BAILSEC.IO
OFFICE@BAILSEC.IO
X: @BAILSECURITY
TG: @HELLOATBAILSEC

FINAL REPORT

Parallel

PRL Token

January 2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Parallel Protocol
Website	mimo.capital
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/parallel-protocol/PRL-token/tree/cee5e7bc82ee23a1e1817006f4565c3dcaef89b4/contracts https://github.com/parallel-protocol/tokenomics/tree/b8e0d3e373940d1b27c1c6f0e52c2e8810942919
Resolution 1	https://github.com/parallel-protocol/PRL-token/tree/0f35481fa57cd217066816fb4fa9318baf67d319 https://github.com/parallel-protocol/tokenomics/tree/e2b851520ae9c89524f8b6c3681374a37a19a9f5

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	5	1			1
Medium	5	4			
Low	8	4		3	
Informational	9	6		2	1
Governance					
Total	27	15		5	2

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

PRL-en/ layerZero contracts

The three contracts OFT, OFTAdpater, and OFTCore are forked from the layerZero contracts by the same names. These contracts handle the cross-chain component of the protocol and interact with the layerZero endpoints on the various chains. The fork used in this project contains a modification to remove the handling of varying decimals on the various chains. The original layerZero contracts allowed cross-chain tokens to have different decimals on different chains and had a system in place to convert the token amounts between the different decimals. That part has been removed so now the cross-chain conapprovalRequiredtracts operate with the same decimal on every chain. This makes the contracts more concise and simpler.

Issue_01	approvalRequired function missing from the OFT contracts
Severity	**Informational**
Description	OFT and OFTAdpater contracts implement an approvalRequired function. https://github.com/LayerZero-Labs/LayerZero-v2/blob/7da76840e41dc593d3c2007ce35b911b1d816b4b/packages/layerzero-v2/evm/oapp/contracts/oft/interfaces/IOFT.sol#L91-L98 According to a comment in the original OFT implementation, this function *@dev Allows things like wallet implementers to determine integration requirements, without understanding the underlying token implementation.* This function is missing in the forked OFT contracts.
Recommendations	Consider adding the approvalRequired function which returns false in the OFT contract and true in the OFTAdapter contract.

Comments / Resolution	Fixed following recommendation.

PRL-token/ PRL contracts

The PRL token itself is defined by the two contracts PRL and PeripheralPRL. The PRL contract is a normal ERC20 contract with permit capabilities deployed on mainnet while the PeripheralPRL contract is a layerZero OFT token deployed on the L2s/other chains. The PeripheralPRL contract has added functionality making it pausable, and able to handle permit-based actions.

Privileged Functions

- pause
- unpause

PRL-token / Lockcox

The LockBox contract exists on mainnet and is responsible for holding PRL tokens on the mainnet and serves as the enabler of cross-chain transfers of the PRL token. The mainnet PRL token does not have built-in cross-chain capabilities and is a simple ERC20Permit token. The LockBox contract is an OFTAdapter and thus has the ability to lock and unlock PRL tokens based on cross-chain messages. It can lock PRL tokens on mainnet and send a message to the layerZero endpoint asking to mint tokens on the destination chain and can receive messages from the endpoint to disburse tokens on the mainnet, allowing cross-chain transfers on a token with no built-in cross-chain capabilities. It also implements a pause mechanism and has additional functionality to allow permit-based operations.

Privileged Functions

- pause
- unpause

Issue_02	Permit-based calls can be made to revert by other users
Severity	**Low**
Description	The sendWithPermit function present in the LockBox and PeripheralPRL contracts and the depositPRLAndWeth and depositPRLAndEth functions in the SPRL2 contract do a permit call in the function. If this permit call fails, the entire transaction will revert. The issue is that when this transaction is in the mempool, the signature parameters for the permit call are visible to all. Thus any user can call permit on the token contract frontrunning this transaction and use up the signature. This will then cause the user's transaction to revert. *IERC20Permit(address(innerToken)).permit(msg.sender, address(this), _sendParam.amount, _deadline, _v, _r, _s);* *return super.send(_sendParam, _fee, _refundAddress);*
Recommendations	Consider using a try-catch for the permit call. This will allow the transaction to go through even if the permit signature is used up by a frontrunner.
Comments / Resolution	Fixed by moving the permit calls into try-catch statements.

PRL-token/ PeripheralMigrationContract

The system has 2 migration contracts, one for the mainnet and the other for all the other chains. PeripheralMigrationContract is the migration contract on the L2/side chains. Migration contracts allow users to swap their old MIMO tokens for the new PRL tokens. This is done via the migrateToPRL function, which is the main function of this contract. On a migration call, the MIMO tokens are transferred in and locked in this contract. Then a cross-chain message is sent via layerZero endpoints to the main migration contract on mainnet, which handles the handing out of the PRL tokens.

