BLOCKSEC

Security Audit
Report for Prosper Contracts

Date: September 8, 2026 **Version:** 2.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Pharos Labs
Target	Prosper Contracts

Version History

Version	Date	Description
1.0	Aug 3, 2026	First release
2.0	Sep 8, 2026	Second release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-09-10 08:42:55+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of Prosper Contracts of Pharos Labs.

Prosper Contract is an on-chain launch protocol designed to integrate external R25 Vaults with dedicated pToken campaigns. For each integrated Vault, the protocol creates and manages an independent campaign that begins with an internal launch and continues through liquidity migration to an external market. After the migration, the protocol supports reward claims and uses collected channel fees to conduct pToken buybacks. This structure provides each R25 Vault with an isolated campaign lifecycle while using a shared framework for token launches, liquidity migration, reward distribution, and fee-funded buybacks. Compared to Version 1 (68c547), Version 2 (b8ef7f) introduces asynchronous R25 deposit requests with user cancellation, canonical pair protection, and time-locked LP positions instead of permanent LP burning. It also extends failed-campaign handling by burning reserved tokens and converting accrued fees into R25 Vault Shares that users can reclaim through a Merkle-based mechanism without an expiration period or administrative sweep.

Note this audit focuses on the codes located in the following directories/files:

- src

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (Version 1), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Prosper Contracts	Version 1	68c54736e16ebafcfe93c84c97b536125c4b4f2c
	Version 2	b8ef7fae5d907c5e2bc2a723ef55d5451e7638ab
	Version 3	5e5f07a8e6e982f786d09b7bd193a20c250d08e3

[1] https://github.com/pharos-labs-dev/prosper-contracts

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness

- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **five** recommendations and **four** notes.

- Medium Risk: 1
- Low Risk: 2
- Recommendation: 5
- Note: 4

ID	Severity	Description	Category	Status
1	Medium	Potential DoS of migration due to single sided V2 pair reserves	Security Issue	Fixed
2	Low	Lack of `shareToken` validation against `R25Vault` in function `createFromR25Vault()`	Security Issue	Fixed
3	Low	Transferring default admin role to existing business accounts circumvents role separation policy	Security Issue	Fixed
4	-	Handle permit failures caused by front-running	Recommendation	Fixed
5	-	Add initializer-disabling constructors to upgradeable implementation contracts	Recommendation	Fixed
6	-	Validate and enforce consistency between buyback router and migration venue	Recommendation	Fixed
7	-	Align the ERC-7201 annotation with runtime storage	Recommendation	Fixed
8	-	Allow failure finalization after reward-pool burning	Recommendation	Fixed
9	-	Potential centralization risks	Note	-
10	-	Reliance on trusted off-chain logic	Note	-
11	-	Reliance on timely migration execution	Note	-
12	-	Trust assumptions on the external v2 factory	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential DoS of migration due to single sided V2 pair reserves

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `MigrationEscrowUpgradeable`, function `addLiquidity()` migrates the reserves of an internal bonding curve market to an external V2 pair after the corresponding

