Rumpel — Point Tokenization Vault

Audited by Ethan Bennett

April 2024

Contents

About Darklinear Solutions

Darklinear Solutions provides unrivaled security for blockchain applications, from the bytecode to the browser. With years of experience in smart contract development and traditional software engineering, we find the bugs that others miss. Learn more at darklinear.com.

Introduction

Rumpel is an on-chain vault for points-earning tokens (such as Pendle's YTs) that allows users to mint pTokens against their accruing points, turning the points into liquid assets for use across other DeFi applications. The protocol allows users to redeem their pTokens for the points-earning tokens' rewards after they have been acquired by the vault, and users can swap their reward tokens the other direction if they want the associated pTokens. It relies on off-chain architecture to account for the points earned by users while their assets are deposited.

This review was conducted on point-tokenization-vault at commit #e757804, from April 26-28, 2024. In addition to manual code review, the audit applied unit, invariant, and fuzz testing, which is described in detail in the Test Coverage section.

Disclaimer: This review is not a guarantee that the reviewed code can never be exploited, and any further development will require further review.

Risk Classification

The issues uncovered during the course of this review can each be described by one of the following two identifiers:

- **Low risk** findings represent issues that could lead to lost funds or improper contract functionality, but rely on an admin mistake to occur. Low risk findings may also consist of vulnerabilities or exploits that do not rely on an admin mistake, but have a low impact and a low likelihood of occurring.
- **Informational** findings are improvements that can be made in the code that do not have an appreciable impact on the user or the protocol.

Urgency according to risk classification

Risk level	Recommendation
High	Must fix
Medium	Should fix
Low/Informational	Could fix

Summary of Findings

Classification	Finding
Low risk (L-1)	Extremely high `rewardsPerPToken` (or low `_amountToConvert`) may result in a small amount of lost rewards
Low risk (L-2)	`isCapped` should be scoped to each points-earning token
Low risk (L-3)	Future implementation upgrades risk storage collisions in proxy
Low risk (L-4)	Valid Merkle proofs may fail if pTokens need to be paused by way of a zero-balance root update
Low risk (L-5)	`execute` should have additional safeguards
Low risk (L-6)	`pointsId` collision risk
Low risk (L-7)	`rewardsPerPToken` should not be updatable
Informational (I-1)	Unnecessary roles in `PToken.sol`

Findings

L-1: Extremely high `rewardsPerPToken` (or low `_amountToConvert`) may result in a small amount of lost rewards

Severity: Low risk

Context: PointTokenVault.sol#L157

Description: Because division will always round down in Solidity, `convertRewardsToPTokens` may mint zero pTokens in exchange for a non-zero amount of reward tokens. However, since the function properly handles the token decimals (assuming they all have `18`) by way of `FixedPointMathLib.divWadDown`, the impact of this issue is limited to extremely small values. For example, in order for a user to lose five reward tokens, the `rewardsRatePerPToken` would need to be set to `50000000000000000000`. Nonetheless, it is best to prevent dust loss whenever possible.

Recommendation: `convertRewardsToPTokens` should add a check like this one:

```
if (FixedPointMathLib.divWadDown(_amountToConvert, rewardsPerPToken) == 0) {
    revert AmountTooSmall();
}
```

Resolution:

- Rumpel: Fixed in commit #175a52c.
- Darklinear: Confirmed.

L-2: `isCapped` should be scoped to each points-earning token

Severity: Low risk

Context: PointTokenVault.sol#L43

Description: `isCapped` is the global value an admin sets to trigger caps on points-earning token deposits. If this value is set to `false` in the future and a single points-earning token needs a cap, setting `isCapped` back to `true` would make any points-earning token without a cap unusable until an admin explicitly sets it (since the default cap value is 0).

Recommendation: Consider scoping `isCapped` to each points-earning token:

```
mapping(ERC20 => bool) public isCapped;  // points-earning token => isCapped
```

Resolution:

- Rumpel: Fixed in PR #15.
- Darklinear: Confirmed.

L-3: Future implementation upgrades risk storage collisions in proxy

Severity: Low risk

Context: `PointTokenVault` storage layout

Description: Future implementations of `PointTokenVault` risk storage collisions, if such issues are not carefully avoided.

Recommendation: Consider using a namespaced storage layout to reduce the complexity (and the likelihood of mistakes) in future upgrades to `PointTokenVault`. EIP-7201, for example, is integrated with OpenZeppelin's Upgrades plugins, which will automatically validate the storage changes when upgrading the contract.