Privileged Functions

- emergencyRescue
- pause
- unpause

Issue_03	Incorrect fee calculation in quote function
Severity	**Low**
Description	In the PeripheralMigrationContract, the quote function is used to estimate the cross-chain transaction fees. However, the issue is that this function always assumes that the final chain is going to be mainnet and thus uses mainEid when combining options. *combineOptions(mainEid, _extraSendOptions.length > 0 ? SEND_AND_MIGRATE : SEND, _extraSendOptions);* This is different from the migrateToPRL function quote call, which uses the destEid. Thus if enforcedOptions are used, this can result in a quote different from the required one.
Recommendations	Switch mainEid with _dstEid in the combining step of the quote function.
Comments / Resolution	Fixed by switching mainEid with _dstEid when building options in the quote function.

Issue_04	msgReceipt.guid is emitted before it is generated
Severity	**Informational**
Description	The migrateToPRL function in the PeripheralMigrationContract contract emits the msgReceipt.guid. *emit MigrationMessageSent(msgReceipt.guid, _dstEid, msg.sender, _receiver, fee, _amount);* *msgReceipt = _lzSend(...* The issue is that it is emitted before the _lzSend call, which is when it is generated. So the emitted event has the default values and the emitted guid is always 0.
Recommendations	Emit the event after generating the receipt.
Comments / Resolution	Fixed by emitting event at the end of the function.

Issue_05	Permit functionality is not necessary in PeripheralPRL
Severity	**Informational**
Description	The PeripheralPRL contract implements a sendWithPermit function. This function's purpose is to do single transaction transfers by leveraging the permit functionality so that a separate approval transaction is not needed. The LockBox contract uses this correctly. However, this permit functionality is not needed in the case of the PeripheralPRL contract. This is because it inherits the OFT contract and not the OFTAdapter contract. The OFT contract directly burns tokens from the caller without any approval, unlike the OFTAdapter which transfers the tokens in, requiring approval.
Recommendations	Resolved. The sendWithPermit does not add any extra functionality to the PeripheralPRL contract.

Comments / Resolution	sendWithPermit removed.

PRL-token/ PrincipalMigrationContract

The PrincipalMigrationContract contract is the main migration contract of the system, which exists only on mainnet. This contract handles the mainnet as well as cross-chain migrations of MIMO tokens into PRL tokens. The contract is first credited a bunch of PRL tokens which will be used for the migration.

migrateToPRL

The migrateToPRL function is for mainnet migrations, which take MIMO tokens and give out PRL tokens.

migrateToPRLAndBridge

The migrateToPRLAndBridge function is for migrating from mainnet to other chains, which takes in MIMO tokens and sends a cross-chain message to the other chain via the LockBox contract, minting PeripheralPRL tokens to the end user.

_lzReceive

The _lzReceive function implements the logic handling cross-chain messages initiated by the PeripheralMigrationContract, which handles migrations from secondary chains to any chain. If the destination chain is mainnet, then the message is consumed and PRL tokens are given out to the receiver. Otherwise, another cross-chain message is sent out via the LockBox, and PeripheralPRL tokens are minted to the end-user on the destination chain.

Appendix: Migration token flows

Migration takes via one of three paths:

- mainnet -> mainnet
- mainnet -> sidechain
- sidechain -> mainnet
- sidechain -> sidechain

Mainnet -> Mainnet

In this migration, MIMO tokens are converted to PRL tokens all on mainnet. This happens via the migrateToPRL function in the PrincipalMigrationContract contract. MIMO tokens are transferred in from the user and PRL tokens present in the contract are sent out.

Mainnet -> Sidechain

In this migration, MIMO tokens from mainnet are converted to PRL tokens on any of the supported secondary chains where they are then credited to the receiver. This is done via the migrateToPRLAndBridge function in the PrincipalMigrationContract contract. MIMO tokens are transferred in from the user and the send function of the LockBox contract is called. This burns the PRL tokens from the migration contract and a cross-chain message is sent. This message is received on the destination chain by the PeripheralPRL contract which mints the PRL tokens to the receiver.

Sidechain -> Mainnet

In this migration, MIMO tokens from any chain are converted to PRL tokens which are credited to the receiver on mainnet. This happens via the migrateToPRL function in the PeripheralMigrationContract contract. MIMO tokens are transferred in and locked on the secondary chain. Then a cross-chain message is sent, which is received by the PrincipalMigrationContract on mainnet. This contract then sends out the PRL tokens to the receiver.

Sidechain -> Sidechain

In this migration, MIMO tokens on any chain are converted to PRL tokens on any chain. This also happens via the migrateToPRL function in the PeripheralMigrationContract contract. Similar to the flow above, MIMO tokens are locked in the secondary chain and a cross-chain message is sent, which is received by the PrincipalMigrationContract on mainnet. This contract then sends another cross-chain message via the LockBox contract’s send function to the destination chain. Hence MIMO tokens are locked in the origin chain, PRL token is burnt on mainnet and PeripheralPRL tokens are minted in the destination chain.

