STRONGHOLD
SECURITY

Clearpool Credit Vaults Security Audit Report

May 02, 2024

Contents

Executive Summary

Title	Description
Client	Clearpool Finance
Project	credit-vaults
Platform	Ethereum
Language	Solidity
Repository	https://github.com/clearpool-finance/credit-vaults-public, https://github.com/clearpool-finance/credit-vaults
Initial commit	d9fd58b54eed66d110519c702b2ddfe45a51a408
Final commit	021974ef85b4e2ed98c76c7d76042f09b7b65b8d
Timeline	March 28 2024 - April 29 2024

Project Overview

Credit Vaults a lending product designed to benefit both lenders and borrowers.

Audit Scope

File	Link
Decimal.sol	Decimal.sol
NFTDescriptor.sol	NFTDescriptor.sol
RewardAsset.sol	RewardAsset.sol
Utils.sol	Utils.sol
Auction.sol	Auction.sol
BondNFT.sol	BondNFT.sol
PoolFactory.sol	PoolFactory.sol
PoolMaster.sol	PoolMaster.sol
WhitelistControl.sol	WhitelistControl.sol

Audit Methodology

General Code Assessment

The code is reviewed for clarity, consistency, style, and whether it follows code best practices applicable to the particular programming language used, such as indentation, naming convention, commented code blocks, code duplication, confusing names, irrelevant
or missing comments, etc. This part is aimed at understanding the overall code structure and protocol architecture. Also, it seeks to learn overall system architecture and business logic and how different parts of the code are related to each other.

Code Logic Analysis

The code logic of particular functions is analyzed for correctness and efficiency. The code is checked for what it is intended for, the algorithms are optimal and valid, and the correct data types are used. The external libraries are checked for relevance and correspond to the tasks they solve in the code. This part is needed to understand the data structures used and the purposes for which they are used. At this stage, various public checklists are applied in order to ensure that logical flaws are detected.

Entities and Dependencies Usage Analysis

The usages of various entities defined in the code are analyzed. This includes both: internal usage from other parts of the code as well as possible dependencies and integration usage. This part aims to understand and spot overall system architecture flaws and bugs in integrations with other protocols.

Access Control Analysis

Access control measures are analyzed for those entities that can be accessed from outside. This part focuses on understanding user roles and permissions, as well as which assets should be protected and how.

Use of checklists and auditor tools

Auditors can perform a more thorough check by using multiple public checklists to look at the code from different angles. Static analysis tools (Slither) help identify simple errors and highlight potentially hazardous areas. While using Echidna for fuzz testing will speed up the testing of many invariants, if necessary.

Vulnerabilities

The audit is directed at identifying possible vulnerabilities in the project's code. The result of the audit is a report with a list of detected vulnerabilities ranked by severity level:

Severity	Description
● Critical	Vulnerabilities leading to the theft of assets, blocking access to funds, or any other loss of funds.
● High	Vulnerabilities that cause the contract to fail and that can only be fixed b y modifying or completely replacing the contract code.
● Medium	Vulnerabilities breaking the intended contract logic but without loss of fun ds and need for contract replacement.
● Low	Minor bugs that can be taken into account in order to improve the overall qu ality of the code

After the stage of bug fixing by the Customer, the findings can be assigned t he following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect it s security.
Acknowledged	The Customer took into account the finding. However, the recommendations wer e not implemented since they did not affect the project's safety.

Findings Summary

Severity	# of Findings
Critical	0
High	5
Medium	11
Low	11

ID	Severity	Title	Status
H-1	High	The reward can accrue wrong for an empty pool	Fixed
H-2	High	The borrower can block the withdrawal of funds	Fixed
H-3	High	Lenders can infinitely withdraw rewards with a `Closed` status of the pool.	Fixed
H-4	High	The borrower can set `lendAPR` to a minimum, use lenders' funds and pay almost nothing.	Fixed
H-5	High	Logical issues in the accrual of interests process	Fixed
M-1	Medium	An unlimited protocol fee and penalty rate	Acknowledged
M-2	Medium	There are no limits for variables	Acknowledged
M-3	Medium	A missing check that the bidder is in the whitelist in the `increaseBid` function	Fixed
M-4	Medium	No checking of input variables	Acknowledged
M-5	Medium	No price slippage protection	Acknowledged
M-6	Medium	The `supply()` function can be called even when the `_fullRepaymentDate` value is set to a non-zero value.	Fixed
M-7	Medium	The penalty accrues only when `block.timestamp` > `currentRepaymentDate`	Fixed
M-8	Medium	The `feeAmount` remains `unpaid` after the auction is resolved.	Acknowledged
M-9	Medium	The auction can be blocked in case `lastBidder` occurs in the `USDC` blacklist.	Acknowledged