`BondingCurveMarketUpgradeable` enters the `GraduatedPendingMigration` phase. If the target pair already exists, the migration can proceed only when `allowExistingPair` is set to true. However, function `addLiquidity()` does not validate the pair's reserve state. An attacker can create the target pair before the migration, transfer only one of the two tokens to it, and invoke function `sync()`. This leaves one stored reserve nonzero while the other remains zero. When the migrator subsequently invokes function `addLiquidity()` with `allowExistingPair` enabled, the Router attempts to calculate the optimal deposit amounts from the existing reserve ratio. The calculation reverts because one of the reserves is zero. As a result, the intended recovery path cannot add liquidity to the pair, and repeated migration attempts remain blocked until the pair state is repaired through a separate process.

```
125   function addLiquidity(
126       address vault,
127       bool allowExistingPair,
128       uint256 pTokenSwapIn,
129       uint256 wphrsSwapIn,
130       uint256 minSwapOut,
131       uint256 minPTokenAdded,
132       uint256 minWphrsAdded,
133       uint256 deadline
134   ) external onlyRole(MIGRATOR_ROLE) nonReentrant returns (address pair, uint256 liquidity) {
135       if (paused) revert Paused();
136       if (block.timestamp > deadline) revert DeadlineExpired();
137       if (pTokenSwapIn != 0 && wphrsSwapIn != 0) revert InvalidAmount();
138       if (minPTokenAdded == 0 || minWphrsAdded == 0) revert InvalidAmount();
139       Campaign storage item = _campaigns[vault];
140       if (item.pToken == address(0) || item.finalized || item.failed) revert InvalidState();
141       if (IMigrationMarket(item.market).phase() != IBondingCurveMarket.Phase.
              GraduatedPendingMigration) {
142           revert InvalidState();
143       }
144
145       pair = IUniV2FactoryView(v2Factory).getPair(item.pToken, wphrs);
146       if (!allowExistingPair && pair != address(0)) revert ExistingPair();
147       _releaseReserve(vault, item);
148       if (pTokenSwapIn != 0) _swap(item, item.pToken, wphrs, pTokenSwapIn, minSwapOut, deadline);
149       if (wphrsSwapIn != 0) _swap(item, wphrs, item.pToken, wphrsSwapIn, minSwapOut, deadline);
150
151       uint256 pDesired = item.pTokenBalance;
152       uint256 wDesired = item.wphrsBalance;
153       if (pDesired == 0 || wDesired == 0) revert InvalidAmount();
154       IERC20(item.pToken).forceApprove(address(router), pDesired);
155       IERC20(wphrs).forceApprove(address(router), wDesired);
156       (uint256 pAdded, uint256 wAdded, uint256 minted) = router.addLiquidity(
157           item.pToken, wphrs, pDesired, wDesired, minPTokenAdded, minWphrsAdded, lpBurnAddress,
                deadline
158       );
159       IERC20(item.pToken).forceApprove(address(router), 0);
160       IERC20(wphrs).forceApprove(address(router), 0);
161       if (minted == 0 || pAdded > item.pTokenBalance || wAdded > item.wphrsBalance) revert
              InvalidAmount();
```

```
162         item.pTokenBalance -= pAdded;
163         item.wphrsBalance -= wAdded;
164         pair = IUniV2FactoryView(v2Factory).getPair(item.pToken, wphrs);
165         if (pair == address(0)) revert InvalidState();
166         item.externalPair = pair;
167         liquidity = minted;
168         emit LiquidityAdded(vault, pair, pAdded, wAdded, minted);
169     }
```

Listing 2.1: src/migration/MigrationEscrowUpgradeable.sol

Impact Migration from the inner market to the external V2 pair may be disrupted by a DoS condition, leaving the market stuck in `GraduatedPendingMigration` and preventing normal outer-market activation.

Suggestion Revise the logic accordingly.

2.1.2 Lack of `shareToken` validation against `R25Vault` in function `createFromR25Vault()`

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description According to the protocol design, `Prosper` is expected to verify canonical fields on-chain during deployment. When creating a new `pToken` via function `ProsperFactoryUpgradeable.createFromR25Vault()`, it accepts a user-provided `shareToken` and forwards it to the downstream contract `FeeFallbackDistributorUpgradeable` as the fallback claim asset for the specified `R25Vault`.