Privileged Functions

- emergencyRescue
- pause
- unpause

<table>
<tr><th>Issue_06</th><th>Users can drain eth from PrincipalMigrationContract</th></tr>
<tr><td>Severity</td><td>High</td></tr>
<tr><td>Description</td><td>The migrateToPRLAndBridge function allows a user to swap MIMO on mainnet for PRL tokens and then bridge them to a desired destination chain. However, the function can be made to use its own eth balance to pay for the layerZero bridging cost.
function migrateToPRLAndBridge(
        SendParam calldata _sendParam,
        MessagingFee calldata _fee
)
    {
lockBox.send{ value: _fee.nativeFee }(_sendParam, _fee, msg.sender)

_fee.nativeFee is used as the value for the lockBox send transaction instead of msg.value. Thus users can send insufficient eth for this transaction and have the contract pay for their transaction instead since the contract never checks if the user sent enough eth to cover fee.nativeFee.
Furthermore, if the contract has a significant amount of eth in the contract, users can even steal them by specifying a larger than necessary fee.nativeFee value. This will do the bridging and then refund all the excess gas to msg.sender, allowing the user to profit from the transaction.</td></tr>
<tr><td>Recommendations</td><td>Use msg.value for the send call</td></tr>
<tr><td>Comments / Resolution</td><td>Fixed following recommendation.</td></tr>
</table>

Issue_07	LayerZero Refund given to contract instead of user
Severity	**Medium**
Description	The PrincipalMigrationContract receives cross-chain messages during migration from other chains. If the destination is mainnet, PRL tokens are given out but if the destination is some other chain, then a cross-chain message is sent via the lockBox to credit PRL tokens on another chain. *lockBox.send{ value: msg.value }(sendParam, MessagingFee(msg.value, 0), payable(address(this)));* The last parameter is the refund recipient, which receives the gas refund in case the executor calls this function with more gas than is needed by the endpoint as fees. The issue is that this refund is kept by the contract itself and not passed on to the caller. Thus if the caller paid more in fees than the current cross-chain message price, they can lose out on refunds.
Recommendations	Consider refunding the excess gas to the receiver address instead of keeping it.
Comments / Resolution	Fixed by refunding to the receiver address.

Issue_08	Users cannot specify a refund recipient
Severity	**Informational**
Description	In the PrincipalMigrationContract contract, the function migrateToPRLAndBridge does not allow users to set a refund recipient for the layerZero call. *lockBox.send{ value: _fee.nativeFee }(_sendParam, _fee, msg.sender);* The refund is always sent to the msg.sender address and not to some other address if needed,
Recommendations	Consider allowing users to set the refund recipient in migrateToPRLAndBridge.
Comments / Resolution	Fixed by allowing a refundAddress field to the affected function.

Issue_09	Owner can mint/drain PRL tokens
Severity	**Informational**
Description	Both the PRL token migration contracts implement the emergencyRescue function, which allows the admin to take out any ERC20 tokens, including MIMO and PRL tokens. This poses a serious centralization risk to the system. Admins can take out MIMO tokens from the contract and re-migrate them into PRL tokens over and over essentially minting PeripheralPRL tokens on the sidechains. They can also directly take out MIMO and PRL tokens from the mainnet migration contract. In case an admin's private key ever gets compromised, the attacker has the ability to control governance as well.
Recommendations	Allow the owner to withdraw all tokens except MIMO tokens, or consider a timelock mechanism for the rescue function.
Comments / Resolution	Acknowledged.

Tokenomics/ Auctioneer

The Auctioneer is a permissionless automated swap contract. Fees are accrued into the contract from other fee-generating contracts in multiple different tokens. Every epoch a new auction starts with some pre-defined starting price, which drops over the course of the epoch. At any point in time, any user can call buy and pay the current price in a designated token and take all the fees in the contract.

Since the auction price decreases linearly and the fees are accrued gradually, at a certain point in time the net value of the accrued fees in various tokens will match the price at fair market rates which is when the buy is expected to be triggered. The contract expects the best execution price if lots of interested users/bots are competing to buy as soon as possible to lock in some profit.

The contract forwards the proceeds of the auction to the paymentReceiver address which is the MainFeeDistributor contract on mainnet or the SideChainFeeCollector on other chains.

Appendix: Auctioneer game theory

The Auctioneer contract is inspired by the Euler FeeFlowController contract. The system relies on bidders constantly monitoring the state of the contract to lock in the best price for the auctions.

Assume the protocol accrues fees from multiple sources. At epoch 0, the initial price is set to 100 USDC. Assume the protocol generates on average $20 worth of accrued fees per day.

After 5 days there are $100 worth of assets to claim. Each day the auction price decreases by 100/30 USDC, assuming an epoch duration of 30 days. So, after five days, the auction price is 100 - 5 * 100 / 30 = 83.333 USDC.

