ExactlySmart Contract Audit

Exactly



Exactly RewardsController January 2023

Smart Contract Audit

V230404

Prepared for Exactly • April 2023

- 1. Executive Summary
- 2. Assessment and Scope
- 3. Summary of Findings
- 4. Detailed Findings
- EXR-1 Insecure downcast in rewards accounting
- EXR-2 Reward indexes view function might return stale data
- EXR-3 Unclaimed rewards might become unrecoverable
- EXR-4 Cost to interact with markets is considerably increased
- EXR-5 Configuration process does not guarantee rewards availability
- EXR-6 Outdated test suite
- EXR-7 Zero token transfers triggered by claim logic
- 5. Disclaimer

1. Executive Summary

In January 2023, Exactly engaged Coinspect to perform a source code review. The objective of the project was to evaluate the security of the RewardsController contract, which implements the rewards distribution.

Exactly will distribute rewards for floating borrow and deposit positions, as well as fixed borrows. Every reward-related operation (allocation, accrual and claim) will be handled by the RewardsController contract.

The following issues were identified during the initial assessment:

High Risk	Medium Risk	Low Risk
Open	Open	Open
0	0	0
Fixed	Fixed	Fixed
0	0	3
Reported	Reported	Reported
0	0	3

Coinspect identified three low-risk and four informational issues, some involving possible business impacts but based on our analysis of the probabilities (Likelihood) we decided to reduce their risks. Below is a brief summary: EXR-1 depicts how an arithmetic overflow can break the reward accounting process. EXR-2 is about how index retrieval would likely return stale values. EXR-3 explains the lack of mechanisms to rescue unclaimed rewards, if necessary.

2. Assessment and Scope

Exactly is developing a new token reward system for users that perform several actions across the active markets. Floating and fixed borrow operations as well as floating deposit operations will start accruing interests that could be claimed afterwards. This functionality is implemented by a RewardsController contract, which handles rewards allocation and provides the reward claim functionality.

The audit started on Jan 23, 2023 and was conducted on the next branch of the git repository at https://github.com/exactly/protocol as of commit da9ddd7f847ebdad24c21fdae049f9d4523e9313 of Jan 24, 2023.

The audited files under the contracts directory have the following sha256sum hash:

faaec0ae09afa35cc453667a33b3ad2b2e642dc61045f0137f7e11b6db87a82f ./RewardsController.sol

Coinspect reviewed the security of the newly added RewardsController contract, and the modifications included to the market contract that will update the rewards system states. It is worth mentioning that the Exactly team stated that this implementation is intended to be deployed on the Optimism Chain, which is relevant due to the current gas consumption of the entire rewards logic (update, allocation, and distribution).

The test suite provided for the RewardsController was thorough and clear to understand. However, as new functionalities are added to the protocol, Coinspect strongly recommends updating previous tests (markets, auditor, etc.) to contemplate the existence of the rewards contract and evaluate its impact on key parameters such as gas consumptions or token flows, EXR-6.

The RewardsController contract is in charge of retrieving, processing and allocating rewards to each user. It uses a global borrow and deposit index system to track the current rewards state. Relevant data is collected via two hooks that update borrow and deposit operations performed by each market. The mentioned hooks were added to the market contract and are triggered when depositing (floating only), borrowing (floating and fixed), repaying, minting, transferring, and liquidating among other relevant actions. In relation to this, Coinspect detected that the operating costs were increased considerably by the hooked callbacks, EXR-4.

The implementation of the RewardsController is based on the rewards system designed by Aave. Coinspect identified several differences between both implementations:

- 1) Exactly allocates rewards via the previewAllocation() function that computes the result of two integrals yielding in an exponential equation.
- 2) Aave has only one update hook whereas Exactly separated this process for borrow and deposit operations.
- 3) Only self-accrued rewards could be claimed on Exactly. Aave includes a delegated claiming system.
- 4) Exactly sets all the parameters of a distribution in a single call. Aave has setters for each distribution's parameter.
- 5) Because Exactly added the borrow and deposit operations differentiation, all loops are nested one more level than Aave's.
- 6) Exactly allows sending zero token transfers while claiming from an account with no accrued rewards, EXR-7. On the contrary, Aave returns immediately without executing further logic if the accrued amount is zero.
- 7) Exactly allows adding multiple reward tokens, looping over all the allowed tokens while performing a claim. Aave allows passing the claimed reward address to make single-token claims.
- 8) When claiming, Exactly updates and processes the claim of the first market, then moves to the next one and performs the same two steps. Aave first updates all the markets and then processes all the claims.
- 9) Exactly does not revert if downcasting from an uint256 when computing the reward accrual (EXR-1) results in an overflow, whereas Aave reverts if that happens.

