSize.abs(),
621                0
622            );
623            // ...
624            uint256 baseAmount = result[1];
625            // ...
626            marginRatio = (baseSize.abs() >= baseAmount || baseAmount == 0)
627                ? 0
628                : baseSize.mulU(10000).divU(baseAmount).addU(10000).abs();
629        } else {
630            // ...
631            uint256[2] memory result = IAmm(amm).estimateSwap(
632                address(baseToken),
633                address(quoteToken),
634                0,
635                quoteSize.abs()
636            );
637            // ...
638            uint256 baseAmount = result[0];
639            // ...
640            marginRatio = baseSize.abs() < baseAmount ? 0 : 10000 - ((baseAmount * 10000) /
                baseSize.abs());
641        }
642    }
```

Listing 2.5: Margin.sol

Impact The unsafe price feeds may cause severe bad debts (and other losses).

Suggestion Use safe price feeds for the price calculation.

Feedback from the Developers The bussiness logic for the `_calMarginRatio` is completely redesigned and this issue is fixed by now. At first, we use `estimateSwap` based on current spot price to calculate margin ratio, then compare margin ratio and `initMarginRatio`. But to be consistent with risk formula, we changed the implementation to the formula, which is based on risk parameters: `beta` and `initMarginRatio`.

2.2.2 Ineffective Price Slippage Check in Amm

Status Fixed

Description In the `Updated` version of Amm, the slippage check is only implemented when `timeElapsed > 0`, which cannot be used to prevent price slippage in the same block.

```
324    function _update(
325        uint256 baseReserveNew,
326        uint256 quoteReserveNew,
327        uint112 baseReserveOld,
328        uint112 quoteReserveOld
329    ) private {
330        require(baseReserveNew <= type(uint112).max && quoteReserveNew <= type(uint112).max, "AMM.
               _update: OVERFLOW");
331        uint32 blockTimestamp = uint32(block.timestamp % 2**32);
332        uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
333
334        if (timeElapsed > 0 && baseReserveOld != 0 && quoteReserveOld != 0) {
335            // ...
336            lastPrice = uint256(UQ112x112.encode(quoteReserveOld).uqdiv(baseReserveOld));
337            price0CumulativeLast += uint256(UQ112x112.encode(quoteReserveOld).uqdiv(baseReserveOld)
                   ) * timeElapsed;
338            price1CumulativeLast += uint256(UQ112x112.encode(baseReserveOld).uqdiv(quoteReserveOld)
                   ) * timeElapsed;
339
340            uint256 numerator = quoteReserveNew * baseReserveOld * 100;
341            uint256 demominator = baseReserveNew * quoteReserveOld;
342            uint256 tradingSlippage = IConfig(config).tradingSlippage();
343            require(
344                (numerator < (100 + tradingSlippage)* demominator ) && (numerator > (100 -
                       tradingSlippage)* demominator),
345                "AMM._update: TRADINGSLIPPAGE_TOO_LARGE"
346            );
347        }
```

Listing 2.6: Amm.sol

Impact Price manipulation attack in the same block cannot be prevented.

Suggestion Remove the improper check for `timeElapsed`.

2.2.3 Possible DoS in Liquidation (Version 1)

Status Fixed

Description This issue exists in the `Updated` version of the smart contract. Specifically, there is a price slippage protection in function `_update()`. This function will be invoked by `Amm.forceSwap()`, which is used for liquidation. However, the slippage is inevitable for a large liquidation. As such, it would cause DoS in liquidation which may result in bad debts.

```
160    function forceSwap(
161        address inputToken,
162        address outputToken,
163        uint256 inputAmount,
164        uint256 outputAmount
165    ) external override nonReentrant onlyMargin {
```

```
166         require(inputToken == baseToken || inputToken == quoteToken, "Amm.forceSwap:
                WRONG_INPUT_TOKEN");
167         require(outputToken == baseToken || outputToken == quoteToken, "Amm.forceSwap:
                WRONG_OUTPUT_TOKEN");
168         require(inputToken != outputToken, "Amm.forceSwap: SAME_TOKENS");
169         (uint112 _baseReserve, uint112 _quoteReserve, ) = getReserves();
170         uint256 reserve0;
171         uint256 reserve1;
172         if (inputToken == baseToken) {
173             reserve0 = _baseReserve + inputAmount;
174             reserve1 = _quoteReserve - outputAmount;
175         } else {
176             reserve0 = _baseReserve - outputAmount;
177             reserve1 = _quoteReserve + inputAmount;
178         }
179         _update(reserve0, reserve1, _baseReserve, _quoteReserve);
180         emit ForceSwap(inputToken, outputToken, inputAmount, outputAmount);
181     }
```

Listing 2.7: Amm.sol

Impact The liquidation may be ineffective during large price slippages and may cause bad debts.

Suggestion Remove the improper checks in `Amm.forceSwap()`.

2.2.4 Possible DoS in Liquidation (Version 2)

Status Acknowledged

Description In the `Updated` version, there is a limitation on the actions that can be done in each block. Specifically, any address that have previously called `openPosition` and `closePosition` cannot execute position-related operations (i.e. `openPosition`, `closePosition` and `liquidate`) in that block. However, this may not be suitable for `liquidate()`, since it may cause DoS in liquidation, especially when there is an excessive price volatility and the liquidators are not enough.