Potential bidders monitor the value of accrued fees and price the auction. They can then pay 83.333 USDC to collect $100 worth of various tokens and make a profit after selling the assets and accounting for transaction gas fees. Each bidder wants to wait as long as possible for the price to drop and for fees to rise to maximize their profit. But allowing any extra time after the auction becomes profitable risks other bidders from winning the auction and locking in their profit. Thus the auction is expected to close at maximal efficiency as long as there are multiple interested bidders.

Privileged Functions

- emergencyRescue
- updatePaymentToken
- updatePaymentReceiver
- updateEpochSettings
- pause
- unpause

Issue_10	Auctioneer only works well with gradual fees
Severity	**Low**
Description	The Auctioneer contract, inspired by the Euler fee-flow contract, implements an auction mechanism with a decreasing price to sell the accrued fee tokens for some target token. The idea is that the auction price drops linearly over time, and the accrued fee value increases linearly over time and at a certain instant, the values match. This is hypothetically the perfect execution point for the auction. In practice, executions happen a bit later to account for gas fees. The issue is that this model only works when fees grow gradually. If fees jump instantly, then the auction is not optimal anymore. This can be abused by users to recoup their fees. The current protocol implements a position opening fee of 0.2%, which is an instant fee and not gradual. Thus users can wait for the auction to almost settle, and then in the same transaction open a large position and buy the auction. Say the auction value and fee value were both at $100. The user can open a large $100k position and pay 0.2% as fees, which amounts to $200. This bumps up the accrued fee to $300. The user can then in the same transaction buy the auction and spend $100 to get $300 in the fee tokens. Thus the user was able to completely negate their position opening fees.

	During liquidations, the protocol also collects a liquidation fee, which is also not gradual and can be similarly gamed.
Recommendations	Make sure all the fee streams grow gradually over time for the best outcome, or be prepared for the resulting losses.
Comments / Resolution	Resolved. The Auctioneer contract has been removed and the revenue tokens will be swapped via OTC swaps.

Issue_11	Auctioneer contract does not implement a deadline parameter
Severity	**Low**
Description	The auction contract allows users to bid for fee tokens in exchange for PAR tokens. It runs a Dutch auction till the price of the auction reaches 0. The issue is that this function does not include a deadline parameter. Thus users who initiated a buy call, in case their buy transaction takes a long time to mine due to network load, will be forced to make the purchase even if they do not want to anymore. Since the auction price only decreases, the impact is limited as the user will pay a lower amount to purchase the fee tokens for PAR tokens.
Recommendations	Consider allowing a deadline parameter in the buy function.
Comments / Resolution	Resolved. The Auctioneer contract has been removed and the revenue tokens will be swapped via OTC swaps.

Tokenomics/ SideChainFeeCollector

The SideChainFeeCollector contract collects the proceeds from the auctions happening on the sidechains. It has one main function, release, which takes the tokens and sends them to the MainFeeDistributor contract on the mainnet.

Privileged Functions

- updateDestinationReceiver
- updateBridgeableToken
- emergencyRescue
- pause
- unpause

Issue_12	Users can grief fee transfers
Severity	**Medium**
Description	The SideChainFeeCollector contract's release function can be used by any user to bridge the collected fees to mainnet. *function release(bytes memory _options)* The issue is that the user is free to specify any _options. The options passed in are used by the layerZero executor to call lzReceive on the mainnet. So if the user specifies a very low gasLimit in these options, the executor call on mainnet can fail. In that case, someone needs to manually replay the lzReceive transaction on mainnet to recover the funds. Allowing users to specify any _options parameters allows any user to temporarily grief the protocol by having the funds be stuck in the bridge awaiting manual recovery.
Recommendations	Consider using an admin-coded options value, or making the release function restricted

Comments / Resolution	Fixed by making the release function restricted.

Tokenomics/ MainFeeDistributor

The MainFeeDistributor contract receives the auction proceeds from all the SideChainFeeCollector contracts present on the various side chains. The contract might be holding some OFT PAR tokens due to how the bridging works with the auction proceeds, so the contract can also swap the OFT tokens for the actual PAR tokens via the swapLzToken function. The collected funds can also be distributed to the various fee receivers according to their share ratios.

Privileged Functions

- updateFeeReceivers
- updateBridgeableToken
- emergencyRescue
- pause
- unpause

Issue_13	Outdated function calls in MainFeeDistributor contract
Severity	**High**
Description	In the MainFeeDistributor contract, the swapLzToken function swaps OFT tokens for PAR tokens after checking the available swap amount *uint256 maxSwapAmount = IBridgeableToken(address(bridgeableToken)).getMaxMintableAmount(); IBridgeableToken(address(bridgeableToken)).swapLzTokenToPrincipalToken(swapAmount);* The issue is that in the latest version of the BridgableToken contract, the getMaxMintableAmount function was renamed to getMaxCreditableAmount and the swapLzTokenToPrincipalToken parameter also takes a to field. This is reflected in the code here: https://github.com/parallel-protocol/bridging-module/blob/audit/bailsec-december-2024/contracts/tokens/BridgeableToken.sol Thus this function is outdated.
Recommendations	Update the function name to match the latest version of BridgableToken.
Comments / Resolution	Failed Resolution. The swapLzTokenToPrincipalToken function call is still missing the to parameter.

Tokenomics/ RewardMerkleDistributor

Stakers in the SPRL vault contracts are eligible for rewards. These rewards are handled by the RewardMerkleDistributor contract. This contract is funded with reward tokens and for every epoch, a Merkle tree is created with the hashed combination of epoch, account address, and the amount of rewards as the leaves of the tree. For every epoch, the root of this tree is stored in the contract via the updateMerkleDrop function.

Users expecting rewards can then call the claims function with valid Merkle proofs to receive their rewards. Claims are only valid up to the expiry timestamp of the tree for that epoch. After expiry, any pending rewards can be forwarded to a recipient address via the forwardExpiredRewards function.

Privileged Functions

- updateMerkleDrop
- emergencyRescue
- updateExpiredRewardsRecipient
- pause
- unpause

Issue_14	Inconsistent merkleDrop expiry check
Severity	**Medium**
Description	In the RewardMerkleDistributor contract, the _claim function checks the expiry timestamp against the current timestamp *if (currentTimetamp > _merkleDrop.expiryTime) revert EpochExpired();* The _getEpochExpiredRewards also checks for expiry before forwarding expired rewards to the designated receiver contract *if (_merkleDrop.expiryTime > uint64(block.timestamp)) revert EpochNotExpired();* The issue is that at timestamp == expiry, the above check claims that the rewards are not expired while the lower check claims that the rewards are expired. So at this timestamp, rewards can both be claimed and sent off as expired. This check is inconsistent and should be fixed.
Recommendations	Change the _getEpochExpiredRewards function expiry check to >=, so that rewards cannot be expired when timestamp==expiry.

Comments / Resolution	Fixed following recommendation.

Issue_15	No solvency check in updateMerkleDrop function
Severity	**Informational**
Description	The updateMerkleDrop function in the RewardMerkleDistributor contract updates the Merkle root for the rewards to be distributed for a particular epoch. However, there is no check to see if the contract has enough funds to pay out all those rewards in the first place. If the contract is insolvent, then early claimers will deny late claimers from receiving any rewards.
Recommendations	Consider adding a check for solvency in the updateMerkleDrop function.
Comments / Resolution	Failed Resolution. Solvency check does not consider rewards from previous epochs that are still not expired. The complete fix would require a pendingRewards variable which is incremented on updateMerkleDrop calls and decremented on _claim and forwardExpiredRewards calls. This should be checked against the current token balance of the Merkle claim contract.

Issue_16	Merkle leaves should be double-hashed
Severity	**Informational**
Description	Merkle tree leaves should be double-hashed to prevent any chance of second-preimage attacks. This is generally a problem if the hash input for a leaf is 2x the size of the hash of the leaf since in that case intermediate hash values can also be used as leaves. In the current implementation of the contracts, the encoded string is 60 bytes, which is 4 short of the required 2x32 bytes needed to trigger the issue. *abi.encodePacked(_epochId, _account, _amount)* A simple way to eliminate any chance of a second pre-image attack is to simply double hash the leaves since that means different hashing algorithms are being used for calculating leaves and calculating the Merkle proof validity. Even though this contract is not vulnerable, it is still good practice to double-hash the leaves. More info about the attack can be found here: https://www.rareskills.io/post/merkle-tree-second-preimage-attack
Recommendations	Consider hashing twice when calculating the leaf hash value.
Comments / Resolution	Fixed by double-hashing the leaves.

Tokenomics/ TimeLockPenaltyERC20

The TimeLockPenaltyERC20 contract implements a timelock mechanism for ERC20 tokens. Normal ERC20 tokens can be deposited into the contract with an admin-defined timeLockDuration value. When withdrawing, users call requestWithdraw which burns their receipt tokens and schedules a withdrawal which takes place timeLockDuration seconds from the current timestamp. Users can withdraw at any time by paying a fee, which gradually drops to 0 after the lock duration.

Privileged Functions

- updateTimeLockDuration
- updateStartPenaltyPercentage
- updateFeeReceiver
- pause
- unpause

Issue_17	Inconsistent withdrawals during emergencies
Severity	**Medium**
Description	During emergencies, the contract can be paused and users can take out their funds by calling the emergencyWithdraw function. This allows instant withdrawals with no penalties. The issue is that if a user requested a withdrawal before the contract was paused, they cannot use the emergencyWithdraw function anymore, since their tokens were already burnt. They will have to wait the entire timelock duration to withdraw without any penalty. They also cannot cancel their withdrawal, since the cancelWithdrawalRequests function has the whenNotPaused modifier. So, when an emergency situation occurs, users with funds in the system can immediately remove them. However, users with pending withdrawals are stuck until their timelock duration is over and don't have the option to cancel their withdrawal and then use the emergency withdrawal mechanism. Thus users get unfairly penalized if they have pending withdrawals.
Recommendations	Consider removing the withdrawal penalty in an emergency. Another option is to remove the whenNotPaused modifier from the cancelWithdrawalRequests function.
Comments / Resolution	Fixed by removing the whenNotPaused modifier from the cancelWithdrawalRequests function. Users can cancel their withdrawal requests in an emergency and use the emergency withdrawal mechanism if needed.

Issue_18	Missing 0 address checks
Severity	**Informational**
Description	The TimeLockPenaltyERC20 contract, as well as a lot of the other contracts in the system, set a lot of addresses in their constructors. These do not check for 0 addresses.
Recommendations	Consider adding 0 address checks in the constructors.
Comments / Resolution	Acknowledged.

Tokenomics/ sPRL1

The sPRL1 contract is an implementation of the TimeLockPenaltyERC20 for staking PRL tokens. Users can directly stake PRL tokens to this contract and expect rewards from the RewardMerkleDistributor contract. Users are subject to a timelock, preventing instant withdrawals. This is 1 of the 2 staking options available to the users.

Tokenomics/ sPRL2

The sPRL2 contract also inherits the TimeLockPenaltyERC20 contract and is used for staking PRL-WETH liquidity. PRL and WETH tokens are added to a Balancer 20% WETH 80% PRL pool and the liquidity tokens are staked on Aura protocol to get further rewards. Users can either stake/withdraw the underlying Aura tokens or directly operate with PRL and WETH tokens. By participating in this staking users receive rewards through the RewardMerkleDistributor contract and all the swap fees and extra rewards generated through Aura are sent to the feeReceiver contract.

Appendix: Balancer liquidity

The sPRL2 contract adds liquidity to Balancer. Balancer is a weighted constant product automated market maker, where tokens can be supplied in any ratio. In exchange, the balancer protocol gives out LP-BPT ERC20 tokens representing their liquidity share. This allows users to earn swap fees.

$$\text{PRL + WETH} \xrightarrow{\text{Balancer}} \text{LP-BPT tokens}$$

These LP-BPT tokens can then be deposited into Aura, which mints the user auraBPT tokens. This can be done via the Aura protocol's Booster contract.

$$\text{LP-BPT Tokens} \xrightarrow{\text{Aura}} \text{AuraBPT tokens}$$

These AuraBPT tokens can then be staked in the rewards pool to earn rewards. This can be done by manually staking in the rewards contract or by setting *stake=true* in the Booster contract when depositing. This removes the AuraBPT tokens and mints stakedAuraBPT tokens to the user.

$$\text{AuraBPT Tokens} \xrightarrow{\text{Aura staking}} \text{stakedAuraBPT Tokens}$$

The sPRL2 contract here calls Booster with stake=true, so the contract ultimately gets stakedAuraBPT Tokens back from Aura. Linked below are some etherscan contracts through which the architecture is implemented.

Booster contract, responsible for taking LP-BPT tokens and minting AuraBPT tokens:
https://etherscan.io/address/0xA57b8d98dAE62B26Ec3bcC4a365338157060B234#code

Rewards contract, responsible for taking AuraBPT tokens and minting stakedAuraBPT tokens:
https://etherscan.io/address/0x4B9f8F3cA7443f1ebcd959D9Bf169a4F03f12eaF#code

In the sPRL2 contract, the Booster contract address is stored in the AURA_BOOSTER_LITE variable and the Rewards contract address is stored in the AURA_VAULT variable.

Appendix: Fork testing

Since the contracts integrate with Balancer and Aura protocols, the only way to completely verify the functionality of the sPRL2 vault is via fork tests. This is because both AURA and BalancerV3 vaults are complex and have several different underlying tokens, staking, and reward-handling flows.

Since new aura vault registrations are a manual process, it can be tricky to set up fork testing. For this purpose, an existing vault can be used as a proxy which fits the following criteria:

- Should be a Balancer V3 vault
- Underlying tokens should be 18 decimals
- Composition of 80-20

After some research, turns out the following pool is a good candidate:
https://app.aura.finance/#/1/pool/244

Its an 80-20 pool with a V3 vault underneath it. pxETH can be treated like PRL token and gtWETHe can be treated like the WETH token and a forking setup can be drafted with these settings. This allows the contracts to interact with the on-chain balancer and Aura vaults without the need to create a new setup for PRL/WETH.