M-10	● Medium	The limits for `_newAPR`	Acknowledged
M-11	● Medium	A possible revert of the `supply` function if `depositCap` is 0.	Fixed
L-1	● Low	The `vaults` variable may not be updated	Fixed
L-2	● Low	Division by zero	Acknowledged
L-3	● Low	A two-step ownership transfer	Acknowledged
L-4	● Low	Reentrancy risk	Acknowledged
L-5	● Low	Unused error types	Fixed
L-6	● Low	Typos	Fixed
L-7	● Low	The NatSpec updates	Fixed
L-8	● Low	An invalid error type	Fixed
L-9	● Low	An additional check	Fixed
L-10	● Low	A memory gap for the upgradable contract is missing	Fixed
L-11	● Low	The `nonReentrant` modifier should occur before other modifiers.	Fixed

Findings

Critical

Not Found

High

H-1	● High	The reward can accrue wrong for an empty pool	Fixed

Description

PoolMaster.sol#L945-L965

The `_accrueReward` function can accrue rewards incorrectly if, after initialization, all users withdraw funds and then someone puts any amount of funds to the protocol.

This occurred because of `totalSupply() != 0`.

Let's consider the following flow:

1. Alice calls the `supply(100)` function and she mints 100 shares, `totalSupply = 100`; the `_accrueReward` function isn't called (because of the first call);
2. Alice calls the `supply(1)` function, she mints 1 share, `totalSupply = 100`, the `_accrueReward` function is called with `totalSupply = 100`;
3. From step 2: the `rewardAssetInfo.updateMagnifiedRewardPerShare` function is called and specifies how many rewards per asset should be distributed;
4. Next, Alice withdraws all shares (101), and the `rewardAssetInfo.updateMagnifiedRewardPerShare` function is updated once again (in the `_burn` function);
5. Now, the pool is empty. We calculate rewards per share from step 1 to step 4 and store them;
6. Wait for two days (for example);
7. Alice calls the `supply(10)` function, but no updates of the `rewardAssetInfo.updateMagnifiedRewardPerShare` function take place;
8. Alice calls the `supply(1)` function again, and the `rewardAssetInfo.updateMagnifiedRewardPerShare` function is updated, but the `rewardAssetInfo.lastRewardDistribution=timeFromStep4`, and now time value is greater;
9. At step 8, we accrue rewards for `10` shares for two days (from step 6);

10. But according to the logic and idea, we must update shares only for time from step 7 till step 8;

Recommendation

We recommend implementing a `dead shares` approach, with the first supply sending 1000 shares to a null address.

Client's commentary

Fixed in f61b0b36.

H-2 ● High **The borrower can block the withdrawal of funds** Fixed

Description

PoolMaster.sol#L439-L440
PoolMaster.sol#L1222-L1233
PoolMaster.sol#L1006-L1012

The borrower can block the withdrawal of funds.

The `_getNextUnpaidRound` function uses the `minimumNoticePeriod` value, which the borrower can decrease with the `changeNoticePeriod` function.