```
298     function liquidate(address trader)
299         external
300         override
301         nonReentrant
302         returns (
303             uint256 quoteAmount,
304             uint256 baseAmount,
305             uint256 bonus
306         )
307     {
308         require(traderLatestOperation[msg.sender] != block.number, "Margin.liquidate:
                ONE_BLOCK_TWICE_OPERATION");
```

Listing 2.8: Margin.sol

Impact N/A

Suggestion Remove the operation limitation in `liquidate()`.

Feedback from the Developers If the liquidator only does liquidation, then it will always meet this requirement and continue to do multiple liquidation in the same transaction (or the same block). If the liquidator does liquidation and also open or close positions, then he/she will have the chance to be denied of service.

2.2.5 Incorrect Calculation for Payouts

Status Fixed

Description The implementation of function `payoutFor()` in BondPool to calculate the payout is incorrect.

```
174    function payoutFor(uint256 amount) public view override returns (uint256 payout) {
175        address baseToken = IAmm(amm).baseToken();
176        uint256 marketApeXAmount = IPriceOracle(priceOracle).quote(baseToken, apeXToken, amount);
177        uint256 denominator = (marketApeXAmount * (10000 - discount)) / 10000;
178        payout = FullMath.mulDiv(amount, amount, denominator);
179    }
```

Listing 2.9: BondPool.sol

Impact The calculated payout is not correct.

Suggestion Fix the payout calculation.

2.2.6 Potential Unsafe Price Oracle

Status Fixed

Description The UniswapV2 price used in Amm is *spot* price, which is the division of reserves in the Uniswap pair. This is vulnerable to the price manipulation.

Besides, the entire price oracle design for `PriceOracle.getIndexPrice()` solely depends on Uniswap V2 and V3 pairs. It means that for each pair of the `baseToken` and the `quoteToken`, a Uniswap V3 pair with sufficient liquidity and correct price must be present. What's more, the `PairFactory.createPair()` function is accessible by any account, which suggests that anyone can deploy a Amm-Margin pair with a potentially unreliable price oracle.

```
173    function quoteFromV2(
174        address baseToken,
175        address quoteToken,
176        uint256 baseAmount
177    ) public view returns (uint256 quoteAmount) {
178        address pair = IUniswapV2Factory(v2Factory).getPair(baseToken, quoteToken);
179        if (pair == address(0)) return 0;
180        (uint112 reserve0, uint112 reserve1, ) = IUniswapV2Pair(pair).getReserves();
181        if (baseToken == IUniswapV2Pair(pair).token0()) {
182            quoteAmount = FullMath.mulDiv(baseAmount, reserve1, reserve0);
183        } else {
184            quoteAmount = FullMath.mulDiv(baseAmount, reserve0, reserve1);
185        }
186    }
```

Listing 2.10: PriceOracle.sol

```
107    function getIndexPrice(address amm) public view override returns (uint256) {
108        address baseToken = IAmm(amm).baseToken();
109        address quoteToken = IAmm(amm).quoteToken();
110        uint256 baseDecimals = IERC20(baseToken).decimals();
111        uint256 quoteDecimals = IERC20(quoteToken).decimals();
112        uint256 quoteAmount = quote(baseToken, quoteToken, 10**baseDecimals);
113        return quoteAmount * (10**(18 - quoteDecimals));
114    }
```

Listing 2.11: PriceOracle.sol

Impact The price feeds of PriceOracle may be manipulated.

Suggestion Use more reliable price feeds.

Feedback from the Developers We have provided a new solution to protect our project from the price manipulation risk:

- First, the original `PriceOracle` contract is only used for the core protocol. The new `PriceOracle` fetches the index price from UniswapV3 TWAP, and saves the price inside Amm. This price will be used to compare with the index price for the `rebase()` function.
- Second, a `BondPriceOracle` contract is added to save and get APEX TWAP price. Specifically, `BondPriceOracle` will maintain the APEX-WETH pair TWAP from SushiSwap and UniswapV2, fetch the BaseToken-WETH TWAP from UniswapV3, and use WETH as a bridge token to get BaseToken-APEX TWAP.

2.2.7 Incorrect Accounting in StakingPool

Status Fixed

Description In the function `StakingPool.unstakeBatch()`, if `stakeDeposit.amount == _amount` (i.e., completely withdraw) `user.deposits[_depositId]` will be deleted in line 113. However, it is assigned to `stakeDeposit` again in line 119.

```
76    function unstakeBatch(uint256[] memory _depositIds, uint256[] memory _amounts) external
          override {
77        require(_depositIds.length == _amounts.length, "cp.unstakeBatch: INVALID_DEPOSITS_AMOUNTS")
              ;
78        address _staker = msg.sender;
79        uint256 now256 = block.timestamp;
80        User storage user = users[_staker];
81        _processRewards(_staker, user);
82        uint256 yieldLockTime = factory.yieldLockTime();
83
84        uint256 yieldAmount;
85        uint256 stakeAmount;
86        uint256 _amount;
87        uint256 _depositId;
88        uint256 previousWeight;
89        uint256 newWeight;
90        uint256 deltaUsersLockingWeight;
91        Deposit memory stakeDeposit;
92        for (uint256 i = 0; i < _depositIds.length; i++) {
```

```
93            _amount = _amounts[i];
94            _depositId = _depositIds[i];
95            require(_amount != 0, "cp.unstakeBatch: INVALID_AMOUNT");
96            stakeDeposit = user.deposits[_depositId];
97            require(stakeDeposit.lockFrom == 0 || now256 > stakeDeposit.lockUntil, "cp.unstakeBatch
                  : DEPOSIT_LOCKED");
98            require(stakeDeposit.amount >= _amount, "cp.unstakeBatch: EXCEED_STAKED");
99
100           previousWeight = stakeDeposit.weight;
101           // ...
102           newWeight =
103               (((stakeDeposit.lockUntil - stakeDeposit.lockFrom) * WEIGHT_MULTIPLIER) /
104                   yieldLockTime +
105                   WEIGHT_MULTIPLIER) *
106               (stakeDeposit.amount - _amount);
107           if (stakeDeposit.isYield) {
108               yieldAmount += _amount;
109           } else {
110               stakeAmount += _amount;
111           }
112           if (stakeDeposit.amount == _amount) {
113               delete user.deposits[_depositId];
114           } else {
115               stakeDeposit.amount -= _amount;
116               stakeDeposit.weight = newWeight;
117           }
118
119           user.deposits[_depositId] = stakeDeposit;
120           user.tokenAmount -= _amount;
121           user.totalWeight -= (previousWeight - newWeight);
122           deltaUsersLockingWeight += (previousWeight - newWeight);
123       }
124       user.subYieldRewards = (user.totalWeight * yieldRewardsPerWeight) /
              REWARD_PER_WEIGHT_MULTIPLIER;
125       usersLockingWeight -= deltaUsersLockingWeight;
126
127       if (yieldAmount > 0) {
128           factory.transferYieldTo(_staker, yieldAmount);
129       }
130       if (stakeAmount > 0) {
131           IERC20(poolToken).transfer(_staker, stakeAmount);
132       }
133       emit UnstakeBatch(_staker, _depositIds, _amounts);
134   }
```

Listing 2.12: StakingPool.sol

Impact As a result, any user is able to remove the staked tokens regardless of the locking period (if there exist multiple stakes with different locking periods).

Suggestion N/A

2.2.8 Incorrect Weight Accounting in StakingPool

Status Fixed

Description In the function `updateStakeLock()` of StakingPool, the `totalWeight` is updated without modifying `subYieldRewards`. As a result, users might be able to claim more rewards.