The reason this is required is that it wasn't immediately obvious that BalancerV3 pools require permit approvals and required a deep dive into the protocol after a fork test attempt to discover this. Thus a complete end-to-end test with these proxies will be able to root out any possible issues regarding the functionality of the vault.

Below is a snippet showing the setup of the fork test. It contains the relevant contract addresses for the previously mentioned pxETH-gtWETHe vault and thus can be used as a starting point for the fork tests. PRL tracks pxETH and WETH tracks gtWETHe. This way the sprl2 contract is supplying pxETH-gtWETHe which is functionally the same as creating a new vault and supplying PRL-WETH.

This requires changing the Aura vault ID in the sPRL2 contract to 244 instead of 19. Also, the evm_version of the project needs to be upgraded to cancun since BalancerV3 uses transient storage slots.

```
vm.createSelectFork("");
auraBpt = ERC20Mock(0x473dA6619e3bf97f946C6Cc991952c010e25eC3E);
balancerV3RouterMock =
BalancerV3RouterMock(0x5C6fb490BDFD3246EB0bB062c168DeCAF4bD9FDd);
auraBoosterLiteMock =
AuraBoosterLiteMock(0xA57b8d98dAE62B26Ec3bcC4a365338157060B234);
```

```
auraRewardPoolMock =
AuraRewardPoolMock(0x473dA6619e3bf97f946C6Cc991952c010e25eC3E);
bpt = ERC20Mock(0x5512fdDC40842b257e2A7742Be3DaDcf31574d53);
prl = ERC20Mock(0x04C154b66CB340F3Ae24111CC767e0184Ed00Cc6);
weth = WrappedNativeMock(0x1e6ffa4e9F63d10B8820A3ab52566Af881Dab53c);
sprl2 = _deploySPRL2(
        address(auraBpt),
        users.daoTreasury.addr,
        address(accessManager),
        DEFAULT_PENALTY_PERCENTAGE,
        DEFAULT_TIME_LOCK_DURATION,
        balancerV3RouterMock,
        auraBoosterLiteMock,
        auraRewardPoolMock,
        bpt,
        prl,
        weth
    );
```

Issue_19	sPRL2 approval mechanism is not compatible with BalancerV3
Severity	**High**
Description	When joining or exiting a balancer pool, the contract attempts to first simply approve the BALANCER_ROUTER. It is assumed the router will do a transferFrom. */// @dev Approve tokens.* *PRL.approve(address(BALANCER_ROUTER), _maxPrlAmount);* *WETH.approve(address(BALANCER_ROUTER), _maxEthAmount);* However, BalancerV3 uses permit2 for transfers. Thus this approval mechanism will not work for BalancerV3 pools and render the sPRL2 contract unusable. The use of permit2 for transfers can be observed in the snippet below. https://github.com/balancer/balancer-v3-monorepo/blob/767a6a137be78bf7b6bb67b8ff423f53ef60939c/pkg/vault/contracts/Router.sol#L315-L319
Recommendations	Change the approval mechanism to permit2, in order to correctly integrate with BalancerV3 pools. Consider implementing a fork test to ensure the correctness of the contract. Consult the fork testing appendix section above for pointers for an easy test setup.
Comments / Resolution	

Issue_20	sPRL2 does not unlock balancerV3 vault before addliquidity and removeLiquidity operations
Severity	**High**
Description	BalancerV3 vault functions cannot be called directly, they must be wrapped in an unlock call (to unlock the vault) or accessed via a router. https://github.com/balancer/balancer-v3-monorepo/blob/main/pkg/vault/contracts/Vault.sol#L133 The addLiquidity and removeLiquidity functions have the onlyWhenUnlocked modifier, which means the functions can only be called during an unlock callback. This means the sPRL functions that internally use _joinPool and _exitPool are effectively broken.
Recommendations	Use the Router contract to interact with Balancer. Otherwise, callbacks need to be implemented in the sPRL2 contract.
Comments / Resolution	To be deliverd. Due to the integration error from another high-severity issue, it is not possible to test if the fix is sufficient and error-free.

Issue_21	sPRL2 withdrawPRLAndWeth is broken
Severity	**High**
Description	As explained in the Balancer liquidity appendix above, the contract stakes LP-BPT tokens and gets back stakedAuraBPT tokens. This is because the deposit sets stake to true. *if (!AURA_BOOSTER_LITE.deposit(AURA_POOL_PID, bptAmount, true)) revert DepositFailed();* The issue is that once staked, the BoosterLite contract cannot be used to withdraw. This is because the BoosterLite contract requires AuraBPT tokens, which are not available anymore since the staking converted them to stakedAuraBPT tokens. This is also mentioned in a comment above the withdraw function in the BoosterLite contract present here: https://etherscan.io/address/0xA57b8d98dAE62B26Ec3bcC4a365338157060B234#code *"Withdraw a given amount from a pool (must already been unstaked from the Convex Reward Pool - BaseRewardPool uses withdrawAndUnwrap to get around this)"* So the BoosterLite contract cannot be used to withdraw if the tokens are staked. Instead, the withdrawAndUnwrap function on the Reward pool contract needs to be used. There are also multiple comments in the sPRL2 contract that mention the underlying as the auraBPT token. This is also incorrect since the staking converts the auraBPT tokens into stakedAuraBPT Tokens, which are incompatible with the BoosterLite contract's withdrawal function.
Recommendations	Withdraw by calling AURA_VAULT.withdrawAndUnwrap()

Comments / Resolution	To be deliverd. Due to the integration error from another high-severity issue, it is not possible to test if the fix is sufficient and error-free.

