Security Audit

April 2025

Rumpel: Passive Point Selling Contracts

Audited by
Ethan Bennett

DARKLINEAR SOLUTIONS

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 a protocol for points-earning tokens that allows users to mint pTokens against their accruing points, turning the points into liquid assets for use across other DeFi applications. The Passive Point Selling Contracts execute bundles of automated pToken claims on behalf of Rumpel wallets, then automatically swap the pTokens for USDC on Uniswap v3. This operation also charges a fee for the automated service and distributes the USDC to each user included the bundle. The contracts allow users to define their minimum acceptable price for each pToken in terms of USDC, and to set a receiver for the USDC.

This review was conducted on the Passive Point Selling Contracts repository at commit #172678e from April 21-24, 2025.

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 three 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. In the context of this review, Informational findings may also communicate important considerations for bundle construction and validation, which is outside the scope of the reviewed contracts but affects their security.
- **Gas Optimization** findings improve the implementation of the contracts strictly in terms of the gas expenditure required to use or deploy them.

Urgency according to risk classification

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

Trust Assumptions

- The architecture of the Passive Point Selling Contracts relies heavily on offchain services for safe and correct execution. The point sale bundles can only be executed by authorized operators, and most of the risk of unsafe or incorrect execution is in how these authorized operators construct and validate the bundles offchain. This process is largely outside the scope of this review, but some recommendations were made where appropriate, according to the Rumpel team's stated plans and how they affect the secure operation of the contracts in scope.
- The output token from the pToken swap is assumed to be USDC, but the contracts do not define this explicitly. This allows Rumpel to change their approach without requiring updates to the contract, but creates some known issues (e.g., the risk that the decimals in a user's defined `minPrice` for a given pToken could be incorrect in terms of the output token). The Rumpel team plans to mitigate these issues by communicating with their users ahead of any changes to the output token.
- Most or all Uniswap v3 pools used in the execution of point sales are expected to have low liquidity under normal conditions. The Rumpel team will coordinate with market makers to provide liquidity for the bundled point sales immediately before executing the bundles.

Summary of Findings

Classification	Finding
Low risk (L-1)	Users can force batch failure by updating `minPrice`
Low risk (L-2)	Spot price usage in batch `minPrice` could enable DOS or MEV
Low risk (L-3)	`rumpelWallet` check can be bypassed by arbitrary contracts
Informational (I-1)	Claim for incorrect `pointsId` included in batch throws unhandled error
Informational (I-2)	Liquidity provided by market makers just in time to enable point sale could be depleted by frontrunners
Informational (I-3)	Dust in contract cannot be recovered
Informational (I-4)	Receivers blacklisted by USDC will revert batch
Gas Optimization (G-1)	Cache storage location for `userPreferences[rumpelWallet]`

Findings

L-1: Users can force batch failure by updating `minPrice`

Low Severity / PointSellingController.sol#L166

Description: Users can frontrun the admin's `executePointSale` call by updating their `minPrice` to be higher than the batch `minPrice` parameter, causing the entire transaction to revert at the `userPreferences` check:

```
require(
    minPrice >= userPreferences[wallet].minPrices[pToken],
    MinPriceTooLow()
);
```

This check verifies that each user's minimum price is met by the batch minimum price. If any user's minimum price is higher, the whole transaction fails with `MinPriceTooLow()`.

This DOS vector is similar to the approval revocation scenario already noted in the comments:

```
/// @dev It is possible for a user to revoke pToken approvals right
///      before a batch.
///      It is the operator's responsibility to ensure that this is
///      dealt with appropriately.
```

Recommendation: Although the operator can respond to this situation like it does for the approval revocation DOS, the contract can avoid the operational griefing entirely by setting `amountToClaim` to 0 for users whose price requirements can't be met rather than reverting the entire batch:

```
for (uint256 i = 0; i < numWallets; i++) {
    address wallet = wallets[i];

    // Skip users with incompatible price requirements instead of reverting
    if (minPrice < userPreferences[wallet].minPrices[pToken]) {
        claims[i].amountToClaim = 0;
        continue;
    }

    calls[i] = abi.encodeCall(
        pointTokenizationVault.claimPTokens,
        (claims[i], wallet, address(this))
    );

    unchecked {
        totalPTokens += claims[i].amountToClaim;
    }
}
```

This approach prevents a single user from blocking the entire batch execution.

Note: although a claim removed by the above means would not result in any funds being sent to its associated user, it would be best to also skip such claims in the loop beginning on L194:

```
// Transfer tokenOut to users.
for (uint256 i = 0; i < numWallets; i++) {
    if (claims[i].amountToClaim == 0) continue;
```

Resolution:

- Rumpel: Will think on this a bit more. Right now leaning towards having the policy on our backend that, if somebody is doing this, we remove them from the batch (perhaps even for future batches too)
- Darklinear: That makes sense, at least for known attackers. One other factor worth considering is that this could also happen by coincidence if a user attempts to update their preferences at the same time as the admin calls `executePointSale` — probably not super likely to occur and fairly low impact if it does, but it could be difficult for the backend/admin to distinguish between DOS attempts and honest mistakes.
- Rumpel: Acknowledged

L-2: Spot price usage in batch `minPrice` could enable DOS or MEV

Low Severity / UniswapV3PointSelling.sol#L47

Description: Although the offchain service responsible for determining the `minPrice` is out-of-scope, the `minPrice` was defined by the team as a function of the pool's spot price. Using spot price as the primary source for the `minPrice` parameter in `executePointSale` exposes the system to two related vulnerabilities:

- **Denial-of-service through price manipulation:** Attackers can drive pool prices below the batch's `minPrice` threshold, causing transactions to fail at the swap execution.

  Since the `minPrice` enforces a lower bound on acceptable swap rates, an attacker can temporarily depress the pool price before the batch transaction is confirmed, causing the entire batch to revert when the `amountOutMinimum` requirement cannot be met.

- **MEV via sandwich attacks:** This risk is partially mitigated by the thin pToken markets and just-in-time liquidity provision from coordinated market makers, but remains theoretically possible with sufficient capital.

Recommendation: When constructing calls to `executePointSale`, replace the spot price with a TWAP-based approach to the determination of `minPrice`.

Resolution:

- Rumpel: Will do. Added to our backend notes

L-3: `rumpelWallet` check can be bypassed by arbitrary contracts

Low Severity / PointSellingController.sol#L94

Description: The `setUserPreferences` function in `PointSellingController` prevents externally-owned accounts from setting user preferences, but it fails to distinguish between genuine Rumpel wallets and arbitrary contracts with a matching interface.

The impact is limited to unnecessary user settings being stored in the contract. However, the existing check intends to prevent exactly this outcome, and it is not sufficient to do so effectively.

Recommendation: Check on the `PointTokenVault` that the `rumpelWallet` has added the `PointSaleController` to its `trustedReceivers`. While not all Rumpel wallets will have done this, any wallet that can be included in a point sale batch would need to have taken this step.

Resolution:

- Rumpel: Unfortunately the code to verify that an address is a Rumpel Wallet (via the Rumpel Wallet Factory) is quite involved, so I think our decision here is to do the checking on our backend. Similar with the `trustedReceivers` allowance. Added both to our backend checklist

---

I-1: Claim for incorrect `pointsId` included in batch throws unhandled error

Informational / PointSellingController.sol#L144

Description: The `executePointSale` function lacks validation for ensuring that claims correspond to the specified pToken. When an admin includes a claim with a different `pointsId` than intended, the function will process it alongside valid claims without error.

This mismatch only becomes apparent during the swap operation, which ultimately fails due to the insufficient pToken balance. But the real cause of this failure will not be indicated by the error thrown in the swap attempt.

Recommendation: Modify the function signature to accept the expected `pointsId` parameter and validate each claim against it:

```
function executePointSale(
    ERC20 pToken,
    bytes32 expectedPointsId,
    ERC20 tokenOut,
    address[] calldata wallets,
    IPointTokenizationVault pointTokenizationVault,
    Claim[] calldata claims,
    uint256 minPrice,
    bytes calldata additionalParams
) external onlyOwner {
    // ...
    for (uint256 i = 0; i < numWallets; i++) {
        // Validate pointsId
        require(
            claims[i].pointsId == expectedPointsId,
            InvalidClaimForBatch()
        );
    // ...
}
```

Alternatively, the function could fetch the `pointsId` from the `PointTokenVault` for the provided pToken and compare that with the `pointsId` defined in each claim. This would be more effective at preventing admin errors, but would need to be fetched on every iteration of the loop in `executePointSale`.

Although it is reasonable to assume the admin will not make mistakes, it is good practice to throw custom errors for each reverting branch.

Resolution:

- Rumpel: Fixed in commit #d2c91bf
- Darklinear: Confirmed

I-2: Liquidity provided by market makers just in time to enable point sale could be depleted by frontrunners

Informational / UniswapV3PointSelling.sol#L47

Description: The `PointSellingController` relies on market makers providing just-in-time liquidity for batch token sales. Attackers monitoring the mempool can frontrun the `executePointSale` transaction, depleting the provisioned liquidity and causing the batch transaction to fail.

Unlike price manipulation attacks, this vulnerability exists regardless of how the `minPrice` is calculated, since it targets available liquidity rather than the price.

Recommendation: Coordinate with market makers to provide excess liquidity beyond the minimum needed to execute the swap.

Resolution: Acknowledged

I-3: Dust in contract cannot be recovered

Informational / PointSellingController.sol#L209

Description: The `PointSellingController` explicitly acknowledges that dust will remain after distribution due to rounding during wallet share calculations, but the contract lacks any mechanism to recover these tokens. It may be worth including such a function, although it would take a lot of time for the dust to accumulate to a substantial amount. This could become more of a concern if future versions of this contract ever reference the contract's USDC balance directly.

Recommendation: Consider implementing a dust recovery function that allows the contract owner to sweep accumulated tokens:

```
function sweepDust(ERC20 token, address recipient) external onlyOwner {
    uint256 balance = token.balanceOf(address(this));
    require(balance > 0, "No dust to sweep");
    token.safeTransfer(recipient, balance);
    emit DustSwept(token, recipient, balance);
}

event DustSwept(ERC20 indexed token, address indexed recipient, uint256 amount);
```

Resolution: Acknowledged

I-4: Receivers blacklisted by USDC will revert batch

Informational / PointSellingController.sol#L210

Description: If any recipient in an `executePointSale` batch is blacklisted by the output token (USDC), the entire transaction will revert when the controller attempts to transfer to that recipient.

Recommendation: The admin should verify recipient addresses against known token blacklists before including them in batches.

Resolution:

- Rumpel: Added to our backend checklist

G-1: Cache storage location for `userPreferences[rumpelWallet]`

Gas Optimization / PointSellingController.sol#L108

Description: The `setUserPreferences` function in `PointSellingController.sol` accesses the same storage mapping twice, which creates a minor gas inefficiency:

```
userPreferences[rumpelWallet].recipient = recipient;
for (uint256 i = 0; i < pTokens.length; i++) {
    userPreferences[rumpelWallet].minPrices[pTokens[i]] = minPrices[i];
}
```

Each access to `userPreferences[rumpelWallet]` requires the EVM to compute the storage location of the mapped value, so caching a reference to this location prevents redundant computation. And since one of these reads is inside a loop, the benefit of this optimization scales with the length of `pTokens`.

Gas measurement tests with `optimizer_runs` set to `10_000` (mimicking production) show the following modest savings:

Metric	Before	After	Savings
Deployment cost	1,573,025	1,570,865	2,160
Function avg. cost	43,192	43,172	20
Function max cost	75,697	75,636	61
Function snapshot	51,740	51,679	61

The savings are minor in typical usage scenarios, but could accumulate if users set preferences for many pTokens in a single transaction.

Recommendation: Cache the storage pointer to reduce redundant lookups:

```
UserPreferences storage preferences = userPreferences[rumpelWallet];

preferences.recipient = recipient;
for (uint256 i = 0; i < pTokens.length; i++) {
    preferences.minPrices[pTokens[i]] = minPrices[i];
}
```

Resolution:

- Rumpel: Fixed in commit #db348ac
- Darklinear: Confirmed