However, function `createFromR25Vault()` only performs a basic non-zero address check on `shareToken` without validating it against the canonical address returned by function `IR25Vault(r25Vault).asset()`. As a result, any non-zero `shareToken` provided by the caller is accepted and registered as the canonical fallback asset for subsequent protocol operations.

```
113     function createFromR25Vault(
114         address r25Vault,
115         address shareToken,
116         address curatorRecipient,
117         string calldata pVaultName,
118         string calldata pVaultSymbol,
119         uint256 rewardCap,
120         uint256 navBase
121     )
122         external
123         onlyProsperRole(IProsperAccessController(accessController()).CREATOR_ROLE())
124         whenProsperModuleActive(keccak256("MODULE_VAULT_CREATION"))
125         returns (Deployment memory dep)
126     {
127         // Idempotency lets the backend safely retry an observed R25 creation event.
128         dep = _deployments[r25Vault];
129         if (dep.pToken != address(0)) return dep;
```

```
130         if (
131             r25Vault == address(0) || shareToken == address(0) || curatorRecipient == address(0) ||
                    rewardCap == 0
132                 || navBase == 0
133         ) {
134             revert ZeroAddress();
135         }
136         if (
137             bytes(pVaultName).length == 0 || bytes(pVaultName).length > 120 || bytes(pVaultSymbol).
                    length == 0
138                 || bytes(pVaultSymbol).length > 32
139         ) revert InvalidConfig();
140
141         ProsperGlobalConfigUpgradeable.LaunchConfig memory cfg = globalConfig.getLaunchConfig();
142         ProsperGlobalConfigUpgradeable.ProtocolAddresses memory addresses_ = globalConfig.
                getProtocolAddresses();
143         TokenAllocation memory allocation = _calculateAllocation(cfg);
144         IMigrationRegistrar migration = IMigrationRegistrar(addresses_.migrationEscrow);
145         if (migration.wrappedNative() != addresses_.wrappedNative) revert InvalidConfig();
146         address token = _deployPToken(pVaultName, pVaultSymbol, addresses_.migrationEscrow,
                migration);
147         address market =
148             _deployMarket(r25Vault, curatorRecipient, token, allocation.inner, allocation.outer,
                    cfg, addresses_);
149         address buyback = _deployBuyback(token, market, addresses_);
150
151         bytes32 campaignId = keccak256(abi.encode(block.chainid, address(this), r25Vault, token));
152         // Commit the idempotency marker before interacting with the registrars. Any downstream
153         // failure reverts this write with the rest of the transaction.
154         dep = Deployment({
155             campaignId: campaignId,
156             pToken: token,
157             market: market,
158             buyback: buyback,
159             shareToken: shareToken,
160             rewardCap: rewardCap,
161             navBase: navBase
162         });
163         _deployments[r25Vault] = dep;
164
165         _registerAndFundCampaign(
166             r25Vault, curatorRecipient, dep, allocation, cfg, addresses_.migrationEscrow, migration
167         );
168
169         emit ProsperVaultLinked(r25Vault, campaignId, token, market, buyback);
170     }
```

Listing 2.2: src/factory/ProsperFactoryUpgradeable.sol

Impact If a caller with the `CREATOR_ROLE` mistakenly supplies an incorrect `shareToken` address, the deployment will still succeed. This misconfiguration causes subsequent fallback distributions in failed campaigns to operate on an invalid asset.

Suggestion Revise the logic to ensure that the on-chain verification validates that the `shareToken` matches the canonical share token retrieved from `R25Vault`.

2.1.3 Transferring default admin role to existing business accounts circumvents role separation policy

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description The contract `ProsperAccessControllerUpgradeable` is designed to enforce strict account separation between operational business roles and the default admin (i.e., `OWNER_ROLE`). The function `_requireUnique()` ensures that business roles (i.e., `CREATOR_ROLE`, `FEE_OPERATOR_ROLE`, `MIGRATOR_ROLE`, and `ROOT_SETTER_ROLE`) cannot be assigned to an address holding the `DEFAULT_ADMIN_ROLE`, while disabling direct updates through standard OpenZeppelin functions such as function `grantRole()`, function `revokeRole()`, and function `renounceRole()`.

However, the inherited `OpenZeppelin` default admin transfer flow does not incorporate this check. Consequently, the current owner can initiate a default admin transfer to an address that already holds an active business role, effectively circumventing the account separation policy. Alternatively, a business role can be assigned to an address that is currently a pending default admin. Once the target account invokes the function `acceptDefaultAdminTransfer()`, the same address will hold both top-level owner authority and operational business roles.

```
 99   function _setRoleAccounts(bytes32 role, address manual, address service) internal {
100       _requireBusinessRole(role);
101       if (manual == address(0) || service == address(0)) revert InvalidRoleAccount();
102       _requireUnique(role, manual);
103       if (service != manual) _requireUnique(role, service);
104
105       RoleAccounts storage current = _roleAccounts[role];
106       address oldManual = current.manual;
107       address oldService = current.service;
108       if (oldManual != address(0)) _revokeRole(role, oldManual);
109       if (oldService != address(0) && oldService != oldManual) _revokeRole(role, oldService);
110
111       _grantRole(role, manual);
112       if (service != manual) _grantRole(role, service);
113       current.manual = manual;
114       current.service = service;
115       current.version += 1;
116       emit RoleAccountsUpdated(role, oldManual, manual, oldService, service, current.version);
117   }
118
119   function _requireUnique(bytes32 targetRole, address account) internal view {
120       bytes32[4] memory roles = _businessRoles();
121       for (uint256 i; i < roles.length; ++i) {
122           bytes32 role = roles[i];
123           if (role == targetRole) continue;
```

```
124             RoleAccounts storage other = _roleAccounts[role];
125             if (account == other.manual || account == other.service) revert
                    AddressUsedByAnotherRole(account, role);
126         }
127         if (account == defaultAdmin()) revert AddressUsedByAnotherRole(account, OWNER_ROLE);
128     }
```