Resolution:

- Rumpel: Fixed in commit #a5d145e.
- Darklinear: Confirmed.

L-4: Valid Merkle proofs may fail if pTokens need to be paused by way of a zero-balance root update

Severity: Low risk

Context: PointTokenVault.sol#L210

Description: One of the larger systemic risks to the protocol is the possibility that a points-token issuer could blacklist it. In this scenario, the admin of the protocol will simulate a pause of the `pointsId` by setting its cap to 0 and pushing a new Merkle root with a balance of `0`. Note that in order for this to work, it will need to be done twice, so that the `prevRoot` will also reflect the new balance.

If a user attempts to validate a proof that was generated just before the admin pushes the zero-balance roots for an action unrelated to the blacklisted pToken, the user's proof will fail: it will not resolve to either of the new zero-balance roots. This is why the `prevRoot` exists in the first place, but if it were to be left with the non-zero-balance value, the blacklisted pToken would not be frozen.

Recommendation: Consider making it possible to pause the `mint` function in `PToken.sol` directly. This could be accomplished without changing any roles by first implementing a `pause` function on the `PToken` itself, then implementing a function in `PointTokenVault` (with restricted access) to execute it.

Resolution:

- Rumpel: "will keep the contract as is and will ensure that this point is considered in our internal response/emergency procedures. We've also added a script function for this situation for easy execution b66705d."
- Darklinear: Confirmed.

L-5: `execute` should have additional safeguards

Severity: Low risk

Context: PointTokenVault.sol#L235

Description: Although the protocol depends on a high degree of centralization generally, the `execute` function is particularly risky. In order to allow for unknown point redemption logic, `execute` uses `delegatecall` to reference an arbitrary external implementation, allowing anyone with access to the account registered as the `DEFAULT_ADMIN_ROLE` the ability to drain all funds with ease.

Recommendation: This could be made considerably more secure against attackers who might steal admin keys by introducing more factors into its authentication. For example, there could be an additional `REDEMPTION_APPROVER` role, which is responsible for setting `address pendingRedemption` to the address of the rewards token that is meant to be redeemed. Then, `execute` could require that `ERC20(pendingRedemption).balanceOf(address(this))` has increased after its `delegatecall`.

Of course, this would not mitigate all of the risk in a function like this, but it would make it substantially more difficult for a malicious external party to gain access to this function.

Resolution: This finding led to ongoing discussions within the Rumpel team about both the suggested mitigation and the approach to centralized role management across other critical functions.

L-6: `pointsId` collision risk

Severity: Low risk

Context: `pointsId`

Description: Since the `pointsId` is a deterministically packed string built from generic token metadata, it is possible that two tokens could have the same `pointsId`. It is critically important that each `pointsId` is unique, since much of the accounting throughout `PointTokenVault` depends on it.

This is low risk because of the centralized mechanism for setting the rewards data for a given `pointsId`: it is less likely that this problem would arise without being noticed given the level of direct human intervention in this version of the code. However, since some aspects of the protocol will be less centralized in time, the severity could worsen as a result.

Recommendation: Consider hashing the packed `bytes32` with a unique salt to reduce the likelihood of collisions.

Resolution:

- Rumpel: "for now, we'll take the route of being careful in our management of pointsIds off-chain, and acknowledge the risk. in future, as we decentralize, we might add more protections off-chain."
- Darklinear: Acknowledged.

L-7: `rewardsPerPToken` should not be updatable

Severity: Low risk

Context: PointTokenVault.sol#L226

Description: After an admin has redeemed a points-earning token's points, they will enable redemption functionality in the `PointTokenVault` by setting the `rewardToken`, `rewardsPerPToken`, and `isMerkleBased` values for the `pointsId`. It is possible for an admin to reset this data by calling `setRedemption` again, because vesting points-earning tokens will require `isMerkleBased` to be set to `false` once they have fully vested.

There is never a scenario when `rewardsPerPToken` will need to be updated, however. And since it will need to be included as a parameter anytime `isMerkleBased` is updated, it is prone to accidental corruption. If it changes, the reward distribution for that `pointsId` could be permanently contaminated.

Recommendation: If `rewardsPerPToken` is already set, `setRedemption` should ignore the supplied `rewardsPerPToken` in favor of its existing value.

If `rewardToken` can also be treated as immutable, it may be best to revert in `setRedemption` if any rewards data has been set for the `pointsId`, and implement a dedicated function for updating `isMerkleBased` specifically.

Resolution:

- Rumpel:
  - "One implicit feature of not considering the reward token immutable is that redemptions can be paused by setting the reward token address to zero. If we implement these changes, and we want the ability to pause redemptions, it would be nice to explicitly have a boolean in the redemption params that could pause redemptions."
  - "for this reason ^, and in the spirit of 1) minimizing short-term changes pre-launch & 2) similar to L-6 restricting functionality as we move from multisig to AVS, we'll keep the code unrestricted for the time being with v2+ locking more functionality out."
- Darklinear: Acknowledged.

I-1: Unnecessary roles in PToken.sol

Severity: Informational

Context: PToken.sol

Description: `PToken.sol` sets three admin roles: `MINT_ROLE`, `BURN_ROLE`, and `DEFAULT_ADMIN_ROLE`. However, these are all set to the `PointTokenVault`.

Recommendation: As long as the `PointTokenVault` is the only account with access to `mint` or `burn` a pToken, a `DEFAULT_ADMIN_ROLE` that does both should be sufficient.

Resolution:

- Rumpel: "keeping this potentially unnecessary flexibility for now since we have future plans for pTokens, and it's very important that the 'canonical' pToken addresses remain the same for future system additions, so it might make sense to ensure we won't need to alter the contracts to work with not-fully-known future designs."
- Darklinear: Acknowledged.

Systemic Risk

There are two major systemic risks to the protocol: its dependence on centralized logic and architecture, and the possibility that the issuers of points-earning tokens may blacklist the point tokenization vault.

Centralization

Broadly, there are two aspects to the centralization risk of the protocol:

1. Important on-chain functionality is restricted to trusted parties, who can access all funds in the vault
2. Important data and validation is provided by proprietary off-chain architecture

The risk inherent in the admin rights of trusted parties, assuming that the owners of the contracts themselves are not malicious, is that a malicious party can more easily gain access to restricted functionality by somehow recovering the private key belonging to an admin (or to an approved signer of a trusted multisig). Discussions around Issue L-5 have led the team to consider similar multi-factor mechanisms for other privileged functionality, such as `_authorizeUpgrade`.

The dependence on off-chain architecture is more difficult to address, but the team has plans to decentralize some aspects of this architecture in future versions of the protocol.

Blacklisting

Issue L-4 revealed a possible problem with the plan for dealing with blacklisting by a points-earning token issuer, and further development is required to define the protocol for the emergency disbursement of reward tokens in such a scenario. However, the main line of defense is forming explicit relationships with points-earning token issuers before adding their tokens to the protocol, and this is a viable solution for the short term.

But considering the potential impact on users, codifying the details of the blacklisting response should be considered a high priority.

Test Coverage

Prior to this review, the test suite consisted of unit tests covering 95% of the SLOC in `PointTokenVault.sol`.

In the process of the review, unit tests were written for the remaining 5% of the code. Additionally, the review utilized stateful fuzz tests for `deposit`, `withdraw`, `claimPTokens`, `redeemRewards`, and `convertRewardsToPTokens`.

These tests generate random values to serve as the parameters for each function, and the test handler repeatedly calls them in random sequences. It also randomly selects one of thirteen accounts — ten randomly generated and three with admin roles — to serve as the actor, and selects another to pass in as the `receiver`. It deploys ten points-earning tokens, ten pTokens (for ten points IDs), and ten reward tokens, and randomly selects from each of these lists wherever relevant throughout the tests.

The test handler locally tracks the state changes that should occur in the contract, which allows for the following invariants to be checked after each run:

- The locally tracked amount of each deposited points-earning token for every user always matches the values stored in the contract
- The locally tracked amount of pTokens claimed against each points-earning token for every user always matches the values stored in the contract
- The total supply of every pToken is always equal to the sum of all user balances

In addition to these invariants, every run asserts that its fuzzed function calls do what they are meant to do when they are supplied with valid inputs, that they revert when supplied with invalid inputs, and that they never revert unexpectedly.

After running this suite for thousands of iterations, this review can conclude with a high degree of confidence that the user-facing functionality and its relevant accounting is reliable and implemented as intended.

All tests written during the course of this review have been transferred to the Rumpel team.