In addition to the differences mentioned before, more aspects of the RewardsController were reviewed. Regarding how borrow and deposit positions' global indexes are retrieved, Coinspect found out that the external view function implemented to do so could return stale values affecting external sources consuming this data, EXR-2.

In terms of how each distribution period is configured, an access controlled config() function is included in the controller's contract. It is only callable by the

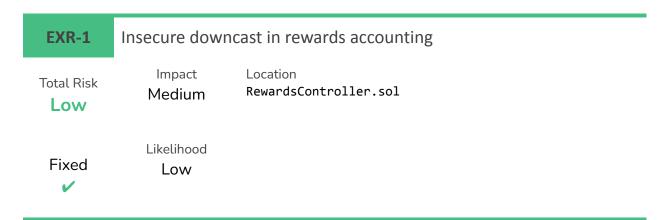
owner and allows setting (and modifying) each distribution's parameter. Coinspect identified that upon adding a new distribution period, reward tokens might be unavailable by the time it begins, EXR-5. Also, the owner has to set each supported market that will receive information via the handleBorrow and handleDeposit hooks. Those hooks are not access controlled and use the msg.sender assuming that the call will come from a market, however anyone can deploy a spoof market that calls those hooks. Coinspect didn't identify any exploit scenario because states will be updated only for those markets previously added via a config() call.

In addition, once funds are sent to the RewardsController, the only way they can exit is via rewards claims. In the event of experiencing an attack, having an undisclosed bug or any other unexpected contingency funds won't be recoverable to its safeguard, EXR-3.

3. Summary of Findings

ld	Title	Total Risk	Fixed
EXR-1	Insecure downcast in rewards accounting	Low	~
EXR-2	Reward indexes view function might return stale data	Low	V
EXR-3	Unclaimed rewards might become unrecoverable	Low	~
EXR-4	Cost to interact with markets is considerably increased	Info	!
EXR-5	Configuration process does not guarantee rewards availability	Info	!
EXR-6	Outdated test suite	Info	!
EXR-7	Zero token transfers triggered by claim logic	Info	✓

4. Detailed Findings



Description

The current account's index (either for deposits or borrows) might overflow because a downcast from a uint256 to a uint104 is performed insecurely. As a result, the update logic will erase previously accruals. Also, accrued rewards could overflow (because of a downcast from uint256 to uint128) meaning that less rewards than expected would be effectively accrued.

Indexes are updated every time an update() call is triggered, either by the hooks or by a claim. While updating the rewards accountancy, new indexes are retrieved by calling previewAllocation():

```
(uint256 borrowIndex, uint256 depositIndex, uint256 newUndistributed) =
previewAllocation(rewardData, market);
```

Later, the indexes are insecurely downcast to uint104:

```
uint256 newAccountIndex;
...
if (ops[i].operation == Operation.Borrow) {
   newAccountIndex = rewardData.borrowIndex;
} else {
   newAccountIndex = rewardData.depositIndex;
}
if (accountIndex != newAccountIndex) {
   rewardData.accounts[account][ops[i].operation].index = uint104(newAccountIndex);
```

The logic inside the preview function computes the new indexes based on current market and reward conditions (such as balances, supplies and configuration

parameters). In the event of having a high amount of custom reward tokens, under specific configuration parameters, the preview allocation might return a value higher than type(uint104).max causing an overflow upon its casting.

In addition, the rewards accrual will be mistakenly calculated for the next update as the account's recently assigned index will be smaller than the latest one:

Moreover, the current account's accrued rewards are increased by the downcast rewardsAccrued variable (from uint256 to uint128).

This process would not be perceived by users because the Accrue event emitted uses uint256 variables instead of the values effectively used.

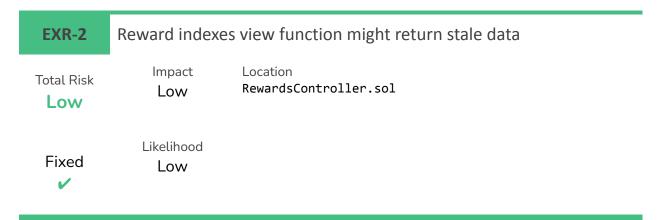
This issue is considered low-risk as the scenarios that would allow exploiting it are unlikely.