If the borrower significantly decreases the `minimumNoticePeriod` value and then calls `repayAll(),` `PoolMaster` doesn't find the next unpaid round and returns empty `RepaymentRoundInfo.`

Let's consider the following case:

1. Alice calls the `supply` function, and the `currentRepaymentDate` value is filled with value X;
2. Alice calls the `requestWithdrawal` function, and the `currentRepaymentDate` value is changed to the `X + repaymentFrequency` value;
3. The time now is `X + repaymentFrequency + 1`;
4. Alice calls the `requestWithdrawal` function, and the `currentRepaymentDate` value is changed to `X + repaymentFrequency + minimumNoticePeriod`;
5. The borrower calls the `changeNoticePeriod` function and decreases the `minimumNoticePeriod` by two times;
6. The borrower calls the `repayAll` function, and the `currentRepaymentDate` value will be handled correctly;
7. The following code begins to handle `_getNextUnpaidRound(true)`;
8. But, since the `getNextUnpaidRound` function uses condition `(block.timestamp + minimumNoticePeriod > nextRepDate)`, where the `minimumNoticePeriod` is decreased by two times (step 5), we don't reach correctly the `nextRepDate` value (where `repaymentRoundInfo[nextRepDate].debtAmount > 0`);
9. `repayAll` happened, but the `repaymentRoundInfo[nextRepDate].paidAt` line won't be executed, and therefore `redeemBond` will be failed for the `next` bond due to `if (roundInfo.paidAt == 0 || roundInfo.paidAt > block.timestamp) revert RepaymentMissing();`;

Thus, the user's funds will be locked.

Recommendation

We recommend forbidding changes to `minimumNoticePeriod` without the owner's approval or implementing limits for `minimumNoticePeriod`.

Client's commentary

Fixed in 6fc6602f.

H-3	● High	Lenders can infinitely withdraw rewards with a `Closed` status of the pool.	Fixed

Description

In the `PoolMaster.sol` contract, the `_accrueReward()`, the `_accumulativeRewardOf()`, and the `_accrueRoundReward()` functions are used to update the pool's reward state.
However, rewards still accrue even after the pool is closed and it has a `Closed` status.

Therefore, the following situation is possible:

When the borrower uses the `repayAll()` function and closes the pool, the lender can call the `withdrawReward()` function on the `PoolFactory.sol` contract to withdraw rewards that were accrued before.
However, it also triggers the `_accrueReward()` function on `PoolMaster`, which updates the user's rewards amount since there is no restriction that it can't be called when the pool is in the `Closed` state.
If there are other active pools, the governor can distribute rewards to the `PoolFactory` contract, and the user can immediately drain the amount of rewards after the factory balance is increased.

Recommendation

We recommend mitigating this issue by restricting the accrual of rewards when the pool has a `Closed` status, as done in the `_accrueInterestVirtual()` function, and updating the rewards state when the pool transitions to a `Closed` status.

```
function _accrueReward() internal {
    // get reward assets array
    if (
        totalSupply() != 0 &&
        rewardAssetInfo.lastRewardDistribution != 0 &&
--      block.timestamp > rewardAssetInfo.lastRewardDistribution
++      block.timestamp > rewardAssetInfo.lastRewardDistribution
++      && _info.status != PoolStatus.Closed
    ) {
      RewardAsset.RewardAssetData[] memory _rewardAsset =
                                          getRewardAssetInfo();
      for (uint256 i; i < _rewardAsset.length; i++) {
        if (_rewardAsset[i].rate != 0) {
          rewardAssetInfo.updateMagnifiedRewardPerShare(
            _rewardAsset[i].asset,
            block.timestamp - rewardAssetInfo.lastRewardDistribution,
            _rewardAsset[i].rate,
            REWARD_MAGNITUDE,
            totalSupply()
          );
        }
      }
      rewardAssetInfo.lastRewardDistribution = block.timestamp;
    }
}
```

```
function _accumulativeRewardOf(
    address _rewardAsset,
    address account
  ) internal view returns (uint256) {
    uint256 currentTime = block.timestamp;
    uint256 currentRewardPerShare =
                rewardAssetInfo.magnifiedRewardPerShare[_rewardAsset];
    uint256 index = rewardAssetInfo.addressIndex[_rewardAsset];
    uint256 rate = rewardAssetInfo.rewardAssetData[index].rate;
    if (
      totalSupply() != 0 &&
      rewardAssetInfo.lastRewardDistribution != 0 &&
      currentTime > rewardAssetInfo.lastRewardDistribution &&
--    rate != 0
++    rate != 0 && _info.status != PoolStatus.Closed
    ) {
      uint256 period = currentTime -
                                rewardAssetInfo.lastRewardDistribution;
      currentRewardPerShare +=
                    (REWARD_MAGNITUDE * period * rate) / totalSupply();
    }
    return
      ((balanceOf(account) * currentRewardPerShare).toInt256() +
        rewardAssetInfo.magnifiedRewardCorrections[_rewardAsset][account])
        .toUint256()
        / REWARD_MAGNITUDE;
    }
```

```
function _accrueRoundReward(uint256 roundDate) internal {
    RepaymentRoundInfo storage roundInfo = repaymentRoundInfo[roundDate];

    uint256 bondId = roundInfo.bondTokenId;
    uint256 lastDistribution =
        rewardAssetInfo.roundLastDistribution[bondId];
--  if (totalSupply() != 0 && block.timestamp > lastDistribution
--      && roundInfo.debtAmount > 0) {
++  if (totalSupply() != 0 && block.timestamp > lastDistribution
++      && roundInfo.debtAmount > 0 && _info.status != PoolStatus.Closed
++  ) {
        RewardAsset.RewardAssetData[] memory _rewardAsset =
                                            getRewardAssetInfo();
        // update round reward per share accumulated since first request
        for (uint256 i; i < _rewardAsset.length; i++) {
            rewardAssetInfo.updateMagnifiedRoundRewardPerShare(
            _rewardAsset[i].asset,
            bondId,
            block.timestamp - lastDistribution,
            _rewardAsset[i].rate,
            REWARD_MAGNITUDE,
            totalSupply()
            );
        }
        rewardAssetInfo.roundLastDistribution[bondId] = block.timestamp;
    }
}
```