Issue_22	PRL or WETH as extra rewards on Aura can be stolen
Severity	**Medium**
Description	Aura pools allow operators to incentivize with extra rewards. The RewardFactory calls the addExtraReward function in the Aura Vault contract to add any ERC20 token as a reward. These tokens are then dripped out similar to the Synthetix staking mechanism. The issue is that the pool can be incentivized with PRL or WETH tokens as well. In that case, these rewards can be stolen. This is because any user can call getReward(address _account) on the Aura vault and have the rewards transferred to the sPRL2 contract. Then, during deposits, all remaining PRL and WETH tokens are transferred to the caller. *uint256 prlBalanceToReturn = PRL.balanceOf(address(this));* *if (prlBalanceToReturn > 0) {* *PRL.transfer(msg.sender, prlBalanceToReturn);* *}* So by triggering the rewards by interacting with the vault directly and then doing a deposit, users can hijack the reward tokens in the contract if the reward tokens happen to be PRL or WETH tokens.
Recommendations	Don't incentivize the Aura vault with PRL or WETH tokens. If PRL tokens are to be used as incentives, then during deposits the initial balance needs to be recorded and only the user’s excess needs to be repaid, not all tokens in the contract.
Comments / Resolution	

Issue_23	Hardcoded aura pool ID
Severity	**Low**
Description	The sPRL2 contract features a hardcoded aura pool ID. *uint256 public constant AURA_POOL_PID = 19;* The issue is that this poolID is already in use on mainnet and on other chains. This needs to be changed before deployment.
Recommendations	Consider saving the poolID in an immutable variable which will be set in the constructor instead of having it hardcoded.
Comments / Resolution	Acknowledged.

Issue_24	Missing unlockingAssets update in withdrawPRLAndWeth
Severity	**Low**
Description	The unlockingAssets value in the TimeLockPenaltyERC20 is used to keep track of the amount of underlying that is pending withdrawal. This value is updated whenever a withdrawal takes place via the withdraw function. *unlockingAssets = unlockingAssets - totalAmountWithdrawn - totalFeeAmount;* The issue is that the sPRL2 contract introduced another way to withdraw, the withdrawPRLAndWeth function. This function serves the same purpose but doesn't modify the unlockingAssets value, which will therefore show incorrect pending withdrawal amounts.
Recommendations	Update the unlockingAssets variable in the withdrawPRLAndWeth function.

Comments / Resolution	Acknowledged. The unlockingAssets value is still not updated in the withdrawPRLAndWeth and withdrawBPT functions.

Issue_25	sPRL2 can only be deployed on mainnet
Severity	**Low**
Description	The team shared that their intention is to deploy the sPRL contracts first to mainnet and then to other chains. The issue is that the sPRL2 contract integrates with BalancerV3 pools, which right now are only available on mainnet and Gnosis chains. This is because V2 pools use the joinPool function but V3 uses the addLiquidity function, which is what is used here.
Recommendations	Balancer V3 is not present on most chains. sPRL2 contracts on other chains need to use V2 vaults.
Comments / Resolution	Acknowledged.

Issue_26	evm_version needs to be updated to cancun
Severity	**Low**
Description	The sPRL2 contract is integrated with balancerV3 which uses transient storage. The problem is that transient storage was introduced in the cancun update. The current evm_version is set to shanghai. End to end tests cannot be written/work correctly without updating the evm_version to cancun.
Recommendations	Consider updating the evm_version to cancun.
Comments / Resolution	

Issue_27	Hanging approvals in sPRL2
Severity	**Informational**
Description	The _joinPool function in the SPRL2 contract gives approval for the tokens to be used by Balancer. *PRL.approve(address(BALANCER_VAULT), _maxPrlAmount);* *WETH.approve(address(BALANCER_VAULT), _maxEthAmount);* This is the maximum amount to be used, so the entire amount may be not used up in the actual addLiqudity call. In that case, there will be pending approval amounts left in the contract, since the approval is not zeroed out at the end of the liquidity addition. It is considered best practice to first zero out the approval before granting another.
Recommendations	Consider zeroing out the approval either at the end of the joinPool function or before granting new approvals.
Comments / Resolution	Fixed by setting approval to 0 after adding liquidity.