Recommendation

Perform the downcasting safely by reverting if the uint256 to be cast is bigger than type(targetUint).max. Modify the emitted parameters on the Accrue event so it shows the correct values.

Status

Fixed in commit 120163937a5d6e28954c01fd3e6b645e8f933886 by updating the event and reverting on overflow.



External sources consuming the returns of rewardIndexes() to perform other time-sensitive actions will likely use stale data, affecting subsequent actions they might conduct.

The rewardIndexes() is an external view function that returns the current deposit and borrow indexes. As they are updated only when a handleDeposit() or handleBorrow() hook is triggered, if the update frequency is not high enough, the returned data will be outdated and won't depict the current rewards controller indexes. Unknown scenarios might arise as the retrieved data won't be up-to-date, for example:

- An external oracle gets the current indexes via rewardIndexes() and performs calculations with their values. If the markets have low volume of interactions, the returned data will be likely outdated yet consumed by the external source periodically.

It is worth mentioning that the rewardsData() function returns the last update of a distribution but this functionality is not provided in rewardIndexes().

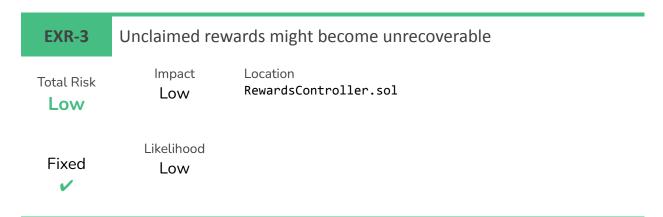
Recommendation

Clearly document this behavior. Prevent using the rewardIndexes() returns as inputs for time-sensitive calculations (e.g. oracles).

Status

Fixed in commit da90aa7b19dcc8c5e41fb0dd9e8becdf635b6cc9.

An external previewAllocation function was added that retrieves the indexes directly by calculating their values with the most recent states instead of getting them via each distribution mapping.



Reward tokens are sent in advance to the Controller contract, on each market the administrator has to set the current Controller's address to start the rewards process. Unclaimed rewards might get stuck if the address of the Controller is changed on a Market.

The RewardsController manager first sends the tokens to that contract and once the distribution period begins, users start to accrue rewards on their positions. When the claiming period is up, they can start claiming the reward tokens. In the event of having to change the current rewards contract (e.g. because of a contingency), the admin has no way to get the rewards back.

It is worth mentioning that a rescue mechanism could be by manually adding a spoof market address via config() and assigning all the remaining rewards to the administrator. However, if the rewards are at risk for some reason, this process could take critical time.

Recommendation

Consider adding a rescue function that could only be executed if previously defined conditions are met, to allow recovering the funds in case of an emergency.

Status

Fixed in commit 81a11e3ce1e97623b625b956f155de304001791c by allowing the admin to withdraw the funds.



Every key action of each market will trigger an update of the rewards controller by a hook call, and as a consequence, performing transfers, borrows and even liquidations becomes considerably more expensive.

Due to the addition of the rewards feature, the handleDeposit() or handleBorrow() hooks were added to most functions of each market. Those hooks perform updates of the current rewards state. Each update has nested loops that go over each reward token, operation and market.

Coinspect evaluated the costs to perform some of the core actions of a market and compared their costs without having a rewards controller enabled:

- transferFrom()
 - a. 2 pools with 2 reward tokens: 200k
 - b. 12 pools with 2 reward tokens: 433k
 - c. Controller disabled: 37k
- liquidate()
 - a. 2 pools with 2 reward tokens: 640k
 - b. Controller disabled: 320k

Also, Coinspect checked the costs to perform a rewards claim via claimAll() and the impact of previous updates. The updates are triggered by calling market.transferFrom():

Without previous update:

Gas used to claimAll: 686368

With previous update

TransferFrom Cost: 521478 Gas used to claimAll: 478692

Performing the Claim first and then making a transfer

Gas used to claimAll: 686368 TransferFrom Cost: 313794

In the event of having considerably high gas prices, this mechanism can be abused in many different ways. For example:

- 1) Users willing to claim rewards can create liquidatable positions, expecting them to be liquidated by a third party, which would update their rewards state and reduce the gas cost to make the claim.
- 2) Cost to liquidate users whose reward state was not updated will be higher, reducing the incentive to perform that liquidation. Users about to be liquidated have no incentive to make actions that call update() via any hook as the incentive to perform a liquidation over their position could increase.