Client's commentary

Fixed in e38ecca8.

H-4 High The borrower can set `lendAPR` to a minimum, use lenders' funds and pay almost nothing. Fixed

Description

PoolMaster.sol#L1088-L1090

The borrower can decrease the `lendAPR` value in the `PoolMaster.sol` contract via the `changeAPR()` function. However, the value that was set is applied under the next condition:

```
while (block.timestamp + 2 * minimumNoticePeriod > applyDate) {
    applyDate += repaymentFrequency;
}
```

It means that `applyDate` will be increased if the current time plus two notice periods is greater than `applyDate.` Borrowers can abuse lenders if they set `minimumNoticePeriod` to `zero` and decrease `APR` immediately after that.

Consider the following scenario:

No lenders used `requestWithdrawals` in the current repayment round. The borrower waits for a few blocks to the `currentRepaymentDate` timestamp and uses `changeNoticePeriod(0)` to set `noticePeriod` to `zero`, then immediately uses `changeAPR(1)`.
Lenders didn't have time to use `requestWithdrawal()` in the current round. With a new round approaching, the borrower can now use lender funds at ~0% APR for the next ten days (based on a `repaymentFrequency` value) and is exempt from paying a `penaltyAmount` since the next repayment date was set in the current round (when the lender made a request).

Recommendation

We recommend mitigating the issue by incrementing the timestamp of the apply date when the `minimumNoticePeriod` value reaches zero.

```
function changeAPR(
    uint256 _newAPR
) external onlyActive onlyBorrower nonSameValue(_newAPR, lendAPR) {
    if (_newAPR == 0) revert WrongNumber();

    // if current repayment date is zero or
    // no supplies yet the decrease apr request is applied immediately
    uint256 applyDate = currentRepaymentDate();
    if (_newAPR > lendAPR || applyDate == 0) {
        // in case an increase request is submitted,
        // the change is applied immediately.
        lendAPR = _newAPR;

        // remove the old request
        _aprChanges[_lastRequestIndex] = 0;
        emit APRChanged(_newAPR);
    } else {
++      if (minimumNoticePeriod == 0) applyDate
++          += repaymentFrequency;
        // APR decrease change is applied in the upcoming repayment
        // after 2 notice periods
        while (block.timestamp + 2 * minimumNoticePeriod > applyDate) {
            applyDate += repaymentFrequency;
        }
        _aprChanges[applyDate] = _newAPR;
        _lastRequestIndex = applyDate;
        emit APRChangeRequested(_newAPR, applyDate);
        }
    }
```

Client's commentary

Fixed in f810eb0b.