```
169    function updateStakeLock(uint256 _depositId, uint256 _lockUntil) external override {
170        uint256 now256 = block.timestamp;
171        require(_lockUntil > now256, "cp.updateStakeLock: INVALID_LOCK_UNTIL");
172
173        uint256 yieldLockTime = factory.yieldLockTime();
174        address _staker = msg.sender;
175        User storage user = users[_staker];
176        Deposit storage stakeDeposit = user.deposits[_depositId];
177        require(_lockUntil > stakeDeposit.lockUntil, "cp.updateStakeLock: INVALID_NEW_LOCK");
178
179        if (stakeDeposit.lockFrom == 0) {
180            require(_lockUntil <= now256 + yieldLockTime, "cp.updateStakeLock:
                EXCEED_MAX_LOCK_PERIOD");
181            stakeDeposit.lockFrom = now256;
182        } else {
183            require(_lockUntil <= stakeDeposit.lockFrom + yieldLockTime, "cp.updateStakeLock:
                EXCEED_MAX_LOCK");
184        }
185
186        stakeDeposit.lockUntil = _lockUntil;
187        uint256 newWeight = (((stakeDeposit.lockUntil - stakeDeposit.lockFrom) * WEIGHT_MULTIPLIER)
              /
188            yieldLockTime +
189            WEIGHT_MULTIPLIER) * stakeDeposit.amount;
190        uint256 previousWeight = stakeDeposit.weight;
191        stakeDeposit.weight = newWeight;
192        user.totalWeight = user.totalWeight - previousWeight + newWeight;
193        usersLockingWeight = usersLockingWeight - previousWeight + newWeight;
194
195        emit UpdateStakeLock(_staker, _depositId, stakeDeposit.lockFrom, _lockUntil);
196    }
```

Listing 2.13: StakingPool.sol

Impact Users can claim more rewards.

Suggestion Update `subYieldRewards` after changing `totalWeight`.

2.3 Additional Recommendation

2.3.1 Potential Unchecked Integer Overflow

Status Fixed

Description In `Amm.mint()`, there is an unchecked integer conversion that may cause integer overflow.

```
143    function mint(address to)
144        external
```

```
145         override
146         nonReentrant
147         returns (
148             uint256 baseAmount,
149             uint256 quoteAmount,
150             uint256 liquidity
151         )
152     {
153         (uint112 _baseReserve, uint112 _quoteReserve, ) = getReserves(); // gas savings
154 
155         // ...
156         int256 baseTokenOfNetPosition = IMargin(margin).netPosition();
157         int256 realBaseReserveSigned = int256(uint256(_baseReserve)) + baseTokenOfNetPosition;
158         uint256 realBaseReserve = uint256(realBaseReserveSigned);
159 
160         baseAmount = IERC20(baseToken).balanceOf(address(this));
161         require(baseAmount > 0, "Amm.mint: ZERO_BASE_AMOUNT");
```

Listing 2.14: Amm.sol

Impact Integer overflows in `mint()` may cause logical errors.

Suggestion Add extra sanity checks.

2.3.2 Partial Liquidation

Status Acknowledged

Description Currently, the liquidation logic is to completely remove the position being liquidated. Under certain circumstances, there is no need to liquidate all users' debts. It is recommended that the liquidation should not continue if it reaches a healthy position, or impose a threshold of liquidation ratio. This may prevent users from second loss.

Impact N/A

Suggestion Implement partial liquidation logic.

Feedback from the Developers It is intended to liquidate the entire position to keep the liquidation in a simple way and protect the liquidity providers from loosing too much.

2.3.3 Inconsistent Calculation Basis for `_mintFee`

Status Fixed

Description The function `Amm.rebase()` is designed to update the reserves of the Amm, but the constant `K` is not updated accordingly. If an invocation to `Amm.rebase()` causes `K` to increase, the next invocation to `mint()` or other related functions (that invoke the `_mintFee()` function) will lead to an incorrect fee with a large quantity.

```
62     function rebase() external override nonReentrant returns (uint256 quoteReserveAfter) {
63         require(msg.sender == tx.origin, "Amm.rebase: ONLY_EOA");
64         (uint112 _baseReserve, uint112 _quoteReserve, ) = getReserves();
65         // ...
66         quoteReserveAfter = IPriceOracle(IConfig(config).priceOracle()).quote(baseToken, quoteToken
               , _baseReserve);
```

```
67        uint256 gap = IConfig(config).rebasePriceGap();
68        require(
69            quoteReserveAfter * 100 >= uint256(_quoteReserve) * (100 + gap) ||
70                quoteReserveAfter * 100 <= uint256(_quoteReserve) * (100 - gap),
71            "Amm.rebase: NOT_BEYOND_PRICE_GAP"
72        );
73        _update(_baseReserve, quoteReserveAfter, _baseReserve, _quoteReserve);
74        emit Rebase(_quoteReserve, quoteReserveAfter);
75    }
```

Listing 2.15: Amm.sol

Impact N/A

Suggestion Invoke the `_mintFee()` function and update `K` in the `rebase()` function.

2.3.4 No Access Control in the Invitation Contract

Status Acknowledged

Description The `register()` function and the `acceptInvitation()` function in the Invitation contract have no access control. It means that anyone (any address) can be registered as a "lower" of any registered account.

Impact N/A

Suggestion Add access control logic in the Invitation contract.

Feedback from the Developers The Invitation contract only provides an on-chain relationship of the registered uppers and lowers. The actual amount of rewards distributed is determined based on the volume of trading, the count of transactions, etc. Therefore, it would be non-profitable to just register in the Invitation contract.