Rumpel Wallet and Point Tokenization Vault Update

Security Review

Ethan Bennett Darklinear Solutions

Contents

About Darklinear Solutions

Darklinear Solutions leverages years of expertise in smart contract and software engineering to provide unrivaled security reviews for blockchain applications. Learn more at darklinear.com.

Introduction

Rumpel is an on-chain vault 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 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 in the Rumpel protocol. Additionally, Rumpel has implemented a custom Safe wallet implementation to enable this functionality for its users.

This review was conducted on the updates to point-tokenization-vault at commit #50883d7 and the Rumpel wallet (at commit #67944bd) from July 22-25, 2024.

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:

- **Medium risk** findings represent issues that could significantly impact the functionality of the protocol, but may not be easily exploitable.
- **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
Medium risk (M-1)	`RumpelGuard` cannot prevent users from directly redeeming points from external protocols in some cases
Medium risk (M-2)	Rumpel admin will not be able to redeem rewards from external protocols on behalf of Rumpel wallets in some cases
Medium risk (M-3)	PToken `pause` functionality can be bypassed
Low risk (L-1)	External protocol team can evade `RumpelGuard` restrictions by forcing Rumpel wallet to call their contract's fallback function
Low risk (L-2)	`RumpelGuard` and `RumpelModule` are not immutable
Low risk (L-3)	`deposit` should not be allowed

Findings

M-1: `RumpelGuard` cannot prevent users from directly redeeming points from external protocols in some cases

Severity: Medium risk

Context: RumpelGuard.sol#L48

Description: In order for the Rumpel wallet to be flexible enough to interact with a wide range of external points-issuing protocols, it allows users to `delegatecall` to `SignMessageLib.signMessage` with an arbitrary message. This message is then marked as "signed" in the Safe's storage, and anyone can validate it by providing that same hashed message as an argument to the `isValidSignature` function.

This mechanism does not specify or implement any functionality related to the conveyance of this signed message to external protocols, and in fact, signatures like these are often sent to external protocols by way of HTTP requests. This is typically the case for logging in with an EOA, but more significantly, it was confirmed to be the likely design for some points-earning protocols' redemption mechanisms, which Rumpel will need to prevent users from accessing directly.

This is how message signing and verification would look with those protocols:

1. The protocol prompts the user to sign two messages: one to confirm ownership of the account, and another to agree to the terms of service. Let's say those messages are the simple strings "I own this account" and "I agree to the terms"
2. The user sends two `delegatecall` transactions to the `SignMessageLib` contract via the Safe's `execTransaction` function: one with the message `"I own this account"` and the other with `"I agree to the terms"`.
3. The Safe checks with `RumpelGuard` before executing the transaction. `checkTransaction` allows the `delegatecall` to `SignMessageLib`, so the transaction proceeds
4. `SignMessageLib` executes its `signMessage` function in the context of the Safe. Unlike the process for EOA (externally-owned account) message signatures, this function simply hashes the user's input and then stores it in a mapping of hashed messages to integers (with the value `1`)

5. After calling `signMessage` with both `"I own this account"` and `"I agree to the terms"`, the user receives the hashes corresponding with each
6. The user now sends each of these hashes back to the external protocol by way of its HTTP API, outside the purview of the `RumpelGuard`
7. The external protocol validates the signature. In this scenario, let's assume they use the arguments `(providedHash, emptyString)` in order to intentionally hit the `signedMessages` mapping (the other possibility is covered in Finding M-2)
8. `isValidSignature` checks its mapping of signed messages for whether the value associated with the provided hashed message is `0`. The values for both of the user's messages were set to `1` when they called `signMessage`, so the function returns the ERC-1271-specified magic value for each to certify to the protocol that the signatures are valid
9. With the authentication handshake complete, the external protocol will now send its reward tokens directly to the Rumpel wallet without requiring further action from the user

The impact of this vulnerability depends on the particular reward token. If, for instance, it was USDC — the prototypical example mentioned internally of a token that will always be transferrable by the user — the `RumpelGuard` could not prevent the user from transferring their rewards out of their Rumpel wallet. This user could then redeem their pTokens for more reward tokens in the Point Tokenization Vault, which would double their earnings at the expense of other Rumpel users and render the protocol insolvent.