H-5 ● High **Logical issues in the accrual of interests process** Fixed

Description

Invariant `_info.borrows` has logical issues in the accrue interests process. `_info.borrows` needs to represent the full borrows in the pool, but it doesn't include calculating future percentages for the current repayment round.

This leads to a situation when the `_borrows` value (from the doc: "calculate the interest accrued for lenders that requested repayment up to the current ts") exceeds the `_info.borrows` value (representing the total borrows in the pool).

The proof of concept is provided in the attachment.

Recommendation

We recommend fixing it.

Client's commentary

Fixed in af6324b5.

Medium

M-1 ● Medium **An unlimited protocol fee and penalty rate** Acknowledged

Description

PoolFactory.sol#L396

PoolFactory.sol#L409

The owner has the ability to assign a value of 100% to both the `penaltyRate` and `protocolFee` variables.

Recommendation

We recommend adding limits for the `_newPenaltyRate` and `_newProtocolFee` values.

Client's commentary

The value is set by stakeholders using a multi-signature mechanism with a minimum of three confirmations.

M-2 ● Medium There are no limits for variables Acknowledged

Description

There are no limits for the following variables:

`_exchangeRate`:

BondNFT.sol#L84

`_newGracePeriod`:

PoolFactory.sol#L385

`_rate`:

PoolFactory.sol#L438

PoolMaster.sol#L1053

`period`:

PoolFactory.sol#L457

PoolMaster.sol#L1030

`auctionDuration_`:

Auction.sol#L138

Auction.sol#L303

Recommendation

We recommend adding upper and lower limits for the variables.

Client's commentary

The value is set by stakeholders using a multi-signature mechanism with a minimum of three confirmations.

M-3 ● Medium **A missing check that the bidder is in the whitelist in the `increaseBid` function** Fixed

Description

A check that the `bidder` is in the whitelist in the `increaseBid` function is missing.

Thus, if the last bidder has been deleted from the whitelist, the user can front-run every new bid and call the `increaseBid` function to increase the current highest bid.

Recommendation

We recommend adding the `checkBidder(pool)` modifier for the `increaseBid` function.

Client's commentary

Fixed in 83c966b6.

M-4 ● Medium No checking of input variables Acknowledged

Description

PoolMaster.sol#L262

PoolMaster.sol#L263

The `repaymentFrequency` value can be set to any value higher than 0 in the `__init` function, so it can be set to "1."

The `getNextRepaymentDate` function will most probably run out of gas. Each iteration of the circle will consume approximately 5200 gas units.
For example, the Avalanche block gas limit is `15_000_000 / 5200 = 2884 seconds = 48 min`.
If `repaymentFrequency` is set to 1 and `minimumNoticePeriod` is less than ~48 mins, this function will always run out of gas 48 mins after the deployment.

Also, only the borrower can change `repaymentFrequency.`

Recommendation

We recommend adding limits for the `repaymentFrequency` and `minimumNoticePeriod` values.

Client's commentary

We are in contact with the borrowers who are creating pools, and we will provide the requirements for them to avoid such issues.

M-5 ● Medium No price slippage protection Acknowledged

Description

PoolMaster.sol#L279-L312

The `supply` function calculates a `tokensAmount` value to mint:

```
uint256 tokensAmount = _amount.divDecimal(_info.exchangeRate);
_info.borrows += _amount;
```

If the exchange rate increases, the `tokensAmount` value will be less than intended.

Let's consider the following case:

1. A user sends a transaction;
2. This transaction is stuck for a long time;
3. The `exchangeRate` changes significantly;
4. The user's transaction is minted, but due to a significant passage of time, they receive fewer tokens than originally intended.

Recommendation

We recommend adding `uint256 minOutputTokensAmount` to the `supply` function and modifying`supply` to:

```
if(tokensAmount < minOutputTokensAmount) {
    revert WrongNumber();
}
```

Client's commentary

No actions are required from our side.