Also, as non approved users can call transferFrom passing zero tokens, this opens a new scenario where externals can update other's states and pay for its cost. This transfer mechanism became recently popular known as "Zero Token Transfer Phishing Scam".

This issue is considered informational only as Exactly intends to deploy the rewards system on the Optimism chain.

Note these updated costs could impact the scenarios described in EXA-06, EXA-34 previously reported issues and make them even worse.

Recommendation

Optimize the update process to reduce its consumption and reevaluate incentives alignment if this implementation is going to be deployed on a chain with expensive gas costs such as Ethereum's mainnet.

Status

Open.

Total Risk Info Likelihood Fixed ! Configuration process does not guarantee rewards availability Location RewardsController.sol Likelihood -

Description

The rewards contract can start a distribution period without funds because the distribution's configuration is agnostic to the balance of each reward token.

Rewards are meant to be sent to the contract in advance and the balance of tokens held in the contract must be the same as the distribution's totalDistribution. If the amount sent to the contract is less than the totalDistribution value, only the first users would be able to claim as the contract might not meet the required transfer amount for the last ones.

The parameter targetDebt is used while calculating the reward allocation and current borrow and deposit position indexes. Because of this, the amount set by the manager should consider the same decimals as the reward token used for the configured distribution. A misalignment on the targetDebt's decimals would result in a mistakenly calculated amount of rewards.

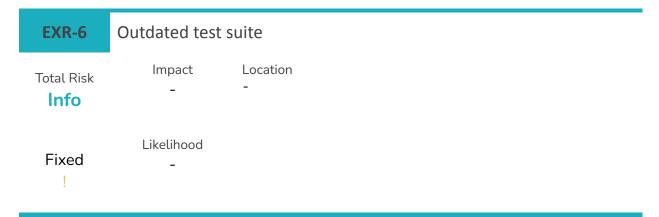
In addition, if the administrator wants to change only one parameter of a distribution, they need to pass again the entire configuration struct when calling config(). This process is error-prone.

Recommendation

Check that the initial amount sent to the contract is the same as the initial totalDistribution. Clearly document how each configuration parameter should be set. Evaluate the need to include setters for specific distribution parameters.

Status

Partially fixed in c25c305a8a21e08ea0355ac5a05e4d3b93c01b22. It now uses a baseUnit instead of the decimals value and keeps the mintingRate and the rewardAssets with the same decimals. No actions will be performed in rewards availability.



The development suite includes tests for each main functionality but they are not updated to consider the event of having a RewardsController contract.

Adding the rewards contract modifies the overall protocol's functionality as several functions now include updating hooks. In addition, users claiming rewards might decide to modify their positions according to their current rewards status.

Coinspect identified by updating the Market's test suite that using a rewards contract increases considerably the gas costs of each action, for example. Also, some tests were failing after setting a RewardsController address.

Recommendation

Update the protocol's test suite to consider the scenario of having a rewards contract deployed and operational.

Status

Open.

Description

The claim process can be abused to overpopulate the latest transactions feed of the victim with zero token transfers having the RewardsController as the sender.

Users that have no positions in any market will have no right to claim any rewards. This state can be leveraged by calling claimAll() targeting a victim and the contract will send a zero token transfer emitting the Transfer and Claim events.

The reward transferring process is handled inside the claim() function:

```
for (uint256 r = 0; r < rewardsList.length; ) {
  rewardsList[r].safeTransfer(to, claimedAmounts[r]);
  emit Claim(msg.sender, rewardsList[r], to, claimedAmounts[r]);
  unchecked {
    ++r;
  }
}</pre>
```

The malicious user must use an account without any reward-yielding positions as the claim process uses the msg.sender to estimate the rewards to accrue and transfer. Under this scenario, claimAll(victim) can be called and the following events for each reward token will be emitted:

```
Transfer(from: RewardsController, to: Victim, amount: 0)
Claim(account: Attacker, reward: MockERC20, to: Victim, amount: 0)
```

The following script was used to get the mentioned event emissions:

```
function test_ClaimFakeEvents() external {
    vm.warp(5 days);
```

```
vm.prank(ATTACKER);
rewardsController.claimAll(VICTIM);
}
```

Recommendation

Perform the transfer and event emission only if the claimedAmounts[r] is greater than zero.

Status

Fixed in commit 2999cd54914b62e7353259654d26b55c12dd5608. It now transfers only when claimedAmounts[r] is greater than zero.

5. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.