Listing 2.3: src/access/ProsperAccessControllerUpgradeable.sol

Impact The separation of duty between administrative and operational accounts can be circumvented via valid on-chain administrative transitions, leading to role overlap and violating authority boundary controls.

Suggestion Revise the logic accordingly.

2.2 Recommendation

2.2.1 Handle permit failures caused by front-running

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `sellInnerWithPermit()`, the Router invokes function `permit()` before executing the sale. An EIP-2612 permit may be submitted separately before the user's transaction is processed, which consumes the nonce and establishes the intended allowance for the Router. In this case, the subsequent invocation of function `permit()` reverts because the signature references a consumed nonce. The function `sellInnerWithPermit()` therefore cannot proceed even when the Router already has sufficient allowance to transfer `pTokenIn`.

```
 86     function sellInnerWithPermit(
 87         address market,
 88         uint256 pTokenIn,
 89         uint256 minOut,
 90         uint256 deadline,
 91         bool nativeOut,
 92         uint8 v,
 93         bytes32 r,
 94         bytes32 s
 95     ) external nonReentrant returns (uint256 amountOut) {
 96         _checkActive();
 97         address token = IBondingCurveMarket(market).pToken();
 98         IERC20Permit(token).permit(msg.sender, address(this), pTokenIn, deadline, v, r, s);
 99         amountOut = _sellInner(market, pTokenIn, minOut, deadline, nativeOut);
100     }
```

Listing 2.4: src/router/ProsperTradeRouterUpgradeable.sol

Suggestion Wrap the invocation of function `permit()` in a try/catch block. If the invocation fails, continue only when the Router's existing allowance is at least `pTokenIn`; otherwise, revert the transaction.

2.2.2 Add initializer-disabling constructors to upgradeable implementation contracts

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The upgradeable implementation contracts inherit `Initializable` and expose external `initialize()` functions for proxy initialization. Their constructors do not invoke function `_disableInitializers()`, so the implementation contracts can also be initialized directly. Direct initialization does not modify the storage of an initialized proxy. However, it allows an external account to establish roles and initialization state in the implementation contract itself. This may create unintended behavior if assets are transferred to the implementation contract or if future upgrades introduce logic that depends on its local initialization state.

Suggestion Add a constructor to each upgradeable implementation contract that invokes function `_disableInitializers()` to prevent the implementation contract from being initialized directly.

2.2.3 Validate and enforce consistency between buyback router and migration venue

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Liquidity migration and subsequent `pToken` buybacks rely on DEX components that must be configured consistently. The migration process uses `v2Factory` to create or validate the canonical pair between the `pToken` and the wrapped native token, while each buyback proxy uses an independently configured `uniswapV2Router`.

During campaign creation, the protocol only verifies that the wrapped native token configured in the migration escrow matches the global configuration. It does not verify that the buyback router uses the same factory as the migration process, that the function `router.WETH()` returns the configured wrapped native token. It also does not check that the router supports the `FaroSwap` fee-array swap selectors. In addition, it does not verify that the `pToken` and wrapped native token pair resolved through the router matches the function `externalPair()`.

```
144       IMigrationRegistrar migration = IMigrationRegistrar(addresses_.migrationEscrow);
145       if (migration.wrappedNative() != addresses_.wrappedNative) revert InvalidConfig();
```

Listing 2.5: src/factory/ProsperFactoryUpgradeable.sol

```
53    function initialize(InitParams calldata params) external initializer {
54        if (
55            params.usdc == address(0) || params.wrappedNative == address(0) || params.pToken ==
                  address(0)
56                || params.market == address(0) || params.router == address(0)
57        ) revert ZeroAddress();
58        __ProsperManaged_init(params.accessController);
59        usdc = IERC20(params.usdc);
60        wrappedNative = IERC20(params.wrappedNative);
```

```
61        pToken = IBurnableERC20(params.pToken);
62        market = IBondingCurveMarket(params.market);
63        router = IUniV2Router(params.router);
64    }
```

Listing 2.6: src/buyback/BuybackUpgradeable.sol

Suggestion Revise the logic to ensure that the `buyback` router is bound to the migration venue during deployment and campaign creation.