If, however, the reward token is unique to the points-issuing protocol, its `transfer` functionality would need to be explicitly enabled by Rumpel admin in the `RumpelGuard` in order for the user to profit from this exploit. If these tokens become stuck in the Rumpel wallet, admin can easily transfer them out with the `RumpelModule`. But crucially, this would not be the case if the external protocol allows the user to define a secondary account to receive their reward tokens — a common enough design pattern to warrant considering the overall impact to be high-severity in more cases than not.

Recommendation: This is a difficult problem, because there is no established standard for the exact data that must be signed in any given context. As a result, it would not be easy to effectively modulate the user's permission to sign messages with their Rumpel wallet. And since signing messages is essential for logging into a great number of external protocols, fully blocking this functionality is also not a viable option.

Although it may be out of scope for the immediate response to this review, the most robust and ideal solution would likely leverage cryptoeconomic incentives to make it possible, but not beneficial, for a user to recover their rewards themselves. It could use similar mechanisms to handle the "bad debt" from unbacked pTokens when a user redeems their points directly from external protocols after minting pTokens, or the Point Tokenization Vault could enforce that all pToken redemptions are "Merkle-based" (ensuring that all pTokens are backed by points before allowing their redemption).

In lieu of such large changes to the protocol, the Rumpel wallet could implement its own version of `SignMessageLib` like the following:

```
// pseudocode: do not copy/paste directly
function signMessage(bytes calldata _data) external {
    bytes32 msgHash = getMessageHash(_data);
    require(_isMessageAllowed(_data), "Message not allowed");
    signedMessages[msgHash] = 1;
    emit SignMsg(msgHash);
}
```

```
function signMessageAdmin(bytes calldata _data) external {
    bytes32 msgHash = getMessageHash(_data);
    signedMessages[msgHash] = 1;
    emit SignMsg(msgHash);
}

function _isMessageAllowed(bytes calldata _data) internal view returns (bool) {
    for (uint256 i = 0; i < signatureDataAllowList.length; i++) {
        if (signatureMessageAllowList[i] == _data) return true;
    }
    return false;
}
```

This solution would implement an allow list for messages that can be signed by the user, which could be maintained by the admin and restricted such that it could only be updated through the `RumpelModule`. `signMessageAdmin` could be called by `RumpelModule` to allow admins the ability to sign messages without restrictions.

As mentioned above, however, the unpredictability of the required messages complicates this approach in the long term.

Resolution:

- Rumpel: This and M-2 are tricky. For this issue, we're actually slightly less concerned than might be expected (though still concerned) for two reasons:

    1. We plan to "permanently allow" very sparingly, and accept that any token we do that for will never be a reward token redeemable by pTokens

    2. If the token is allowed to transfer prior to release, we expect we'll need to disable it prior to distribution if there's any risk of something like this. We already have a situation where we expect to do this with ENA, where we need to enable transfers for it because users will want to add it to their wallet for the points boost. But because it's the reward token as well, we plan to freeze it a day before reward token release, claim, sweep, then unfreeze. In that case, even if the user is able to claim somehow, we can still sweep what's needed

  That said, still the ideal situation is that we're working with the protocol teams, and we have a nice way to claim rewards for users, but, as you mention, a lot depends on these external protocols and we can't always rely on that.

  There are still the unhappy paths for us

    1. Where we're caught unaware by a reward token release event, and the token is enabled for transfer, and the users can claim with their signature
    2. if the external protocol allows the user to define a secondary account to receive their reward tokens (this was a good point)

  So we're still thinking about those. For now, we plan to make the change mentioned in the other signature issue, while we keep thinking about this

- Darklinear: Acknowledged.

M-2: Rumpel admin will not be able to redeem rewards from external protocols on behalf of Rumpel wallets in some cases

Severity: Medium risk

Context: RumpelModule.sol#L45

Description: If an external protocol attempts to verify a Safe signature by normal means (i.e., by passing a signature into the verification function), Rumpel admin will not be able to provide a valid signature for Rumpel wallets. The Rumpel wallet will treat the signature as a concatenated series of owners' signatures, and it will check that each signer is actually an owner of the Safe. The Rumpel admin is not an owner, so their signature will not be accepted.

Despite lacking a path for exploitation, this vulnerability represents a substantial issue in the event that a points-earning protocol requires a user signature to redeem points. If the protocol cannot redeem points on behalf of Rumpel wallets, the associated pTokens will be unbacked, and may cause the protocl to become insolvent.

Recommendation: Rumpel should create its own version of the compatibilityFallbackHandler, which retains all the functionality of the original, but implements `isValidSignature` in such a way that it will only reference the Safe's `signedMessages` mapping when validating signatures. This way, any signature registered by way of the `signMessage` function — which is available to Rumpel admin via the Rumpel module — will return the valid ERC-1271 magic value and be treated as valid by external protocols.

Additionally, it would be best to make this functionality upgradeable, as there is some degree of uncertainty around how external protocols will implement their points redemption mechanisms.

The following pseudocode is an example of how this could be implemented:

```
contract SignatureValidationBeacon is Ownable {
    address public signatureValidationContract;

    constructor(address validationImplementation) {
        owner = msg.sender;
        signatureValidationContract = validationImplementation;
    }

    function updateImplementation(address newImplementation) public onlyOwner {
        signatureValidationContract = newImplementation;
    }
}

contract SignatureValidationImplementation {
    function isValidSignature(
        bytes memory _messageData,
        bytes32 _messageHash,
        bytes memory _signature,
        Safe _safe
    ) public view returns (bool) {
        // Always check storage for signed messages; never checkSignatures
        require(safe.signedMessages(messageHash) != 0, "Hash not approved");
        return true;
    }
}

contract CompatibilityFallbackHandler {
    // ...

    // Note: it would also be best to avoid hard-coding this address
    SignatureValidationBeacon validationBeacon = 0x123;

    // Non-standard signature that matches expected Safe v1.4.1 ABI
    function isValidSignature(
        bytes memory _data,
        bytes memory _signature
    ) public view override returns (bytes4) {
        // Caller should be a Safe
        Safe safe = Safe(payable(msg.sender));
        bytes memory messageData = encodeMessageDataForSafe(safe, _data);
        bytes32 messageHash = keccak256(messageData);

        address implementation = validationBeacon.signatureValidationContract();

        require(
            SignatureValidationImplementation(implementation).isValidSignature(
                messageData,
                messageHash,
                _signature,
                safe
            )
        );

        return EIP1271_MAGIC_VALUE;
    }

    function isValidSignature(
        bytes32 _dataHash,
        bytes memory _signature
    ) public view returns (bytes4) {
        // Caller should be a Safe
        Safe safe = Safe(payable(msg.sender));
        bytes memory messageData = encodeMessageDataForSafe(
            safe,
            abi.encode(_dataHash)
        );
        bytes32 messageHash = keccak256(messageData);

        address implementation = validationBeacon.signatureValidationContract();
        require(
            SignatureValidationImplementation(implementation).isValidSignature(
                messageData,
                messageHash,
                _signature,
                safe
            )
        );

        return UPDATED_MAGIC_VALUE;
    }

    // ...
}
```

Resolution:

- Rumpel: Fixed in commit #e9f5c96.
- Darklinear: Confirmed.

M-3: PToken `pause` functionality can be bypassed

Severity: Medium risk

Context: PToken.sol#L21, PToken.sol#L25

Description: `PToken.sol` is `Pausable`. When a pToken is paused, attempts to transfer the pToken will revert. However, minting and burning pTokens is possible even when a pToken's contract is in a paused state, which allows users to indirectly transfer pTokens through the Point Tokenization Vault (as long as redemptions have been enabled for the pToken).

First, the user would call `redeemRewards` with their pTokens, which will burn their pTokens in exchange for reward tokens. Then, with the reward tokens they have just received, they will call `convertRewardsToPTokens`. This function allows the user to specify a `receiver` address, which can be defined as the intended recipient of the transfer. When executed, it will mint new pTokens directly to the recipient, completing the transfer by way of an indirect sequence of burning and minting.

Recommendation: The easiest way to prevent this would be to add the `whenNotPaused` modifier to `mint` and `burn`.

If it is important to allow this functionality in the paused state, Rumpel could implement a cooldown period after each action a user takes with a particular pToken (when the pToken is paused). This would preserve a user's ability to redeem their pTokens as intended, while preventing them from exploiting the process to bypass paused transfers.

Resolution:

- Rumpel: Fixed in commit #36e40f2.
- Darklinear: Confirmed.

L-1: External protocol team can evade `RumpelGuard` restrictions by forcing Rumpel wallet to call their contract's fallback function

Severity: Low risk

Context: RumpelGuard.sol#L46

Description: The Safe's `execTransaction` function works by constructing a low-level `call` to a specified address with some user-provided data. This data can include a function signature, which the `RumpelGuard` uses (along with the destination address) to check whether a user's action should be allowed by the Rumpel wallet. Function signatures are four bytes, so `RumpelGuard.checkTransaction` attempts to isolate the signature from the rest of the `data` by casting it to type `bytes4`.

If `data` is smaller than `bytes4`, however, casting it to `bytes4` will pad the end of the user-provided data with extra zeroes. This creates the possibility that the function signature checked by the `RumpelGuard` differs from the one ultimately used to execute the transaction in the Safe.

In fact, this problem could arise without malicious intent, and it is exploitable by anyone. But without a malicious external protocol designing their contract with the attack in mind, it is unlikely to actually occur. If a protocol that Rumpel integrates with sets out to exploit them, however, it could be accomplished by these means.

Take, for example, the function `settle(uint256)`: its function selector is `0x8df82800`. Let's say this function is approved in the `RumpelGuard` for some external contract `0x1`. The developers of `0x1` have subtly added a fallback function to their contract, which allows users to redeem their points for reward tokens. They now pass the malformed function selector `0x8df828` into their Rumpel wallet's `execTransaction` function.

When the `RumpelGuard` checks this transaction, it casts the malformed selector to `bytes4`. This fills the rest of the local variable's space with zeroes, mutating the user's input to `0x8df82800`. This mutated function selector is allowed by the `RumpelGuard`, so the checks will pass. But the Safe goes on to execute the low-level call with the original malformed one. Since no function exists corresponding to this shortened selector, the contract's fallback function will execute instead, which will contain the attacker's malicious code.

Recommendation: `RumpelGuard.checkTransaction` should check that the length of `data` is at least four bytes, and revert if it is not. In some cases, it may also be important to allow `data` of length `0`.

Resolution:

- Rumpel: Fixed in commit #65ed648.
- Darklinear: Confirmed.

L-2: `RumpelGuard` and `RumpelModule` are not immutable

Severity: Low risk

Context: RumpelGuard.sol, RumpelModule.sol

Description: The `RumpelGuard` and `RumpelModule` are never meant to be updated or replaced, and some restrictions exist in the code to block either from happening. These restrictions do not actually succeed in permanently blocking this functionality, but it is also not possible for either the user or Rumpel admins to accomplish this by default.

If, however, Rumpel admin were to enable the Safe functionality relating to enabling and disabling modules or guards, an attacker could trivially undermine many of the security assumptions made by the protocol. Although there would be no reason for Rumpel admin to do this, it could be done unintentionally, it could be done by a compromised admin account, or future versions of the protocol could create other vulnerabilities that could give malicious actors access to these functions.

Recommendation: In the same way that the `RumpelModule` permanently blocks some functionality from being executed through the module, the `RumpelGuard` should permanently block `enableModule`, `disableModule`, and `setGuard`. This would make it impossible to update either by any means: any future upgrades to the Rumpel wallet would need to occur by way of migrations to entirely new instances.

Additionally, `disableModule` and `setGuard` should be permanently blocked for the Rumpel admin (along with `enableModule`, which is already blocked) on RumpelWalletFactory.s.sol#L41.

Resolution:

- Rumpel: Fixed in commit #ae6b41a.
- Darklinear: Confirmed.

L-3: `deposit` should not be allowed

Severity: Low risk

Context: PointTokenVault.sol#L114

Description: Since points-earning tokens are no longer intended to be deposited directly into the Point Tokenization Vault, this functionality should not remain accessible by users (even though it will also be blocked in the UI). There is no way to exploit this currently, but accepting arbitrary amounts of users' tokens into a deprecated function is nonetheless an unnecessary risk that can easily be avoided.

Recommendation: If the priority is to retain this functionality for possible use in the future, the easiest solution is to ensure that the `cap` for each points earning token always remains at `0`. This is not at all explicit, however, and it may be worth adding a pausable modifier to `deposit` to make it less likely to become accidentally enabled in the future.

Resolution:

- Rumpel: Noted. I think for our purposes, the 0 cap should work for this release (especially since it defaults to 0). But we will make sure internal documentation is clear on this
- Darklinear: Confirmed.