M-6 ● Medium The `supply()` function can be called even when the `_fullRepaymentDate` value is set to a non-zero value. **Fixed**

Description

PoolMaster.sol#L279

In the `PoolMaster.sol` contract, the governance can set the `_fullRepaymentDate` value to a non-zero value. It means that the borrower must repay all debts by this date.
It makes no sense to supply funds into the pool for the lender since the `requestWithdrawal()` function can't be used. The lender can receive their funds only after the borrower has repaid all the debt; however, the `supply()` function doesn't restrict it like the `requestWithdrawal()` function.

```
if (_fullRepaymentDate != 0) revert ActionNotAllowed();
```

Recommendation

We recommend implementing the following additional check in the `supply()` function:

```
function supply(uint256 _amount) external ... {
    if (msg.sender == borrower) revert ActionNotAllowed();
++  if (_fullRepaymentDate != 0) revert ActionNotAllowed();
    ...
}
```

Client's commentary

Fixed in ba1cf8bb.

M-7 ● Medium The penalty accrues only when `block.timestamp` > `currentRepaymentDate` Fixed

Description

PoolMaster.sol#L299

When the lender supplies an asset into the pool, `currentRepetitionDate` is updated with `repayFrequency`.

```
_currentRepaymentDate = block.timestamp + repaymentFrequency;
```

After that, the lender can use the `requestWithdrawal()` function to request assets from the borrower.

PoolMaster.sol#L319

The pool becomes `Overdue` in 2 cases:

- either the current timestamp is between the `currentRepaymentDate` and the `currentRepaymentDate` + `gracePeriod` values and the debt of the current round is non-zero
- the current timestamp is between the `currentRepaymentDate` and the `currentRepaymentDate` + `gracePeriod` values, and the `_fullRepaymentDate` value is non-zero.

Additional interest is accrued during the `Overdue` period.

```
uint256 currentRepDate = currentRepaymentDate();
    if (
        block.timestamp > currentRepDate &&
        block.timestamp < currentRepDate + gracePeriod &&
        (repaymentRoundInfo[currentRepDate].debtAmount > 0
         || _fullRepaymentDate != 0)
    ) {
        return PoolStatus.Overdue;
```

PoolMaster.sol#L1251-L1254

If governance calls the `requestFullRepayment()` function with the `_fullRepaymentDate` value < the `currentRepaymentDate` value, the pool will set the status to `Overdue` only when the `timestamp` value > the `currentRepaymentDate` value (the timestamp of the lender's request). Therefore the `penaltyAmount` value will not accrue during the period when `_fullRepaymentDate` < `timestamp` < `currentRepaymentDate`.

Recommendation

We recommend adding the following check:

```
function requestFullRepayment(uint256 _repaymentDate)
    external onlyGovernor onlyActive
{
    if (_fullRepaymentDate != 0 || _currentRepaymentDate == 0)
        revert ActionNotAllowed();

    /// Disallow repayment date in past;
    uint256 currentRepDate = currentRepaymentDate();
    if (block.timestamp > _repaymentDate) revert InvalidArgument();
++  if (currentRepDate > _repaymentDate) revert InvalidRepaymentDate();
    /// Repayment date is either future date,
    /// or the current unpaid round date;
    _currentRepaymentDate = currentRepDate;
    _fullRepaymentDate = _repaymentDate;
    emit RepaymentRequested(_repaymentDate);
}
```

Client's commentary

Fixed in aa994647.

M-8 ● Medium **The `feeAmount` remains `unpaid` after the auction is resolved.** Acknowledged

Description

When the borrower uses the `repay()` or `repayAll()` functions, it transfers the `fee` to `treasury` via the `_transferRepayment()` function that was accrued during the time when the lender's money was being used. However, when the pool is `closed` and the borrower doesn't use the `repay()` or `repayAll()` function, the fee stays `unpaid`, and `treasury` doesn't receive any fee.

Recommendation

We recommend collecting the `fee` when the borrower doesn't use the `repay()` or `repayAll()` functions.

Client's commentary

We don't collect fees in case of `Pool Default`.

M-9 ● Medium **The auction can be blocked in case `lastBidder` occurs in the `USDC` blacklist.** Acknowledged

Description

[Auction.sol#L182]()

[Auction.sol#L235]()

[Auction.sol#L276]()

The protocol uses the `USDC` token; the `USDC` contract has a blacklist. If the `USDC` contract blacklists the `lastBidder` address, `lastBid` can't be received by him. Therefore, no one can make a bid.

Moreover, neither the `resolveAuctionWithoutGoverment()` function nor the `resolveAuction()` function with `resolution == false` can be used in case the `lastBidder` was blacklisted.

Recommendation

We recommend implementing withdraw functionality, so that the `lastBidder` can withdraw their bid themselves, without breaking the functionality of the auction.

Client's commentary

We don't implement any changes (that issue will block the auction), but after the auction period ends, we will be able to call the `resolveAuction` function (only with a positive resolution) and transfer the tokens in the pool.

M-10 ● Medium The limits for `_newAPR` Acknowledged

Description

PoolMaster.sol#L1075

PoolMaster.sol#L1261-L1274

The `changeAPR` function checks the `_newAPR` value only when it equals 0.
The `accrueInterest` is public and can be called at any frequency.

POC:

1. The borrower front-runs first the `supply` function to the pool and calls the `changeAPR` function with some small value of `_newAPR` and is applied immediately.
2. The lender transaction with the `supply` function is executed after `_newAPR` is set.
3. The borrower starts to call `accrueInterest` with some sufficient frequency.
4. After every call of `accrueInterest`, the APR percentage is 0.

For example:

- The amount of borrowing = 10000000e6
- lendAPR = 1e10
- The `accrueInterest` call frequency = 5 min
  If `lendAPR = 1e9` with the same borrow amount, the `accrueInterest` function needs to be called every 50 minutes, and so on.

Also, if the borrower has set the `_newAPR` value too high by mistake, then they will have to pay the erroneous APR `2 * minimumNoticePeriod` which is 24 days by default.

Recommendation

We recommend imposing limits for the `_newAPR` value and updating the `lendAPR` value after a pending period, during which the borrower can cancel the update.

Client's commentary

In that case, the lender won't receive any interest, but the full principal is guaranteed. Also, the borrower will become undesirable for Clearpool. Even if the createPool can be triggered by anyone, only the whitelisted borrowers will be able to receive deposits.

M-11 ● Medium A possible revert of the `supply` function if `depositCap` is 0. Fixed

Description

PoolMaster.sol#L285-L295

PoolMaster.sol#L986-L989

The changeDepositCapacity `function` allows the borrower to set a null value to `depositCap`, which means unlimited deposits without cap validation.
The `supply` function has the wrong check for the cases when `depositCap` equals 0:

```
    if (_amount == 0 || minDeposit > _amount
       || _amount + currentSize > depositCap)
      revert WrongNumber();
```

In this case, when the value of `depositCap` is 0, all user calls to the `supply` function will be reverted.

Recommendation

We recommend updating this check.

Client's commentary

Fixed in c022826b.

Low

L-1 ● Low The `vaults` variable may not be updated **Fixed**

Description

PoolFactory.sol#L232

PoolFactory.sol#L427

The `vaults` variable contains an array of pools created by the corresponding borrower.

However, if we call the borrower address replacement function, the owner of the pools in `vaults` will not be replaced with the new address.

Recommendation

We recommend updating the `vaults` variable with the 'changePoolBorrower' function.

```
   function changePoolBorrower(
     address _pool,
     address _newBorrower
   ) external onlyOwner onlyWhitelistedBorrower(_newBorrower) {
     if (!isPool[_pool]) revert ActionNotAllowed();
 +   vaults[_newBorrower].push(_pool);
     return IPoolMaster(_pool).changeBorrower(_newBorrower);
   }
```

Client's commentary

Fixed in 3185c2bc

L-2 ● Low Division by zero Acknowledged

Description

RewardAsset.sol#L94

RewardAsset.sol#L107

Decimal.sol#L25

The `totalSupply` input value in the `updateMagnifiedRewardPerShare` and `updateMagnifiedRoundRewardPerShare` functions in the `RewardAsset` contract could be equal to zero.

Recommendation

We recommend adding zero checks for the `totalSupply` and `decimal` values.

Client's commentary

We are calling the function only if `totalSupply` is greater than zero.

L-3 ● Low **A two-step ownership transfer** Acknowledged

Description

[WhitelistControl.sol#L175-L180]

[Auction.sol#L7]

[PoolFactory.sol#L6]

The contract owner can call the `transferOwnership` function with an inactive address, leading to loss of access to the contract. `OwnableUpgradeable.sol` also has a one-step transfer of ownership.

Recommendation

We recommend using the `Ownable2StepUpgradeable` contract.

Client's commentary

We don't have the issue as we are using a multi-sig wallet and are in direct touch with the borrower.

L-4 ● Low Reentrancy risk Acknowledged

Description

BondNFT.sol#L137

BondNFT.sol#L160

The `safeTransferFrom` and `safeBatchTransferFrom` functions can potentially allow a reentrancy attack when transferring tokens to an untrusted contract when invoking {onERC1155Received} on the receiver.

Recommendation

We recommend adding a reentrancy guard modifier.

Client's commentary

The pool functionality in the `Bond` contract is solely designated for minting and burning bond tokens. Transfers between accounts are the responsibility of the users themselves.

L-5 ● Low Unused error types Fixed

Description

Auction.sol#L96

PoolFactory.sol#L75

Recommendation

We recommend removing the unused error types.

Client's commentary

Fixed in 1193f059

L-6 ● Low Typos Fixed

Description

- `borrowwer` to `borrower`:
  PoolFactory.sol#L230
- `accross` to `across`:
  PoolFactory.sol#L302
- `penality` to `penalty`:
  PoolFactory.sol#L392
  PoolFactory.sol#L394
- `supplyed` to `supplied`:
  PoolMaster.sol#L32
- `redeeemed` to `redeemed`:
  PoolMaster.sol#L34
- `repaing` to `repaying`:
  PoolMaster.sol#L467
- `penality` to `penalty`:
  PoolMaster.sol#L803
- `mininum` to `minimum`:
  PoolMaster.sol#L992
- `occured` to `occurred`:
  RewardAsset.sol#L19
- `occured` to `occurred`:
  RewardAsset.sol#L25
- `goverment` to `government`
  Auction.sol#L225
  Auction.sol#L222

Recommendation

We recommend fixing the typos.

Client's commentary

Fixed in 40215d3e

L-7 ● Low The NatSpec updates Fixed

Description

NatSpec is missing for all functions in `RewardAsset.sol`:

RewardAsset.sol

NatSpec is missing for `WhitelistControlChanged`:

PoolFactory.sol#L65

NatSpec is missing for `bidIncrementInfo`:

Auction.sol#L58

Natspec has an incorrect description for emits (replace `LenderWhitelisted` with `BorrowerWhitelisted`):

PoolFactory.sol#L266

Recommendation

We recommend updating NatSpec.

Client's commentary

Fixed in 3765e2d8

L-8 ● Low An invalid error type Fixed

Description

PoolFactory.sol#L256

PoolFactory.sol#L271

These functions have the `NotSameValue()` error, despite having input parameters of type `address`.

Recommendation

We recommend using the `NotSameAddress()` error.

Client's commentary

Fixed in 4a8556ee

L-9 ● Low **An additional check** Fixed

Description

Auction.sol#L228

Auction.sol#L252

The `resolveAuctionWithoutGoverment` function has a check:

```
if (currentAuction.end == 0) revert AuctionNotStarted();
```

However, the `resolveAuction` function does not have this check.

Recommendation

We recommend adding the check.

Client's commentary

Fixed in 855f48e8

L-10 ● Low A memory gap for the upgradable contract is missing Fixed

Description

WhitelistControl.sol

The `WhitelistControl`contract is an ungradeable contract.

The recommendation from OpenZeppelin is to have a memory gap for upgrades.
(https://docs.openzeppelin.com/contracts/4.x/upgradeable#storage_gaps)

Recommendation

We recommend adding a gap.

Client's commentary

Fixed in 1193f059

L-11 ● Low **The `nonReentrant` modifier should occur before other modifiers.** Fixed

Description

PoolMaster.sol#L279

The `nonReentrant` modifier comes after the `onlyActive` modifier to avoid reentrancy in other modifiers (in case of contract update).
The best practice is to put `nonReentrant` before all other modifiers.

Recommendation

The recommend setting the `nonReentrant` modifier before all other modifiers.

Client's commentary

Fixed in 1193f059

Conclusion

Altogether, the audit process has revealed 5 HIGH, 11 MEDIUM, and 11 LOW severity findings.

Disclaimer