2.2.4 Align the ERC-7201 annotation with runtime storage

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Contract `ProsperManagedUpgradeable` annotates struct `ManagedStorage` with the ERC-7201 namespace `prosper.storage.Managed`, which derives storage location `0xccd8f93a658f8533f645cdb11f0a8b069a58ace3d8f6430b4a249221b6f3b100`. However, function `_managedStorage()` reads from and writes to a different runtime storage slot defined by `MANAGED_STORAGE_LOCATION`: `0x812864b39be727b3ea73fc433566cdf99505d61f09e4cc1dfca4087a0b74d100`.

All current callers consistently access struct `ManagedStorage` through function `_managedStorage()`. Therefore, the mismatch does not currently cause a storage collision. However, the annotation does not reflect the actual runtime storage layout, which may mislead storage-layout tooling and future upgrade reviews. In addition, changing `MANAGED_STORAGE_LOCATION` to the slot derived from the existing annotation would prevent deployed proxy contracts from accessing the state stored at the current slot.

```
8   /// @custom:storage-location erc7201:prosper.storage.Managed
9   struct ManagedStorage {
10      IProsperAccessController accessController;
11  }
12
13  bytes32 private constant MANAGED_STORAGE_LOCATION =
14      0x812864b39be727b3ea73fc433566cdf99505d61f09e4cc1dfca4087a0b74d100;
```

Listing 2.7: src/access/ProsperManagedUpgradeable.sol

Suggestion Preserve `MANAGED_STORAGE_LOCATION` for existing deployments and update the ERC-7201 annotation so that its derived storage location matches the runtime slot. If the runtime slot must be changed, explicitly migrate the existing storage before adopting the new location.

2.2.5 Allow failure finalization after reward-pool burning

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In contract `MigrationEscrowUpgradeable`, function `finalizeFailure()` burns the migration allocation of a failed campaign, invokes function `burnRewardPool()` to burn the corresponding reward allocation, and then invokes function `markFailureAssetsFinalized()` on the associated Market contract to record the completion of failure-asset settlement. However, contract `PTokenClaimUpgradeable` also permits an account with `ROOT_SETTER_ROLE` to invoke function `burnRewardPool()` independently for the same campaign.

If function `burnRewardPool()` is invoked directly before function `finalizeFailure()`, the subsequent call from function `finalizeFailure()` reverts because flag `burned` has already been set to true. Since the transaction is atomic, the revert also rolls back the migration-allocation burn and all state updates previously performed by function `finalizeFailure()`. Consequently, function `finalizeFailure()` cannot complete the remaining settlement operations for that campaign.

```
194
195   function finalizeFailure(address vault)
196       external
197       nonReentrant
198       whenProsperModuleActive(keccak256("MODULE_MIGRATION"))
199       returns (uint256 migrationBurned, uint256 rewardBurned)
200   {
201       Campaign storage item = _campaigns[vault];
202       if (item.pToken == address(0) || item.finalized) revert InvalidState();
203       if (item.failed) return (0, 0);
204       if (IMigrationMarket(item.market).refreshPhase() != IBondingCurveMarket.Phase.
              FailedInnerOnly) {
205           revert InvalidState();
206       }
207       item.failed = true;
208       migrationBurned = item.pTokenAmount;
209       item.pTokenAmount = 0;
210       IMigrationPToken(item.pToken).burn(migrationBurned);
211       rewardBurned = rewardClaim.burnRewardPool(vault);
212       IMigrationMarket(item.market).markFailureAssetsFinalized();
213       emit FailedAllocationsBurned(vault, migrationBurned, rewardBurned);
214   }
```

Listing 2.8: src/migration/MigrationEscrowUpgradeable.sol

```
217
218   /// @notice Burns the entire reward pool when a launch failed or graduated with zero eligible
          users.
219   function burnRewardPool(address r25Vault)
220       external
221       whenProsperModuleActive(keccak256("MODULE_REWARD_ROOT"))
222       returns (uint256 amount)
223   {
224       if (
225           msg.sender != finalizer
226               && !IProsperAccessController(accessController()).hasRole(
227                   IProsperAccessController(accessController()).ROOT_SETTER_ROLE(), msg.sender
228               )
```

```
229         ) revert UnauthorizedRole(keccak256("PROTOCOL_FINALIZER"), msg.sender);
230         Campaign storage item = _campaigns[r25Vault];
231         if (item.pToken == address(0) || item.burned || item.enabled || item.totalClaimed != 0)
                revert InvalidState();
232         if (block.timestamp < IBondingCurveMarket(item.market).endsAt()) revert InvalidState();
233         IBondingCurveMarket.Phase current = IBondingCurveMarket(item.market).phase();
234         if (current == IBondingCurveMarket.Phase.ActiveInner) revert InvalidState();
235         item.burned = true;
236         amount = item.rewardPool + item.curatorRewardPool;
237         IBurnableClaimToken(item.pToken).burn(amount);
238         emit RewardPoolBurned(r25Vault, amount);
239     }
```

Listing 2.9: src/claim/PTokenClaimUpgradeable.sol

Suggestion Revise function `finalizeFailure()` to complete successfully when the reward pool has already been burned.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In this protocol, several privileged roles (e.g., `DEFAULT_ADMIN_ROLE` and `MIGRATOR_ROLE`) can conduct sensitive operations, which introduces potential centralization risks. For example, `LaunchConfig.lpLockDuration` determines how long the LP tokens deposited during migration remain locked in contract `LpTimeLock`. During campaign registration, the configured duration is passed through function `registerCampaign()`, while function `LpTimeLock.register()` only verifies that the value is greater than zero. Therefore, the documented ten-year lock period is not enforced on-chain. An incorrectly or maliciously configured duration could allow the LP tokens to be released significantly earlier than intended, after which liquidity could be removed from the external market. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.2 Reliance on trusted off-chain logic

Introduced by `Version 1`

Description The protocol relies on trusted off chain processes to determine the outcomes of several critical operations. In contract `PTokenClaimUpgradeable`, an account holding `ROOT_SETTER_ROLE` can set the Merkle root and total allocation for each campaign. The contract only verifies that a submitted claim is included in the configured root and does not verify how the eligible accounts or claim amounts were determined. The parameters used for migration and buyback operations are also determined off-chain and submitted by privileged roles for on chain execution. Therefore, the correctness of reward distributions, migrations, and buybacks depends on the accuracy and integrity of the corresponding off chain processes and authorized operators.

2.3.3 Reliance on timely migration execution

Introduced by Version 1

Description The transition from a graduated bonding curve market to the external market requires an account holding `MIGRATOR_ROLE` to manually invoke the migration functions. These functions initialize the external liquidity and finalize the campaign launch. Until the migration is completed, the campaign remains in the pending migration state and the external market is not activated. The protocol therefore relies on the authorized operator to monitor campaign status and complete the migration in a timely manner.

2.3.4 Trust assumptions on the external v2 factory

Introduced by Version 2

Description Function `migrateToExternal()` relies on an external v2 factory that is outside the audit scope. The availability of the migration process depends on this factory restricting canonical pair creation to the configured `MigrationEscrow`.

The canonical pair address is determined by the factory address, token addresses, fee rate, and init code hash. The `curator` address is not included in this derivation. Therefore, if the factory permits arbitrary pair creation, an external account could create the canonical pair before the migration transaction and assign a different curator address. Function `migrateToExternal()` would identify the existing pair but revert because the address returned by function `lpMtCurator()` does not match the expected curator. Since the canonical pair address is already occupied, the affected campaign could remain in phase `GraduatedPendingMigration` and be unable to complete migration.

The project documentation states that the factory only permits the configured `MigrationEscrow` to create pairs. The project should ensure that the factory deployed in production enforces this access restriction.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS